Secure Semantic Web Data Management

Confidentiality, Integrity, and Compliant Availability in Open and Distributed Networks

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte

Dissertation

vorgelegt von

Andreas Kasten

Vorsitzender des Promotionsausschusses
Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau

Vorsitzender der Promotionskommission
Prof. Dr. Steffen Staab, Universität Koblenz-Landau

Berichterstatter
Prof. Dr. Rüdiger Grimm, Universität Koblenz-Landau
Prof. Dr. habil. Ansgar Scherp,
Universität Kiel und Leibniz-Informationszentrum Wirtschaft

Datum der wissenschaftlichen Aussprache
11.11.2016

Datum der Einreichung
30.05.2016

Abstract

Confidentiality, integrity, and availability are often listed as the three major requirements for achieving data security and are collectively referred to as the C-I-A triad. Confidentiality of data restricts the data access to authorized parties only, integrity means that the data can only be modified by authorized parties, and availability states that the data must always be accessible when requested. Although these requirements are relevant for any computer system, they are especially important in open and distributed networks. Such networks are able to store large amounts of data without having a single entity in control of ensuring the data's security. The Semantic Web applies to these characteristics as well as it aims at creating a global and decentralized network of machine-readable data. Ensuring the confidentiality, integrity, and availability of this data is therefore also important and must be achieved by corresponding security mechanisms. However, the current reference architecture of the Semantic Web does not define any particular security mechanism yet which implements these requirements. Instead, it only contains a rather abstract representation of security.

This thesis fills this gap by introducing three different security mechanisms for each of the identified security requirements confidentiality, integrity, and availability of Semantic Web data. The mechanisms are not restricted to the very basics of implementing each of the requirements and provide additional features as well. Confidentiality is usually achieved with data encryption. This thesis not only provides an approach for encrypting Semantic Web data, it also allows to search in the resulting ciphertext data without decrypting it first. Integrity of data is typically implemented with digital signatures. Instead of defining a single signature algorithm, this thesis defines a formal framework for signing arbitrary Semantic Web graphs which can be configured with various algorithms to achieve different features. Availability is generally supported by redundant data storage. This thesis expands the classical definition of availability to compliant availability which means that data must only be available as long as the access request complies with a set of predefined policies. This requirement is implemented with a modular and extensible policy language for regulating information flow control. This thesis presents each of these three security mechanisms in detail, evaluates them against a set of requirements, and compares them with the state of the art and related work.

Zusammenfassung

Als wichtigste Anforderungen an Datensicherheit werden oft Vertraulichkeit, Integrität und Verfügbarkeit genannt. Vertraulichkeit von Daten bedeutet, dass nur berechtigte Parteien auf sie zugreifen können. Datenintegrität erfordert, dass nur berechtigte Parteien die Daten ändern dürfen. Verfügbarkeit von Daten bedeutet, dass auf die Daten jederzeit bei Bedarf zugegriffen werden kann. Obgleich die Umsetzung dieser Sicherheitsanforderungen für jedes Computersystem relevant ist, gilt dies insbesondere bei offenen und verteilten Netzen. Solche Netze speichern große Mengen an Daten ohne eine zentrale Instanz, die den sicheren Zugriff und die sichere Verarbeitung der Daten steuert. Das Semantic Web teilt diese grundlegende Eigenschaft, da es das Erstellen eines globalen und dezentralen Netzes von maschinenlesbaren Daten anstrebt. Auch für solche Daten muss daher durch entsprechende Sicherheitsmaßnahmen die Vertraulichkeit, Integrität und Verfügbarkeit garantiert werden können. Obgleich die aktuelle Referenzarchitektur des Semantic Webs durchaus das Umsetzen von Sicherheitsmaßnahmen vorsieht, definiert sie selbst noch keine konkreten Maßnahmen. Stattdessen enthält sie lediglich einen abstrakten Baustein, dem solche Maßnahmen zugeordnet werden können.

Diese Arbeit stellt drei konkrete Sicherheitsmaßnahmen vor, welche sich in die Referenzarchitektur des Semantic Webs einbetten lassen und die Anforderungen an die Vertraulichkeit, Integrität und Verfügbarkeit der Semantic-Web-Daten umsetzen. Die einzelnen Maßnahmen setzen die Anforderungen dabei nicht minimalistisch um, sondern bieten zugleich noch weiterführende Funktionen. Vertraulichkeit von Daten wird üblicherweise durch Datenverschlüsselung umgesetzt. Diese Arbeit stellt einen Ansatz zum Verschlüsseln von Semantic-Web-Daten vor, der zugleich auch ein Suchen auf den verschlüsselten Daten ohne vorhergehende Entschlüsselung erlaubt. Datenintegrität wird meist durch digitale Signaturen sichergestellt. Diese Arbeit definiert ein formales Rahmenwerk zum Signieren beliebiger Semantic-Web-Daten, welches mit verschiedenen Algorithmen flexibel konfiguriert werden kann. Verfügbarkeit wird oft durch eine redundante Datenhaltung garantiert. In dieser Arbeit wird eine Erweiterung der klassischen Definition von Verfügbarkeit verwendet, die als konforme Verfügbarkeit bezeichnet wird. Konforme Verfügbarkeit bedeutet, dass Daten nur dann bei einem Zugriffsversuch verfügbar sein müssen, wenn der Zugriff konform ist zu einem vordefinierten Regelsatz. Diese Sicherheitsanforderung wird umgesetzt durch eine modulare und erweiterbare formale Sprache zum Beschreiben von Regelsätzen zur Steuerung von Informationsflüssen. Jede der drei Sicherheitsmaßnahmen wird in dieser Arbeit im Detail beschrieben, anhand definierter Anforderungen evaluiert und mit verwandten Arbeiten verglichen.

Acknowledgments

Although a PhD thesis should officially be the work of the one person writing the thesis, it is completely unrealistic to avoid any support of other people. Quite the contrary, not asking for any kind of help and trying to solve all problems alone even contradicts with the basic principles of scientific work which involves communication and interaction with other researchers. As I do not want to offend anyone by not listing her or him on this page, I will use a brute-force approach by simply listing all people who have helped me in the past years with at least something related to this work. In order to avoid a large and confusing list of names, all persons listed below are grouped according to their type of assistance. First of all, I would like to thank my supervisors Ansgar Scherp and Rüdiger Grimm for their extensive support on this thesis and on any related scientific activities. This includes paying the conference fees and the trip to the respective locations as well. I would also like to thank all former co-workers of the University of Koblenz for giving various types of advice and for enduring my annoying personality. As most of the software and some related artifacts mentioned in this thesis were developed by students of this university, I would like to thank them as well. In addition to the developed software, they also provided helpful suggestions for improving the different security mechanisms presented in this thesis. Finally, I have to thank all people who participated in the time-consuming and tedious proofreading process. The different categories of thankfulness and their respective participants are listed below, sorted in alphabetical order by last name. Multiple categories per person are possible.

Supervisors

Rüdiger Grimm and Ansgar Scherp

Former co-workers

Katharina Bräunlich, Nico Jahn, Helge Hundacker, Brigitte Jung, Marco Krause, Daniel Pähler, Daniela Simić-Draws, Stefan Stein, and Tim Wambach

University students

Alexander Balke, Stefan Becker, Felix Gorbulski, Katharina Großer (née Naujokat), Benjamin Hück, Michael Kornas, Dominik Mosen, Michael Ruster, Peter Schauß, Artur Schens, Erwin Schens, Arne Fritjof Schmeiser, Johann Tissen, and Rainer Weißenfels

Proofreaders

Katharina Bräunlich, Nicole Köhler, Katharina Krause, Marco Krause, Johannes Siebel, Daniela Simić-Draws, and Tim Wambach

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1.

Secure Data Management in the Semantic Web

The Data Management Association (DAMA) defines data management as the "development, execution, and supervision of plans, policies, programs, projects, processes, practices and procedures that control, protect, deliver, and enhance the value of data and information assets" [207]. According to this definition, data management comprises all steps of processing data including its initial creation, storage, and usage. In order to implement these steps, DAMA defines a data management framework [207] consisting of ten basic components, each of which covers a different aspect of managing data. One of these components is data security management which aims at implementing the security requirements confidentiality and integrity of data as well as protection against unauthorized data access [208]. Confidentiality of data means that only authorized parties are aware of the data's existence and are able to see its contents [37]. Integrity of data means that any unauthorized modification of the data must be prohibited [37]. Protection against unauthorized data access prevents a party from accessing the data in such a way the party is not allowed to. Although general definitions of computer security [37] also focus on confidentiality and integrity, they do not include protection against unwanted data access as one of the main security requirements. Instead, this requirement is often replaced by availability of data. Availability means that data must always be accessible to any requesting party [28]. The three security requirements confidentiality, integrity, and availability of data are often collectively referred to as the *C-I-A triad* [28].

Although DAMA is mainly concerned with data management in closed environments such as organizations [207], its basic principles of data management can also be transferred to open and distributed networks [288] such as the Internet. The Semantic Web relies on these characteristics of the Internet as its own design is inherently open and decentralized as well. The aim of the Semantic Web is to create a global network of machine-readable data [35] by interlinking various distributed data sources [39]. Each data source provides a different type of data, including both publicly available data such as media [187, 240], life science [30], and e-government [134] as well as sensitive private data like medical records [256, 106] and information about business processes [143]. As the application areas and the size of the Semantic Web are still increasing, securing and protecting the data stored in the Semantic Web is also gaining more importance [185].

Since the Semantic Web is essentially a large collection of data, DAMA's security requirements for data management can be applied to it as well.

The architecture of the Semantic Web is often depicted as the Semantic Web layer cake [31, 32, 33, 47] as shown in Figure 1.1. Although not officially published, the layer cake serves as the current reference architecture when implementing Semantic Web applications [128]. It divides existing Semantic Web technologies into disjoint layers and depicts their interdependencies. The bottom layers describe the basic encoding of Semantic Web data which is usually represented by using RDF graphs [275] as data format. This data format is accompanied by data models, enriched by additional logic, and integrated into an application domain which is covered in the top layer. The layer cake also contains a *crypto* layer which comprises different cryptographic operations such as digital signatures or data encryption. Although cryptographic operations can be used to implement particular security requirements such as confidentiality and integrity of data, not all security mechanisms are based on cryptographic operations. For example, solutions for access control which protect Semantic Web data against unauthorized access have already been proposed [239, 247, 2, 164]. As access control does not necessarily require cryptographic operations, these solutions cannot be mapped to the Semantic Web layer cake depicted in Figure 1.1. Even alternative versions of the layer cake [141, 153, 127] do not provide additional security layers. As the *crypto* layer does not suggest any particular security mechanisms for protecting Semantic Web data, the current architectural representation of the Semantic Web can be considered as incomplete regarding security.

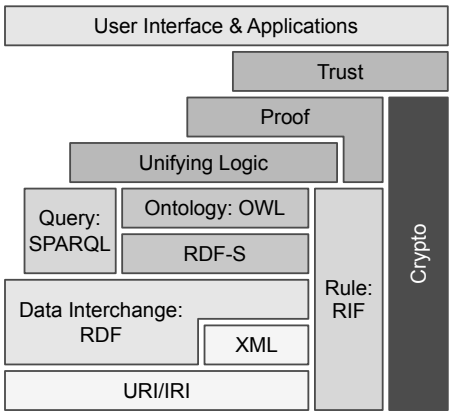


Figure 1.1.: The Semantic Web layer cake [47].

This thesis provides three different security mechanisms to facilitate a secure data management for the Semantic Web. Each mechanism implements one of the three security requirements of the C-I-A triad which are confidentiality, integrity, and availability of Semantic Web data. These security mechanisms can be integrated into the Semantic Web architecture depicted in Figure 1.1 by replacing the *crypto* layer with a more generic *security* layer. Such a *security* layer does not necessarily imply any cryptographic operations and is able to comprise all types of security mechanisms independent from their particular design and implementation. The rest of this chapter defines the

research questions of this thesis, identifies its main contributions, and describes the used methodology.

1.1. Research Questions

This section defines the research questions which are answered in this thesis. These research questions derive from the motivation outlined in the previous section and from the three security requirements of the C-I-A triad. Each of the security requirements confidentiality, integrity, and availability is mapped to one research question. As integrity of Semantic Web graph data is closely related to the data's authenticity [37], this security requirement is mapped to a fourth research question. Integrity requires that only authorized parties are able to modify the graph data and authenticity means that the data is retrieved from a verified source [28]. If the source of the data is not verified, it cannot be guaranteed that the graph data is only modified by authorized parties. In the following, the four research questions are explained in more detail.

RQ.1: How can confidentiality of Semantic Web data be ensured in open and distributed networks so that only authorized parties are able to access parts of the data?
Implementing confidentiality of a Semantic Web graph requires that the contents of the graph are hidden from any unauthorized party. Even authorized parties are not necessarily able to access all contents of the graph. Instead, their access may be restricted to particular triples of the graph.

RQ.2: How can integrity of Semantic Web data be achieved in open and distributed networks so that any unauthorized modification of the data is detected?
Integrity of a Semantic Web graph requires that only authorized parties are able to alter the graph's contents. Any unauthorized modification destroys the graph's integrity and must therefore be detected. Please note that parties who are allowed to view the contents of a graph may still be prohibited from altering it.

RQ.3: How can authenticity of Semantic Web data be ensured in open and distributed networks so that the data can be related to a verified source?
Authenticity of a Semantic Web graph requires that the identity of the source which the graph is retrieved from can be verified. In providing a Semantic Web graph, the party acting as the data source approves of the graph's contents.

RQ.4: How can Semantic Web data be made available in open and distributed networks to such parties whose access requests are compliant with predefined and transparently communicated policies?
Availability of data usually requires that the data is always accessible for authorized parties [28]. This thesis enhances this classical definition of availability to *compliant availability* which states that graph data must only be accessible to authorized parties as long as the parties' access complies with a set of predefined conditions. Compliant availability corresponds to DAMA's security requirement

for protection of data against unauthorized access [208]. If the conditions are not met, the graph data must not be available to the requesting party. In contrast to research question **RQ.1**, this research question aims at regulating the availability of complete graphs and not of individual triples.

1.2. Contributions

The main contributions of this thesis are three different security mechanisms which answer the four research questions from the previous section. The security mechanisms implement the security requirements confidentiality, integrity, authenticity, and compliant availability of Semantic Web graph data. They can be integrated into the Semantic Web layer cake as shown in Figure 1.1 by replacing the *crypto* layer with a more generic *security* layer. In the following, the three contributions are described in more detail.

Confidentiality is implemented by T-Store, an approach for searching in encrypted Semantic Web graphs. Data encryption is a common security mechanism to achieve the data's confidentiality [96]. Encrypting plaintext data results in ciphertext data which can only be decrypted by authorized parties with access to the correct decryption keys. Searching in encrypted data extends this basic encryption scheme by supporting queries on the ciphertext. T-Store encrypts a plaintext graph in such a way that the resulting ciphertext graph can be directly used for processing queries without decrypting it first. A query corresponds to a decryption key which only decrypts those parts of the ciphertext graph that fulfill the query. T-Store supports arbitrary queries on the ciphertext graph and is not restricted to a set of queries which are defined at encryption time. It distinguishes between a data owner who encrypts the plaintext graph and users who are authorized by the data owner to perform queries on the ciphertext graph. As unauthorized parties are not able to access any part of the plaintext graph, T-Store answers research question **RQ.1**. A preliminary version of T-Store was first published in [171].

Integrity and authenticity are implemented by Siggi, a formal framework for signing Semantic Web graph data. Digital signatures are a security mechanism for achieving both integrity [267] and authenticity of the signed data [215]. Siggi formally defines a generic signature pipeline for signing arbitrary graph data. The pipeline is independent from any particular algorithm and can be configured with various algorithms to provide different features such as minimum signature overhead or minimum runtime. It divides the signing process into separate functions, each of which implements a specific step of the process. These functions include a canonicalization function, a serialization function, a hash function, and a signature function. Siggi specifies the input and output of each function. The functions are designed in such a way that the resulting signature is independent from the encoding of the signed graph. The signature only covers the graph's semantics and not its syntactical representation. An additional assembly function is applied at the end of the signing process. It stores all information about how a signature was created and how it can be verified by another party. A signature associates a signed graph with the signing party. Modifying the semantics of a graph after it was

signed invalidates the signature and also destroys the graph's integrity. Furthermore, the modification affects the graph's authenticity as well as the signing party only signed the original graph and has not approved of its modified version. Thus, Siggi answers research question **RQ.2** and **RQ.3**. Preliminary versions of Siggi were published in [168, 169, 173].

Finally, compliant availability is implemented by InFO, a policy language for regulating information flow in open and distributed networks. Although availability is usually implemented by providing redundant data storage systems [28], compliant availability requires a different implementation. Compliant availability of data requires the data to be accessible to any requesting party as long as the access complies with a predefined policy. InFO is specifically designed to regulate communication flow in open networks such as the Internet. A policy is a set of rules which share the same purpose and allow or deny a particular communication. A communication is described by a sender and receiver, the exchanged data, and the used communication channel. Policies define the conditions under which a party can access data stored at a server. Each policy contains all details for technically enforcing a regulation and can be implemented on various communication systems such as application-level proxy servers, name servers, and routers. InFO's modular and extensible design also allows to support additional enforcing systems as well. A policy is further enriched by a legal justification and an organizational motivation. As this background information can be transparently communicated to all involved parties, InFO fulfills research question **RQ.4**. A preliminary version of InFO was first published in [172].

1.3. Methodology

Design science research is a paradigm for developing computer-related artifacts in information systems research and computer science [299, 231, 22]. Possible artifacts include abstract models, algorithms and processes, and software implementations [145]. T-Store, Siggi, and InFO comply with this definition and can therefore be considered as artifacts as well. The development process of all three artifacts is based on the design science research paradigm. Several different suggestions have been made to define the particular steps involved in design science research [194, 299, 145, 231]. Although all suggestions define the creation of an artifact and its evaluation as the two most important steps, additional steps have also been proposed. The particular steps for developing T-Store, Siggi, and InFO are based on Vaishnavi and Kuechler [299], Hevner et al. [145], and Peffers et al. [231]. These steps are depicted in Figure 1.2 and are further explained in the following. The overall process of developing an artifact is iterative and uses the results of one iteration to further improve the artifact in the next iteration.

Step 1: Problem identification and motivation

Design science research focuses on developing computer-related artifacts which solve a particular problem. The first step is therefore the identification of this problem [299, 145, 231]. It describes why this problem is important and motivates the development of possible solutions.

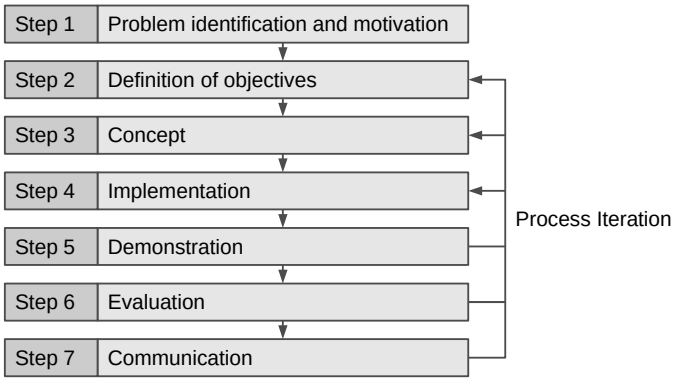


Figure 1.2.: The individual steps of the design science research paradigm [299, 145, 231]. As indicated, the steps can be conducted in multiple iterations.

Step 2: Definition of objectives

In the second step, the functional and non-functional requirements for the artifact are defined [231]. Functional requirements cover the general functions that an artifact must provide and non-functional requirements define general properties and constraints of the artifact [282]. Requirements usually derive from the problem description of the first step.

Step 3: Concept

The third step is the development of the artifact's concept [299]. Depending on the type of artifact to be developed, this concept may already be the final artifact or only a conceptual model of it. As the concept is the core part of an artifact, this step is the most important one in the process of design science research.

Step 4: Implementation

In the fourth step, a prototype of the artifact is implemented based on the concept which is developed in the third step [299]. The particular implementation of the prototype depends on the type of artifact being developed. For example, algorithms can be implemented in software or hardware.

Step 5: Demonstration

The fifth step demonstrates that the artifact can solve a particular instance of the problem identified in the first step [231]. Possible demonstrations include simulations with artificial data and detailed scenarios which show the artifact's utility [145]. The fifth step is a particular type of evaluation which is further conducted in the sixth step.

Step 6: Evaluation

The sixth step is closely related to the fifth step and also evaluates the artifact [299, 145, 231]. However, the sixth step focuses on assessing how well the artifact solves the identified problem [231]. The form of the evaluation depends on

the developed artifact and includes security analyses, performance measurements, or a comparison with the requirements identified in the second step [145].

Step 7: Communication
In the last step, the artifact and its importance is communicated to other researchers and professionals [145, 231]. This allows to receive feedback on the artifact in order to further improve it. Possible forms of communication include scientific publications and conference presentations.

Steps two, three, and four cover individual aspects of the artifact's creation whereas the last three steps evaluate the artifact and assess its quality. These steps provide feedback on the artifact's design and can be used in further iterations of the process as depicted in Figure 1.2. As design science research is only an abstract paradigm, conducting each of the seven steps requires a corresponding methodology [22]. Selecting a suitable methodology for a particular step depends on the type of artifact being development. Table 1.1 summarizes how the individual steps are applied to the three artifacts InFO, Siggi, and T-Store. Each of these artifacts is presented in a separate chapter. InFO is described in Chapter 3, Siggi is covered in Chapter 4, and T-Store is described in Chapter 5. As depicted in Table 1.1, all three chapters are subdivided into different sections which correspond to the individual steps of the design science research paradigm shown in Figure 1.2. In the following, this mapping is further described in more detail.

Table 1.1.: Methodological overview of this thesis. Each of the artifacts InFO, Siggi, and T-Store is developed along the individual steps of the design science research paradigm. The table shows how the steps are mapped to the development process of each artifact.

	Description	**InFO**	**Siggi**	**T-Store**
Step 1	Scientific background	Chapter 1	Chapter 1	Chapter 1
	Practical scenarios	Chapter 2	Chapter 2	Chapter 2
Step 2	Related work	Section 3.1	Section 4.1	Section 5.1
	Identified requirements	Section 3.2	Section 4.2	Section 5.2
Step 3	Concept	Section 3.3	Section 4.3	Sections 5.3 to 5.5
Step 4	Prototypical implementation	Section 3.5	Section 4.6	Section 5.6
Step 5	Scenario implementation	Section 3.4	Section 4.7	Section 5.8
	General applicability	–	Sections 4.4 to 4.6	–
Step 6	Fulfillment of requirements	Section 3.6	Section 4.8	Section 5.9
	Performance analysis	–	–	Section 5.6
	Cryptanalysis	–	–	Section 5.7
	Limitations	Section 3.7	Section 4.9	Section 5.10

The motivation of all three artifacts is divided into a scientific background and two practical scenarios which collectively implement **Step 1** of the design science research

paradigm. The scientific background is based on the four security requirements confidentiality, integrity, authenticity, and compliant availability as described at the beginning of this chapter. These requirements derive from the literature on computer security [37, 28]. Their application to the Semantic Web is motivated by analyzing its architecture which is represented by the Semantic Web layer cake depicted in Figure 1.1. Although the layer cake contains a section which summarizes different security mechanisms, it does not define any particular mechanism yet. The practical scenarios are introduced in Chapter 2 and provide two comprehensive example use cases for applying the three artifacts T-Store, Siggi, and InFO to Semantic Web data. The scenarios necessitate the implementation of the security requirements confidentiality, integrity, authenticity, and compliant availability. The first scenario covers the regulation of information flow on the Internet and derives from analyzing the current practice of such regulations [321, 89]. The second scenario discusses the secure management of medical data in electronic healthcare. It results from analyzing the literature on electronic healthcare [224, 136] as well as its legal requirements [297, 90].

Step 2 is implemented by first identifying the general application domain of each artifact and then extracting its functional and non-functional requirements from the identified domain. In particular, the domain is comprised of the two scenarios defined in Chapter 2 and the related work of each artifact. The related work is summarized in separate sections of the three chapters 3, 4, and 5. Each summarized approach is analyzed regarding its individual features and design characteristics. The result of this analysis is used together with the general requirements of the two scenarios to define the specific requirements for each artifact which are listed in Sections 3.2, 4.2, and 5.2.

The conceptual design resulting from **Step 3** is created differently for each artifact. InFO is designed as a set of ontology design patterns [237, 121], which are described in Section 3.3. The patterns extend the foundational ontology DOLCE+DnS Ultralite (DUL) [119] which defines several ontological concepts and axioms for various application domains [266]. By reusing and further specifying these concepts and axioms, the vocabulary of InFO can be related to the basic categories of human cognition [222] which results in a better linguistic foundation. Siggi provides a mathematical formalization of a generic signing framework that is presented in Section 4.3. The formalization defines a signature pipeline consisting of several steps, which are implemented using different algorithms. The pipeline is based on the XML signature syntax and processing standard [20] for signing and verifying XML documents. The framework is designed in such a way that it is compatible with already existing algorithms that can be used in the individual steps of the signature pipeline. T-Store is basically a collection of mathematical algorithms and data structures. Its concept is described in three different sections. Section 5.3 summarizes the overall process of T-Store by outlining its general design and terminology, Section 5.4 provides a mathematical formalization of the terminology, and Section 5.5 applies the formalization to describe all steps of T-Store in more detail.

InFO, Siggi, and T-Store are essentially formal models and not particular software implementations. As such, the implementation of these artifacts conducted in **Step 4** is not an integral part of this thesis and is mainly used for demonstrating their practical applicability and for supporting further evaluations. InFO is implemented on three

prototypical systems which enforce particular regulation policies and are presented in Section 3.5. The feedback drawn from each implementation was used for improving the design of InFO's ontological model. Siggi is implemented by mapping its formal specification to source code. The implementation is described as part of Section 4.6 which discusses the performance of four example configurations of the framework. T-Store is implemented similarly to Siggi by transforming its mathematical model into a software application. The implementation is also used for evaluating the artifact's performance and is described as part of Section 5.6.

Step 5 demonstrates the applicability of an artifact and is a particular type of evaluation. The applicability and utility of all three artifacts are demonstrated by using them to implement the two scenarios of Chapter 2. InFO is used for creating example regulations which are enforced by the prototypical implementations described in Section 3.5. The policies and their enforcement show that InFO can in fact be used for regulating information flow in open and distributed networks. Section 4.7 shows how Siggi is applied to sign the example regulations and thereby demonstrates the general signing process of Semantic Web graphs. In addition, Section 4.4 shows four different example configurations of a particular signature pipeline. These configurations demonstrate that Siggi in fact supports different algorithms. The configurations are further analyzed with respect to their cryptographic security in Section 4.5 and their performance in Section 4.6. The applicability of T-Store is demonstrated by outlining two possible applications for searching in encrypted data in Section 5.8. Both applications derive from the example scenarios of Chapter 2 and extend the basic concept of T-Store with additional features.

The evaluation of an artifact as required in **Step 6** depends on the type of the artifact and on the particular criteria to be assessed [145, 238]. All three artifacts are compared with their related work and analyzed regarding the fulfillment of their functional and non-functional requirements defined in **Step 2**. This analysis is conducted manually in Sections 3.6, 4.8, and 5.9 by considering the artifacts' individual characteristics. In addition, all artifacts are evaluated against their conceptual limitations in Sections 3.7, 4.9, and 5.10. T-Store provides a particular algorithm for searching in encrypted data which is further evaluated with regard to its performance and cryptographic security in Sections 5.6 and 5.7. The performance evaluation is based on an evaluation framework [40] which provides artificial graph data. It is conducted in several experiments in which different artificial graphs are encrypted and queried. In each experiment, the runtime and memory usage are measured. The cryptanalysis is based on different attacks which derive from the related work and state of the art. It is conducted by carefully analyzing the mathematical design of T-Store. As neither InFO nor Siggi provide a particular algorithm, they cannot be evaluated the same way. Although four example configurations of Siggi are analyzed regarding their cryptographic security and performance, these analyses do not apply to the whole framework. Instead, they only cover the particular configurations and are part of **Step 5**.

In order to receive feedback from other researchers on all three artifacts as required in **Step 7**, each artifact was published at scientific workshops and conferences as well as in journals. InFO was published in [172], Siggi was published in [168, 169, 170, 173], and T-Store was first published in [171]. All publications were peer-reviewed and the feedback

from the reviewers was used for further improvement together with the discussions at the workshops and conferences. In addition, InFO and Siggi were implemented by university students who also provided helpful comments on the artifacts' conceptual design.

Chapter 2.

Scenarios for Secure Semantic Web Data Management

This section describes two different scenarios which demonstrate the need for secure semantic web data management in open and distributed networks. The scenarios consist of several parts which motivate the research questions **RQ.1**, **RQ.2**, **RQ.3**, and **RQ.4** defined in Section 1.1. They also serve as two example applications of the artifacts developed in this thesis which are described in Chapters 3 to 5. The first scenario is given in Section 2.1 and focuses on regulating communication in open networks such as the Internet. The scenario requires a policy language for modeling allowed and prohibited communication, a framework for signing regulation policies, and a mechanism for securely evaluating log data of Internet activities. The second scenario is described in Section 2.2 and covers the secure distribution of medical data between patients and medical institutions. The scenario requires a framework for signing medical data, a policy language for managing the secure distribution, and a mechanism for regulating access to the patients' personal data. At the end of each subsection, the scenario is summarized and its relation to the research questions is demonstrated. The implementation of the two scenarios is demonstrated in Chapters 3 to 5 as part of the developed artifacts.

2.1. Regulating Internet Communication

The Internet is a global communication medium which interconnects several computer networks located in different countries and managed by different authorities. The content provided on the Internet can generally be accessed by anyone from anywhere. However, each country connected to the Internet has its own national laws and wants to enforce these laws on the Internet as well. For example, neo-Nazi material can legally be accessed in the USA but its access is regulated in Germany due to the country's history [198, 293]. Additionally, organizations and institutions want to enforce their own rules within their local computer network on top of existing national regulations. For example, companies may want to prohibit their employees from accessing any non-work-related Internet content which distracts them from their actual work [280, 186]. Schools are even required by law to protect their students from accessing any content which is

inappropriate for minors [151, 61]. Besides these examples, there are many other cases where such regulations are desired or even needed [89].

This section outlines a scenario for regulating Internet communication. The scenario covers an example computer network, a workflow for creating and distributing regulation details, and a concept of privacy compliant logging of Internet activities. The example network consists of several communication nodes and smaller sub-networks which are managed by different authorities. These authorities regulate the communication flow in the networks by applying a workflow for creating and distributing regulation policies. The policies are implemented by technical regulation systems which may also record all Internet traffic. In order to support traffic analyses without interfering with the privacy of the affected Internet users, the logging mechanism stores all recorded data in encrypted form and restricts decryption to authorized parties only.

2.1.1. Example Network Topology

Computer networks consist of communication end nodes such as web servers and client systems as well as intermediary communication nodes like routers and application-level proxy servers. An example computer network connecting various communication nodes located in the USA, Germany, and Saudi Arabia is depicted in Figure 2.1. Each of the three countries has its own national network which includes smaller subnetworks such as access provider networks or networks of organizations and institutions. National networks and their subnetworks again contain several communication end nodes and intermediary nodes. The communication end nodes cover both end user computers such as the US client, the DE client, and the SA client as well as web servers such as weather servers and pornography servers. End users and small institutions such as schools do not access the Internet directly. Instead, they are customers of access providers and access the Internet through their respective access provider network in their country. For example, the US client resides in the USA and uses the TDS Telecom[1] as its access provider. The SA client is located in Saudi Arabia and is connected to the Internet via the network of Sahara Net[2]. The DE client and the comprehensive school are located in Germany and use the German Telecom[3] as their access provider. The network of the comprehensive school contains several student computers which act as client systems. These computers only access the Internet after having passed the intermediary communication nodes of the school's network and the network of the German Telecom.

Each web server in the example network is operated by a content provider and can be accessed by any user from any country. A content provider can in principle regulate the access to its provided content at server side. However, content providers may not always be capable of or even interested in denying access for particular users based on national laws. Thus, regulations of information flow on the Internet are often implemented on intermediary nodes like routers, name servers, and application-level proxy

[1] `http://www.tdstelecom.com`, last accessed: 01/21/16
[2] `http://www.sahara.com`, last accessed: 01/21/16
[3] `http://www.telekom.de`, last accessed: 01/21/16

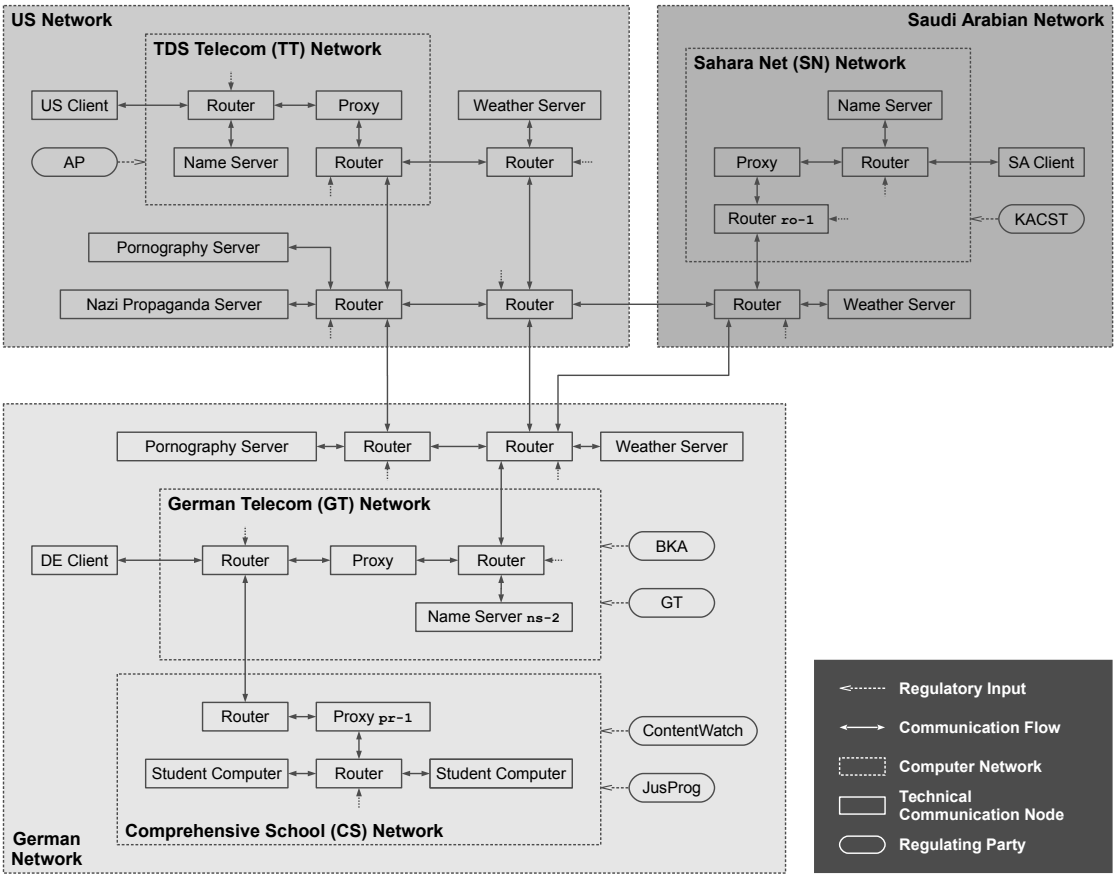


Figure 2.1.: Illustrating example of a network topology and its involved authorities.

servers [212, 321, 79]. Routers are able to regulate Internet communication by dropping IP packets. Application-level proxy servers can control the flow of information by filtering the accessed URLs and evaluating the content of web-pages. Name servers can restrict the access to a web server by returning a wrong IP address or no IP address at all. The example network depicted in Figure 2.1 contains the router `ro-1`, the name server `ns-2`, and the proxy server `pr-1` which can be used for regulating the information flow. The router `ro-1` and the name server `ns-2` are both operated by their respective access providers. Access providers are able to regulate the Internet communication between their users and the accessed web servers [95], since they operate in the same country their users reside in. Unlike content providers, access providers are not only familiar with the laws that the users must abide by but are also required to enforce them. Although they are required to enforce the same laws, access providers often interpret these laws differently which results in different flow control implementations [95]. Even if the access providers reside in different countries, they often have to implement transnational laws such as EU directives. Depending on the country where the user lives in, access to particular websites may be either legal or illegal. For example, pornographic

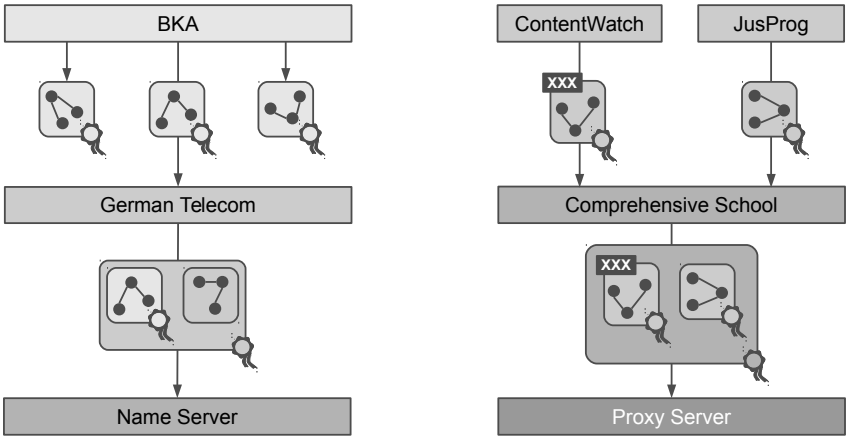
content may be legally accessible by German users and US users over a specific age[4], but not by Saudi Arabians according to §6 of the Saudi Arabian Anti-Cyber Crime Law [179]. Access to neo-Nazi propaganda is legal in the USA and in Saudi Arabia but not in Germany according to §86 of the German Criminal Code [62]. Finally, weather information provided by a weather server can be accessed by users of all three countries. In order to reduce the number of possible errors when interpreting national laws or transnational directives, this interpretation is sometimes made by third parties. The details of the interaction between all parties involved in the regulation process are further described in the next section.

2.1.2. Creating and Distributing Regulation Policies

As outlined above, information flow control can be enforced at three different types of network nodes [212, 321, 79], namely routers, application-level proxy servers, and name servers. The example network depicted in Figure 2.1 provides different instances of these enforcing communication nodes. Each type of node requires specific content identifiers such as IP addresses, domain names, or URLs. The collection of such identifiers is often based on interpreting national laws or transnational directives. This process differs from country to country and is implemented by access providers and/or by third parties which may even be authorized by the country's government. In Saudi Arabia, all content identifiers are collected and managed centrally by the King Abdulaziz City for Science & Technology (KACST)[5] [226]. The USA does not have such a central institution. Instead, the identifiers of the regulated web content are collected and managed decentrally by private parties such as Internet access providers [227]. In Germany, there is a hybrid situation in which the Federal Criminal Police Office (Bundeskriminalamt; BKA)[6] centrally collects content identifiers and delivers them to the access providers [64]. In addition, several court decisions have required German access providers to manage content identifiers themselves in order to block access to particular web servers [95]. Apart from the national laws of a country, access providers can also define their own code of conduct or guiding principles for information flow control. An example of such principles is the code of conduct of the German Telecom [91]. It basically states that the internationally operating company abides by the national law of the physical location of its subsidiary. Another example of a code of conduct are the Principles on Freedom of Expression and Privacy [129] issued by the Global Network Initiative (GNI). The GNI consists of large companies of the information and communications technology sector including Google Inc., Microsoft Corporation, and Yahoo! Inc. It aims at providing more transparency in Internet regulations. Furthermore, organizations and institutions such as the comprehensive school located in Germany may also want to enforce their own rules and regulations. In the case of the comprehensive school, the underage students must be prevented from accessing mature content such as pornography according to §184 of the German Criminal Code [61]. Instead of creating the corresponding regulations itself,

[4] Please note that this excludes specific content like child abuse images.
[5] `http://www.kacst.edu.sa`, last accessed: 01/21/16
[6] `http://www.bka.de`, last accessed: 01/21/16

the comprehensive school entrusts third parties such as ContentWatch and JusProg to compile such regulations. ContentWatch Inc.[7] is a private company located in the USA which provides different solutions for regulating Internet communication within organizations and institutions. JusProg e. V.[8] is a registered society in Germany that creates and distributes filter lists of websites which are considered to be harmful to minors. By using regulation policies from two different sources, the comprehensive school achieves a larger coverage of undesirable web content.



(a) Regulating the network of the German Telecom

(b) Regulating the network of the comprehensive school

Figure 2.2.: Process of distributing policies for regulating Internet communication.

All regulating authorities encode their collected content identifiers and other regulation details as Semantic Web graph data. If the implementation of a particular regulation involves several authorities, these authorities follow a specific workflow for creating and exchanging regulation information. Each regulating authority receives signed graph data from another authority, adds its own graph data, digitally signs the result, and sends it to the next authority. Digitally signing the graph data allows the authorities to verify the data's integrity and authenticity. Integrity means that the data was not modified after the signature had been created and authenticity means that the signing authority has approved of the data. Figure 2.2a depicts the regulation workflow for the German Telecom and Figure 2.2b shows the workflow for the German comprehensive school depicted in Figure 2.1. The regulation workflow of the German Telecom involves the BKA and the access provider itself. The BKA not only provides the content identifiers for specific regulations but also a set of formally defined ontologies for describing them. The ontologies consist of several ontology design patterns [122] which allow to represent different types of knowledge including wanted persons, recent crimes, and details for regulating Internet communication. The content identifiers provided by the BKA are summarized as a blacklist of web sites. These web sites contain neo-Nazi material and

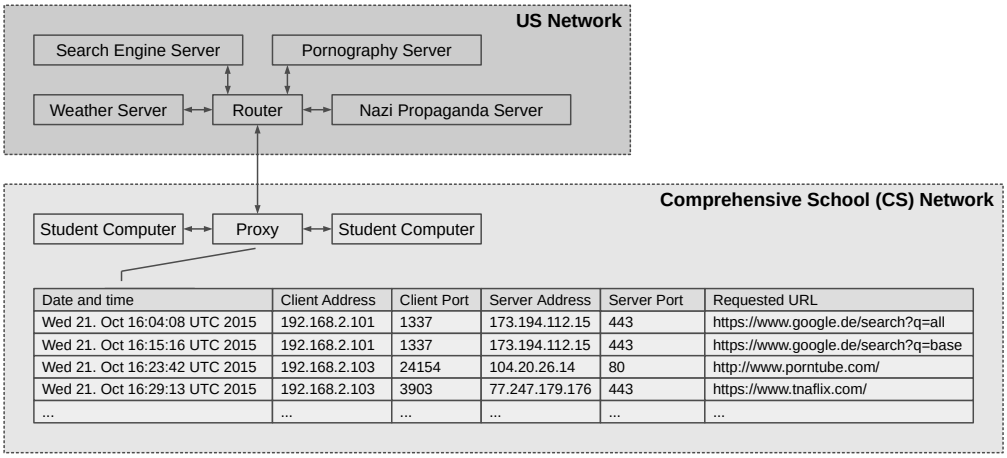
[7] `http://www.contentwatch.com`, last accessed: 01/21/16
[8] `http://www.jugendschutzprogramm.de`, last accessed: 01/21/16

are to be blocked according to §86 of the German Criminal Code. Each entry in the blacklist corresponds to a particular regulation rule that describes which web sites are to be regulated. As the German Telecom implements the regulations on a name server, the BKA uses domain names as content identifiers for the web sites. After having created the blacklist, the BKA digitally signs the list and the ontologies as well and publishes the ontologies on the web. It then sends the signed blacklist to the German Telecom via a secure communication channel by using, e. g., an SSL [113] connection. The German Telecom receives the regulating information from the BKA and verifies its signature. The blacklist provided by the BKA only describes the URLs of the web sites which are to be regulated but not how the regulation shall be implemented. Thus, the German Telecom interprets the data received from the BKA and adds concrete implementation details such as the IP address of the name server used for blocking the web sites. As shown in Figure 2.2a, the German Telecom compiles its technical regulation details as RDF graph which is based on the BKA's ontology design pattern. It digitally signs the BKA's blacklist together with its own regulation graph and sends it to its name server. The name server verifies the signature of the German Telecom in order to prohibit any unauthorized party from manipulating its implemented regulations. If the verification is successful, the name server maps the blacklist of the BKA to a native format. This format can then directly be used by the name server for regulating access to web sites.

As depicted in Figure 2.2b, the regulation workflow of the German comprehensive school involves ContentWatch, JusProg, and the school itself. The comprehensive school has to ensure that its students cannot illegally access neo-Nazi content and other mature material such as pornography. As the comprehensive school is connected to the Internet via the network of the German Telecom, all regulations implemented in the network of the access provider also affect the school's network. Thus, the comprehensive school does not need to implement any regulations for blocking access to neo-Nazi material since such regulations are already implemented by the German Telecom. However, the school still needs to regulate access to pornographic content as the German Telecom does not provide such regulations. To this end, the school receives regulation information for adult content from ContentWatch and JusProg. ContentWatch and JusProg collect URLs of web pages which contain pornographic content and other adult material which is harmful to minors. These URLs are used for creating specific regulation rules which are signed and sent to the comprehensive school via a secure communication channel. ContentWatch provides its regulation information as Named Graphs whereas JusProg provides its regulation data as regular RDF graphs. After having received both blacklists, the school verifies their signatures and consolidates them into a single regulation policy. The school digitally signs this regulation policy again and sends it to its local proxy server. The proxy server verifies the policy's signature and maps the regulation rules to its local database. By redirecting all Internet traffic from the student computers located in the school's network through this proxy server, the school ensures that its students can only access the Internet after having passed the predefined regulation mechanisms.

2.1.3. Privacy Compliant Logging of Internet Activities

The proxy server of the comprehensive school not only regulates access to the Internet for all student computers but also records their Internet traffic. This is necessary as German schools are required by law to supervise the actions of their students [144] which also includes their Internet activities [284]. To achieve this, the proxy server maintains a database with several log entries each of which stores all details of a particular Internet communication. These details include the IP address and the port number of the student computer and the contacted web server, the URL of the requested web page, and the date and time of the access. Figure 2.3 shows an example log database of the school's proxy server. It also shows how the proxy server acts as a gateway to other computer networks and thus is able to record all Internet traffic of the student computers. As depicted, log entries may contain personal information such as the student's search interests [81]. Therefore, the proxy server encrypts all log entries before storing them in its database. Encrypting the entries ensures their confidentiality, which means that they can only be accessed by authorized parties [37]. In addition, the encryption also prohibits any abuse of the logged Internet traffic [310].

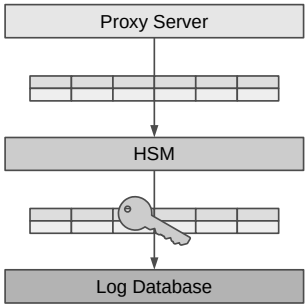


Date and time	Client Address	Client Port	Server Address	Server Port	Requested URL
Wed 21. Oct 16:04:08 UTC 2015	192.168.2.101	1337	173.194.112.15	443	https://www.google.de/search?q=all
Wed 21. Oct 16:15:16 UTC 2015	192.168.2.101	1337	173.194.112.15	443	https://www.google.de/search?q=base
Wed 21. Oct 16:23:42 UTC 2015	192.168.2.103	24154	104.20.26.14	80	http://www.porntube.com/
Wed 21. Oct 16:29:13 UTC 2015	192.168.2.103	3903	77.247.179.176	443	https://www.tnaflix.com/
...	...	...	...	...	...

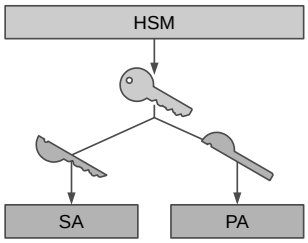
Figure 2.3.: Example log database with several entries of students' Internet activities. The database is maintained by the school's proxy server which also serves as a gateway to other computer networks.

In order to prevent the comprehensive school from abusing the log entries itself, access is restricted to a supervisory school authority and first must be authorized by both the school's administration and the parent's association. These two parties have different organizational functions, consist of different members, and have different interests. Neither of them is able to authorize any access to the log entries without the other party's consent. The supervisory school authority acts as an independent investigator and is neither associated with the school's administration nor with the parent's association. It oversees all actions of the school and regularly analyzes the students' Internet activities. During an analysis, the authority checks if the school's regulation is implemented cor-

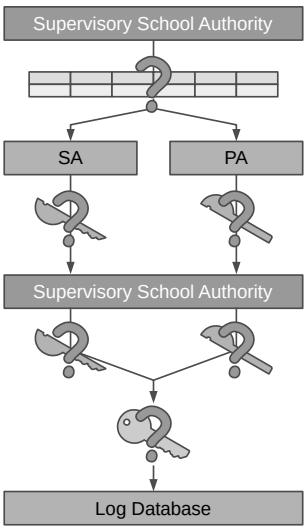
rectly and if any additional regulation might be necessary. To support an analysis while simultaneously protecting the entries' confidentiality, the encrypted log database can be queried without decrypting it first. The different steps of storing and querying the log entries are depicted in Figure 2.4. When recording an Internet communication, the school's proxy server sends all technical communication details to its integrated hardware security module (HSM) [287]. The HSM encrypts the log entries and also creates and securely stores the used encryption keys in such a way that they cannot be extracted. Directly embedding the HSM in the proxy server ensures that the plaintext entries are not processed outside of the server. The process of encrypting and storing the log entries is shown in Figure 2.4a. In order to apply a query to the encrypted database, it first must be authorized by both the school's administration (SA) and the parents' association (PA). To this end, each encryption key stored in the HSM is split into two parts which are sent to the two parties. Figure 2.4b depicts the process of splitting encryption keys and storing the resulting fragments and Figure 2.4c shows how the two fragments are used for authorizing a particular query. When the supervisory school authority wants to analyze the log database, it creates a corresponding query and sends it to both the school's administration and the parents' association. The two parties combine the query with their own fragment of the encryption key independently and return the result to the supervisory school authority. The authority combines the two values into a single decryption key which encodes the authorized query. A valid decryption key can only be created if both the school's administration and the parents' association have authorized the query. Otherwise, the supervisory school authority has an invalid decryption key which cannot be used for decrypting any entry in the log database. If the decryption key is created correctly, the authority applies it to the encrypted database and thereby



(a) Storing the log entries



(b) Splitting the encryption key



(c) Querying the database

Figure 2.4.: Overview of the steps for storing and querying encrypted log entries.

searches for all log entries which match the encoded query. Finally, the authority uses the resulting entries for further analyzing the student's Internet activities and checks them for any inappropriate access.

2.1.4. Summary of the Scenario

The scenario for regulating Internet communication requires a policy language for describing allowed and prohibited communication flow. This policy language must be able to describe the communication flow in the example network described in Section 2.1.1. The policy language answers research question **RQ.4** as it makes Internet content available as long as it is compliant with a set of predefined rules. A particular policy is created by several authorities which communicate with each other. In order to provide integrity and authenticity of their exchanged data, all data is signed before its transmission. The provided signature is permanently attached to the signed data and not restricted to the existence of a secure communication channel. Creating a signature requires a corresponding signing framework which answers research questions **RQ.2** and **RQ.3**. Internet regulations are enforced by different communication nodes such as routers, application-level proxy servers, and name servers. As an enforcing system's regulations may be further analyzed by its operator or a third party, the system logs all of its activities. However, logging the Internet activities of regulated Internet users may invade their privacy. Thus, each enforcing system encrypts its log files and makes them only partially available to authorized parties. This is achieved by providing a mechanism for searching in encrypted data. The corresponding approach achieves confidentiality of data and thus answers research question **RQ.1**.

2.2. Securing Medical Data Records in Electronic Healthcare

E-health commonly refers to the application of electronic technologies in health care and aims at improving medical processes in various aspects [224]. A core part of e-health is the digitization of medical data records and their exchange between different care delivery organizations (CDOs) by using information and communication technology such as the Internet [105]. CDOs are medical institutions such as general practitioners, medical specialists, or hospitals. The use of electronic records in e-health aims at easing the process of exchanging medical information between different CDOs and providing more accurate data than paper-based records [177, 136]. In addition, electronic records can be processed automatically by software applications and can be used to support medical decisions and treatments [279, 220]. These main goals of e-health can be achieved by improving the interoperability between different CDOs and their patients. As e-health relies on electronic records, the aspired interoperability requires a standardized data format for exchanging such records [154]. This section presents a scenario which focuses on the secure exchange of electronic records based on a standardized format. The scenario distinguishes between different types of medical data records, defines the security requirements for exchanging them between different parties, and provides an example use case of the data exchange.

2.2.1. Medical Data Records Used in Electronic Healthcare

Medical data records of a patient can generally be distinguished between electronic medical records (EMR), electronic health records (EHR), and personal health records (PHR) [135, 125, 289]. An EMR is an electronic data record that stores the results of a single medical test, examination, or operation which is conducted by a single CDO for an individual patient. This CDO not only creates the EMR but also manages all access requests to it [177]. An EHR is a collection of several EMRs of the same patient and may be created by different CDOs [177]. EHRs support the aspired interoperability of e-health as they are used for exchanging medical information between different CDOs. An EHR may focus on a particular medical case and only covers EMRs which are relevant to that case.[9] Both EMRs and EHRs are created, managed, and owned by CDOs while PHRs are managed and owned by the patients themselves. A PHR contains both subjective and objective medical information of an individual patient [289]. Subjective data is created by the patient directly and includes symptoms as well as their descriptions and assessments. Objective data covers measured values such as blood pressure or weight and is created by technical devices used by the patient. Each patient is associated with a single PHR which is stored in a personal storage device or in a cloud system. Medical information about the same patient may also be stored in several EHRs which are scattered across different CDOs.

Depending on their actual use, EMRs, EHRs, and PHRs may store similar data and overlap with each other. However, each type of record also contains some information which is not part of other record types. Thus, combining all types of records results in a comprehensive collection of medical information about a single patient which can be used by CDOs to conduct medical treatments [289]. An expedient integration of all records requires that they share a common data format or that they are stored in different data formats which can easily be mapped to each other [303]. Compatibility between all records is also necessary in order to achieve interoperability between different CDOs and their patients' PHRs. Vizenor et al. [300] argue that Semantic Web ontologies are best suited as a common data format for different e-health records. Ontologies can be used for aligning different vocabularies and terminologies from different medical departments. Furthermore, they allow to reason on medical data which can be used to support medical decisions by relating a patient's medical records to medical knowledge bases. Example ontologies which are specifically designed for reasoning on medical data are presented in [279, 220, 246].

2.2.2. Security Requirements for Medical Data Records

Medical data records store sensitive information about a patient and must therefore be protected against unauthorized access. The legal foundation for e-health applications in the USA is the *Health Insurance Portability and Accountability Act* (HIPAA) [297]. HIPAA does not only motivate the use of interoperable e-health applications and data

[9] Please note that there are varying definitions of EMRs and EHRs in the literature [318]. In this thesis, the terms EMR and EHR are used as defined in this section.

formats but also defines several requirements for achieving security and privacy of the medical data. The security requirements are further refined by the *Security Standards for the Protection of Electronic Protected Health Information* [90] and correspond to confidentiality, integrity, and availability of the data. Confidentiality states that the patient's medical data must be protected from any unauthorized access. This requirement is especially necessary since medical data contains highly sensitive information such as a patient's health status and diseases [312]. Integrity requires that the data is not modified or destroyed by unauthorized parties. Since medical data is used for medical examinations and treatments, ensuring that the data is correct and accurate is essential to ensure a patient's safety [108]. Availability states that the data must be accessible to authorized parties whenever needed. Supporting this requirement is also fundamental to a health care system as incomplete or missing data may otherwise affect a patient's treatment [108, 136]. In Europe, the legal foundation for e-health applications is based on several directives which focus on protecting any type of personal data. The *Data Protection Directive* [100] requires that processing and transmitting personal data is only conducted if the data's privacy and confidentiality can be guaranteed. The directive can be equally applied to electronic processing and non-electronic processing. The *Directive on Privacy and Electronic Communications* [101] complements the Data Protection Directive and is specifically designed for electronic processing and transmission of personal data. The directive states that the basic security requirements confidentiality, integrity, and availability of the data must be implemented in computer systems which process the data. As both directives can be applied to various types of personal data, they cover medical data records as well.

2.2.3. Example Medical Case

Marla manages her own PHR in order to keep track of her health. She uses the PHR to monitor various medical information including her weight, blood pressure, and pulse. Whenever she feels sick, she records her symptoms in the PHR to support a medical professional's diagnosis. She also uses her PHR to record any prescribed medicine and its recommended taking. In order to ensure the confidentiality of her health data, Marla encrypts her PHR and stores it on a portable device. If she consults a CDO such as a doctor or hospital, she grants the medical personnel access to particular parts of her PHR to support her medical treatment. Marla provides all medical information in her PHR as Semantic Web graph data to achieve a better interoperability between her PHR and the EHRs and EMRs managed by CDOs. Figure 2.5 depicts the application of this data for a particular medical case.

When Marla senses a lump in her throat and suffers from swallowing difficulties, she consults her general practitioner (GP). The GP asks Marla to grant him access to those parts of her PHR which he suspects to be related to her symptoms. After having analyzed the requested entries in the PHR, the GP relates the symptoms to a goiter and performs an ultrasonography to examine Marla's thyroid. He records all results of the examination as graph data and digitally signs it to ensure the data's integrity and authenticity. The resulting graph corresponds to an EMR and is stored on a local

server in the GP's office. As Marla's thyroid is slightly enlarged, the GP prescribes her iodine tablets after having reviewed Marla's PHR to ensure that she does not take any medicine which is incompatible with the tablets. The GP records the prescription as an additional EMR, signs it, and stores it on the local server as well. Marla also records the GP's prescription and the recommended intake in her PHR.

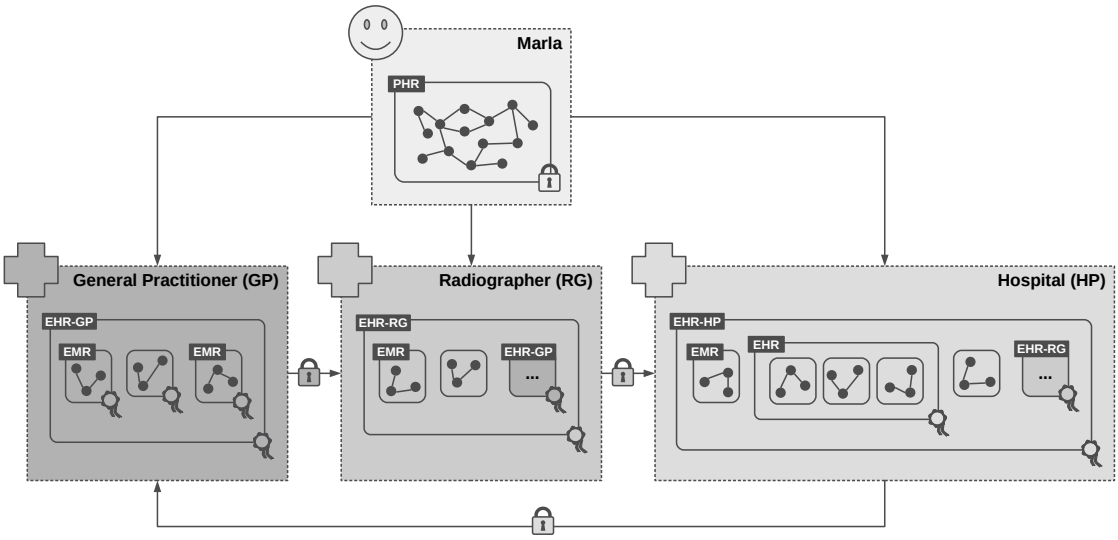


Figure 2.5.: Example medical case which depicts the secure transmission and storage of medical data. Arrows indicate the flow of medical data.

Even after taking the prescribed medicine, Marla's symptoms do not disappear and she consults her GP again. The GP performs another ultrasonography, digitally signs the resulting EMR, and stores it on the local server. As the second ultrasonography shows a noticeable enlargement of Marla's thyroid, the GP refers her to a radiographer for additional examinations. The GP creates an EHR which contains the EMRs of the two examinations and his prescription. He digitally signs the EHR and sends it to the radiographer via the Internet. By signing the EHR, the GP states that all medical records contained in it are part of the same medical case and that this data is complete. The proxy server of the GP's office ensures that medical data is only transmitted to other CDOs via a secure communication channel such as an SSL [113] connection. If the GP accidentally tries to use an insecure communication channel or send the data to a different destination, the transmission is blocked by the proxy server. Thus, the proxy server implements the requirement for compliant availability of medical data. After having received the EHR from the GP, the radiographer verifies its signature in order to ensure its integrity and authenticity. Based on the EHR from the GP and on Marla's PHR, the radiographer performs a scintigraphy on Marla's thyroid in order to test whether or not Marla suffers from hyperthyroidism. The results of the examination correspond to an EMR which is stored on a local server in the radiographer's office. As the results confirm the radiographer's assumption on hyperthyroidism, he refers Marla to a hospital. The radiographer creates an additional graph containing his diagnosis and

signs it together with the EMR of the scintigraphy and the EHR received from the GP. In doing so, the radiographer states that his own examinations are based on the findings of the GP. The signed graph corresponds to another EHR which is sent to the hospital. Similar to the GP's office, the proxy server of the radiographer's office ensures that medical data such as EHRs are only transmitted via a secure communication channel.

The hospital receives the EHR from the radiographer and verifies its signature. After having evaluated the findings from the radiographer and the GP as well as Marla's symptoms stored in her PHR, the hospital staff conducts a second scintigraphy on the patient's thyroid. Again, the results of this scintigraphy are provided as graph data and stored as an EMR on the hospital's server. Since the second scintigraphy provides similar results as the first scintigraphy, the hospital staff initiates a thyroidectomy in order to surgically remove parts of Marla's thyroid. An operation generally consists of several steps which must all be documented according to the guidelines of the World Health Organization [313]. The steps include the preoperative note, operating room records, and postoperative notes. Each of these steps is recorded as a separate graph. After having completed the operation, the graphs are collectively signed by the operation team as an EHR. When discharging Marla from the hospital, the hospital staff compiles a discharge note and signs it along with the EMR of the performed scintigraphy and the EHR from the operation. As depicted in Figure 2.5, the hospital also includes the EHR from the radiographer in the new graph to state that the operation was conducted after having reviewed the radiographer's examination results. The resulting EHR is sent back to Marla's GP for documentation reasons by using a secure communication channel. Again, the hospital's proxy server ensures that medical records are only transmitted via secure channels.

2.2.4. Summary of the Scenario

E-health aims at improving various aspects of medical processes such as the storage and the transmission of medical data records. Medical records are created by different parties including patients and care delivery organizations (CDOs). Patients who maintain their own medical records encrypt the records in order to ensure their confidentiality. A CDO accesses specific parts of the encrypted records by applying queries to them which are authorized by the patients. Such a protection mechanism requires an approach for querying encrypted data which answers research question **RQ.1**. CDOs create medical data records after having examined the patients or having operated on them. A CDO signs the medical records in order to ensure their integrity and authenticity. The organization can even sign medical data records from other CDOs which have already been signed. Such an iterative signing can be used to track the provenance of the medical records and document the data flow of the signed data. Signing medical data requires a signing framework which answers research questions **RQ.2** and **RQ.3**. Exchanging the signed records between different CDOs must be done via a secure communication channel. In addition, transmitting the records to any other destination which is not a CDO must be prohibited. Describing such rules for sending content via open networks requires a policy language for regulating information flow control. Such a policy language answers

research question **RQ.4** as it ensures that medical records are only available to parties which comply with the predefined rules.

Chapter 3.

InFO: A Policy Language for Regulating Information Flow

This chapter presents the *Information Flow Ontology* (InFO), a generic policy language for regulating information flow in open and distributed networks such as the Internet. Regulations expressed with the InFO policy language can be enforced by various communication nodes operating on different levels of the OSI reference model [159] such as the Internet layer or the application layer. Example nodes which are natively supported by InFO as enforcing systems are routers, application-level proxy servers, and name servers. However, InFO's generic language model can also be further refined to support other types of communication nodes as well. An InFO policy consists of multiple rules which share the same purpose. Each rule regulates one particular communication flow and provides several technical regulation details for implementing the regulation on an enforcing node. The purpose of a policy is expressed with human-readable background information which is directly embedded into the policy. This allows an easy comparison between different policies and to check whether or not a policy implements the correct high-level regulation. The provided background information covers a regulation's legal foundation and its organizational motivation. The InFO policy language achieves compliant availability and thus answers research question **RQ.4** by making information available to authorized parties as long as they comply with the rules of the regulation policies. If a policy rule prohibits the access to any requested information, the party trying to access the information is considered to be unauthorized. A prior version of the InFO policy language was published in [172]. This chapter is based on this publication but rephrases its contents and further enriches them with additional aspects.

The remainder of this chapter is organized as follows: The state of the art and related work for regulating information processing is summarized in Section 3.1. Based on this section and on the scenarios introduced in Chapter 2, Section 3.2 defines the functional and non-functional requirements for the InFO policy language. The design of InFO is described in Section 3.3. Section 3.4 demonstrates how the policy language is used for expressing specific regulations which can be enforced by routers, application-level proxy servers, and name servers. These example regulations are implemented on prototypical systems which are presented in Section 3.5. Section 3.6 assesses the state of the art and

related work and compares it with the InFO policy language. Limitations and possible improvements of InFO are discussed in Section 3.7 before the chapter is concluded.

3.1. State of the Art and Related Work

Policy languages for regulating information processing can generally be distinguished between access control languages, flow control languages, and usage control languages. A simplified depiction of these three different types of policy languages for information regulation is provided in Figure 3.1. As depicted, classical access control focuses on regulating access to information at the content provider's side (i. e., the server) whereas usage control covers the regulation of information at the consumer's side (i. e., the client) [229, 260]. In contrast, flow control allows to regulate the flow of information between the provider and the consumer. This section provides a summary of different policy languages for access control, usage control, and flow control as well as general purpose languages. This section also covers content labeling schemes as they are closely related to the regulation of information processing. Content labeling schemes provide descriptions of the content being regulated which may be used directly by other policy languages. A detailed comparison of the outlined policy languages and content labeling schemes with InFO is provided in Section 3.6.

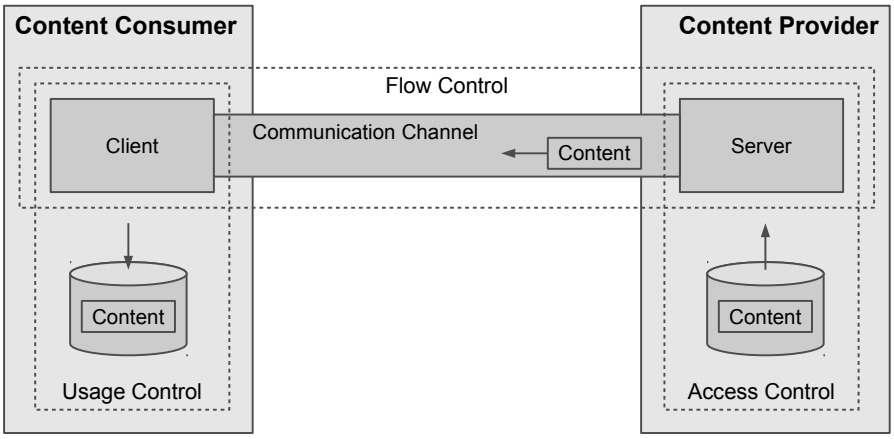


Figure 3.1.: Three different types of policy languages and their involved systems for regulating information processing. The types for regulating information processing include access control, flow control, and usage control.

3.1.1. Access Control Languages

Access control languages ease the configuration of access control systems which regulate access to digital resources [257]. The Access Management Ontology (AMO) [58] is an RDFS [195] ontology for describing access control rules for collaborative web-based

document systems such as wikis or content management systems. In order to ease the creation of access rules and their integration into such systems, AMO's design is very simple and only allows to model permitted actions. Other types of rules such as prohibitions are not supported. Actions, which are not explicitly allowed by AMO, are considered to be forbidden. CommonPolicy [276] is an XML-based language for describing access rules for personal data. Similar to AMO, the language model of Common Policy is rather lightweight and only allows to define permitted actions. Furthermore, Common Policy is designed to be used only in combination with additional application-level protocols such as FTP or HTTP. These protocols must cover the authentication of the requesting user and the transmission of the requested data. WebAccessControl[1] is another lightweight RDFS ontology for describing access control rules for web resources. It is designed for decentralized systems in which the web resources and the users can be managed by different parties. All users are identified by their WebID[2] which serves as a globally unique identifier. User authentication is based on the WebID authentication protocol. Similar to AMO and CommonPolicy, WebAccessControl has a simple design which only supports permitted actions. The Enterprise Privacy Authorization Language (EPAL) [15] and the eXtensible Access Control Markup Language (XACML) [205] are XML-based access control languages which allow to create much more expressive policies than AMO, Common Policy, or WebAccessControl. Both languages are designed to be used within closed network environments such as intranets of large corporations. While EPAL merely focuses on regulating access to personal data, XACML does not have a predefined use case and can be used for regulating the access to arbitrary data. Specific use cases are implemented by creating corresponding XACML profiles. A profile for regulating access to personal data is given in [206]. XACML is much more expressive than EPAL and can replace it in many applications [12].

In summary, classical access control regulates access to data within a closed environment [229, 260]. XACML and EPAL can be considered as classical access control systems since they require a centrally controlled enforcing infrastructure. On the other hand, AMO, Common Policy, and WebAccessControl focus on regulating access to pieces of information in a rather open environment such as the Internet. In all access control systems, content providers regulate the access to their content. However, such regulations cannot be used for regulating the communication flow between an arbitrary server and arbitrary client in the Internet.

3.1.2. Usage Control Languages

Rights Expression Languages (REL) allow to define usage control policies for digital objects. RELs often define only an abstract policy model which is accompanied by an additional Rights Data Dictionary (RDD). The REL defines a basic syntax shared by all policies while the RDD provides a vocabulary for creating specific policies. The PLUS License Data Format (LDF)[3] provides a lightweight RDFS ontology for creating usage

[1] `http://www.w3.org/wiki/WebAccessControl`, last accessed: 01/21/16
[2] `http://www.w3.org/wiki/WebID`, last accessed: 01/21/16
[3] `http://www.useplus.com`, last accessed: 01/21/16

policies for digital images. The policies are primarily designed for human recipients and cannot directly be enforced by a technical system. Instead, the design of PLUS focuses on a technical support for communicating usage policies between all involved parties. ccREL [3] is a lightweight RDFS ontology primarily designed for describing Creative Commons[4] licenses. Such licenses describe actions that may, must, or must not be applied to the digital good. In order to be easy to use, there are six predefined licenses which can be applied to arbitrary goods. Although ccREL can generally be extended with additional terms, using such terms may lead to licenses which do not correspond to Creative Commons licenses. The Linked Data Rights ontology (LDR) [253, 254] is a lightweight OWL [302] ontology which supports usage control licenses for linked data resources. Although it defines a few terms itself, it is mainly designed to be extended with additional terms for particular use cases. The Metadata Encoding and Transmission Standard (METS) [93] is a general language model for describing different types of metadata of digital resources. METS itself only defines a basic XML language structure which must be extended with additional vocabularies in order annotate the digital resource. METSRights[5] is an example vocabulary which provides a very basic REL. It only allows to define which parties are allowed to perform which actions on a digital resource. It does not provide any means for defining prohibitions. More complex usage control languages are MPEG-21 REL [306] and the Open Digital Rights Language (ODRL) [156, 157]. MPEG-21 REL is the successor of XrML [308] and shares the same basic architecture [307]. It is part of MPEG-21 [65] which is a language framework similar to METS for annotating digital resources with different types of metadata. Both MPEG-21 REL and ODRL can be used for the same applications and allow to create almost arbitrary usage control policies. MPEG-21 REL is an XML-based language while ODRL provides an abstract language model which can be expressed with different encodings. The current version of ODRL provides an OWL ontology [199] as well as encodings for XML [155] and JSON [221]. Although both MPEG-21 REL and ODRL define their own REL and RDD, using the default RDD is not mandatory when creating specific policies. Instead, the creation of user-defined RDDs are also possible. In ODRL, user-defined RDDs are called profiles. An example profile is RightsML [162] which is designed for managing usage rights in the news industry.

In summary, RELs allow to describe which users are permitted to perform which actions on which digital resources. Contrary to flow control languages, these descriptions are rather abstract and must be interpreted manually in order to enforce them on a technical system. The more abstract a particular policy is, the more interpretations and implementations of the same policy are possible. For example, ODRL's RDD defines the action *anonymize* as the process to "anonymize all or parts of the Asset" [157]. Although an additional comment in the RDD further explains this as the process to "remove identifying particulars", it still remains unclear what the identifying particulars cover in detail. Thus, the anonymizing action cannot be directly enforced by a computer system since the system does not have a precise understanding of what the identifying

[4] `http://creativecommons.org`, last accessed: 01/21/16
[5] `http://www.loc.gov/standards/rights/METSRights.xsd`, last accessed: 01/21/16

particulars are. Instead, enforcing the action on a technical system requires manual interpretation and therefore human interaction. However, since RELs usually do not provide a precise human-readable description of their policies, different interpretations of the same policy may be considered as valid.

3.1.3. Flow Control Languages

Flow control languages are primarily designed for managing a closed network environment supervised by a single organization or institution. The languages aim at easing the configuration of the network by mapping high-level organizational security policies to different network systems such as routers and switches. The XML-based firewall metamodel proposed by Cuppens et al. [85], the UML-based DEN-ng [286], and the OWL-based Ontology-Based Policy Translator (OPoT) [21] provide languages for describing flow control regulations. While both the firewall metamodel and DEN-ng merely focus on low-level routers and do not directly support communication end points such as web servers, OPoT also covers different nodes of a communication path including the end systems. The firewall model of Cuppens et al. only supports permitting rules. It does not provide any means for defining prohibited communication flows. Any communication that is not explicitly allowed is considered to be forbidden. OPoT uses a set of twelve predefined basic policies each of which covers a particular use case. A basic policy can be considered as a template for implementing a specific organizational security policy. In order to actually enforce such an organizational policy, a corresponding basic policy has to be chosen and mapped to the current network environment. This mapping corresponds to filling in the basic policy with specific IP addresses and other implementation details.

In summary, flow control languages focus on regulating the flow of communication within a closed network environment which is centrally administrated by a single entity. The existing languages are designed to allow a direct enforcement of their policies by network nodes without requiring any further interpretation. This is achieved by precisely describing the actions that must be performed by the nodes including all necessary parameters such as IP addresses, port numbers, and communication protocols.

3.1.4. General Purpose Languages

General purpose languages do not focus on one particular type of information regulation but rather follow a more general approach in order to cover several scenarios such as access control or flow control. KAoS [298], Rei [166], and Ponder [88] are examples of such languages. Since these languages support different types of control policies, they cannot be clearly categorized as access control, usage control, or flow control. KAoS is based on OWL and allows to create policies that describe which systems a user can access within a closed management environment such as an organization. An example of such a policy is granting a user access to a specific server. However, KAoS only allows to regulate access to the server or its provided services and cannot further distinguish between the data hosted by them. More specifically, KAoS does not directly provide

any means for regulating the content hosted by a server. A KAoS policy can generally be enforced by any application-level communication system including content providers, content consumers, and application-level proxy servers. Contrary to KAoS, both Rei and Ponder allow to define document-centric policies. This allows to create more precise access control policies. Rei is based on OWL and merely considers reasoning about policies and is not designed for enforcing them [294]. Ponder uses its own low-level object-oriented language syntax which is not compatible with W3C standards such as XML or RDF. The language is able to define policies which can be enforced on the content providers' server, the end users' clients, as well as on intermediary communication nodes. However, due to its low-level descriptive language, Ponder is not able to cover different levels of abstraction on the same regulation [294] such as organizational or legal background information.

In summary, the applicability of a general purpose language as well as the implementation and enforcement of its policies heavily depends on the language's design. While KAoS and Rei focus on a more abstract view of a policy, Ponder merely covers its technical implementation details.

3.1.5. Content Labeling Schemes

Content labeling schemes allow to annotate digital resources with additional metadata describing their topic. Although these schemes usually do not provide a policy language for regulating information processing, they are specifically designed to be used together with such languages. The Restricted to Adults (RTA) label[6] and age-de.xml [264] allow to annotate web resources with an age category. Both formats are used by child protection software such as Net Nanny[7] or the Jugendschutzprogramm[8] for prohibiting minors from accessing adult web content. The RTA label is a simple label for flagging adult content. It consists of a single string which represents the age category of all adults. The label is either directly included in a web page or part of the HTTP response that a client receives from the server when requesting the page. In contrast, the XML-based age-de.xml supports arbitrary age categories which can be defined for single web pages or whole web sites. Similar to the RTA label, age categories defined with age-de.xml can also be included directly in the affected web pages or into the HTTP responses that contain the web pages. The Platform for Internet Content Selection (PICS) [183] supports more complex descriptions of web content than the RTA label or age-de.xml. It defines a proprietary format for annotating web content with arbitrary labels. A label consists of several ratings which contain a string and a numerical value. Labels can refer to single web pages or complete web sites. PICSRules [103] provides a policy language for allowing or denying access to web pages based on PICS labels or the pages's URLs.

In summary, content labeling schemes focus on the description of web content. With the exception of PICS, the discussed schemes do not provide a policy language for describing how these annotations shall be used for regulating information processing. How-

[6] `http://www.rtalabel.org`, last accessed: 01/21/16
[7] `https://www.netnanny.com/support/changelog/`, last accessed: 01/21/16
[8] `http://www.jugendschutzprogramm.de/faq6.php`, last accessed: 01/21/16

ever, content labeling schemes can also be used together with other policy languages. The policy language InFO described in this chapter also supports the description of web content. Such descriptions can either be modeled by using the InFO vocabulary or by integrating one of the existing content labeling schemes.

3.2. Requirements for a Policy Language

The policy language InFO defines several ontology design patterns [237, 121] for describing policies for regulating information flow control on the Internet. These policies cover both a human-readable description of their actual meaning as well as the technical implementation details for enforcing them on a particular communication node. A particular flow control policy models a specific use case and consists of several flow control rules that implement this use case. Multiple use cases require several policies. Based on these general objectives, this section defines the specific requirements for InFO which are divided into functional requirements (**RA.F.***) and non-functional requirements (**RA.N.***). Functional requirements describe the services and functions that a system must provide [282]. In the case of the InFO policy language, these requirements define the content of a particular flow control policy which must cover all details for regulating information flow control. Thus, the functional requirements for InFO focus on the language's vocabulary. On the other hand, non-functional requirements define general properties and constraints of a system [282]. While functional requirements basically focus on a particular aspect of the system, non-functional requirements are usually abstract and cover the system as a whole [282]. In the case of InFO, non-functional requirements define restrictions of the language's design and implementation. Although there is no clear distinction between functional and non-functional requirements [182, 282], separating both types of requirements can help to better identify the purpose of a particular requirement. The specific requirements for the policy language InFO derive from the scenario for regulating Internet communication presented in Section 2.1 and on the related work summarized in Section 3.1. InFO must support the following functional requirements:

RA.F.1: Supporting different types of enforcing communication nodes

The policy language must be able to describe flow control policies which can be enforced by different intermediate communication nodes on the Internet such as routers (**RA.F.1.1**), name servers (**RA.F.1.2**), and application-level proxy servers (**RA.F.1.3**). As outlined in Section 2.1, different enforcing nodes are in fact used in practice for regulating information processing on the Internet [212, 321, 79]. Thus, these nodes must also be supported by the InFO policy language. The example computer network introduced in the scenario in Section 2.1.1 includes all three types of enforcing nodes.

RA.F.2: Operationalizing the policies

The interpretation of a particular control policy by a corresponding enforcing node requires a detailed description of the communication flow that shall be regulated.

This description must contain all relevant parameters such as the IP addresses of the communicating parties, the URL of the web content, or the domain name of the web server. The policy language InFO must be able to describe such parameters. Each of the enforcing nodes of the scenario requires their own parameters in order to operate properly.

RA.F.3: Supporting different modalities of control rules

The InFO policy language must be able to describe control rules that either allow (**RA.F.3.1**) or deny (**RA.F.3.2**) a communication flow between two communicating parties. Supporting both types of rule modalities allows a more flexible creation of regulations such as regulations based on whitelisting or blacklisting [19]. Blacklisting allows every communication which is not explicitly forbidden whereas whitelisting prohibits all communication which is not explicitly allowed. Prohibiting a particular communication flow can be implemented in different ways such as redirecting to a different communication party or preventing the establishment of the communication channel. InFO must also support such different implementations.

RA.F.4: Resolving rule conflicts

Conflicts between two control rules occur when a particular flow of communication is allowed by one rule and prohibited by another one. The contradicting rules may appear in the same control policy (**RA.F.4.1**) or in different policies (**RA.F.4.2**). The policy language InFO must provide mechanisms for resolving both types of rule conflicts. In the scenario, the regulations derive from different sources such as ContentWatch and JusProg which may lead to conflicting rules.

RA.F.5: Identifying involved parties

The policy language must provide information about the parties who are responsible for a particular policy. This includes the party who technically enforces the policy (enforcer) (**RA.F.5.1**), the party who provides the details for this enforcement (provider) (**RA.F.5.2**), and the party who legislates the enforcement (legislator) (**RA.F.5.3**). Distinguishing between all three parties can help in achieving more transparency in the regulation's implementation. In the scenario of Section 2.1, access providers such as the German Telecom act as regulation enforcers. Examples of regulation providers are the BKA and ContentWatch and examples of regulation legislators are the German states and their federation [60].

RA.F.6: Identifying regulated content

Regulating access to web content may cause unwanted side-effects such as over-blocking or under-blocking [249]. Over-blocking affects the access to more content than is actually intended. Under-blocking covers only parts of the content to be regulated. In order to reduce such unwanted side-effects, InFO must allow to identify the content or its hosting server as precisely as possible. Examples of precise identifiers are the IP address of the hosting web servers, the domain names of the websites, and the URLs of the individual web pages.

RA.F.7: Classifying regulated content

Access regulation is often based on the content's topic such as online gambling, adult websites, or neo-Nazi propaganda. In order to provide a clear reason for why access to particular web content is regulated, the topic of the content must be identified. Thus, InFO must provide means for representing the classification of content. In the scenario, the classification of the provided web content is given by the label of its respective web server, e. g. each "Weather Server" provides information about the weather.

RA.F.8: Providing users' access location

Each information flow control policy is ultimately based on a set of laws issued for and active in a specific country. Since an end user's current location also defines the laws she must abide by, InFO must relate a user to her corresponding location. The scenario presented in Section 2.1 distinguishes between end users of the three example countries USA, Germany, and Saudi Arabia.

RA.F.9: Providing background information

The regulation represented by a flow control policy is authorized by a legal foundation and/or motivated by an organizational code of conduct. In order to enrich a policy with human-readable explanations, InFO must be able to attach corresponding background information to the policy in form of its legal justification (**RA.F.9.2**) and/or organizational motivation (**RA.F.9.1**). As outlined in the scenario, §86 of the German Criminal Code [62] is an example of a legal justification which prohibits the distribution of neo-Nazi material. The code of conduct of the German Telecom [91] is a statement to actually enforce this law as well as the local laws of other countries in which the Internet access provider operates.

Besides these functional requirements, the InFO policy language must also fulfill the following non-functional requirements:

RA.N.1: Complying with standards

The policy language must provide a particular encoding for creating specific policies. This encoding must be based on common standards such as XML or RDFS. Avoiding a proprietary syntax simplifies the process of interpreting and implementing particular regulation policies on an enforcing system. Furthermore, it eases the creation and distribution of the policies for all involved parties. In the scenario, different parties such as the BKA and the German Telecom interact with each other by exchanging regulation details. Using a common standard for encoding these details improves the interoperability between the parties.

RA.N.2: Supporting a modular design

The InFO policy language must have a modular design in which each module implements a particular function of the language [266]. This allows a flexible use of the actually needed parts of the ontology. For example, requirement **RA.F.1** states that the policy language must be able to support different systems as enforcing

nodes. Each type of enforcing nodes requires specific parameters and enforcement details, which may be irrelevant for other enforcing nodes. By designing a particular module for routers, name servers, and proxy servers, a specific enforcing node only needs to implement its respective regulation module. In the scenario, each enforcing node only implements those regulation modules which it can technically enforce.

RA.N.3: Supporting an extensible design

Although InFO must natively support three different types of enforcing nodes, this support can only cover a limited set of all possible attributes and functions that a specific node may have. For example, the build-in support for routers as enforcing nodes does not guarantee a complete support for all functions of all possible routers like Cisco's 3945 Integrated Services Router[9]. An extensible design [266] allows to further enrich InFO with product-specific concepts. Furthermore, such a design can also be used for defining new concepts of future regulation mechanisms.

The fulfillment of all functional and non-functional requirements by the InFO pattern system and a comparison with the state of the art and related work is provided in Section 3.6.

3.3. Design of the InFO policy language

This section presents the pattern-based design of the InFO policy language. The design is based on the state of the art and related work discussed in Section 3.1 as well as on the requirements stated in Section 3.2. InFO is a set of several ontology design patterns [237, 121] for describing flow control policies which regulate the exchange of information on the Internet. Ontology design patterns are adapted from software engineering. They provide both a description of a specific, recurring modeling problem of a particular modeling context and present a proven, generic solution to this problem [67]. The provided solution consists of a description of the required components, their relationships and responsibilities, and the possible collaboration between these components [67]. Similar to a software design pattern, an ontology design pattern is also independent of a particular application domain [117] and can be used in a variety of different application contexts. Each pattern of InFO implements a different aspect of controlling the flow of information that distinguishes it from the other patterns. The patterns are not a collection of independent ontology design patterns but are instead designed to be used together in order to create flow control policies. Thus, InFO corresponds to a so-called *pattern system* [67]. The pattern system reuses and extends design patterns from the foundational ontology DOLCE+DnS Ultralite (DUL) [119] and the Ontopic core ontology[10]. It is implemented using the Web Ontology Language (OWL) [302] and axiomatized using Description Logics [16]. This section first briefly describes these reused ontologies and

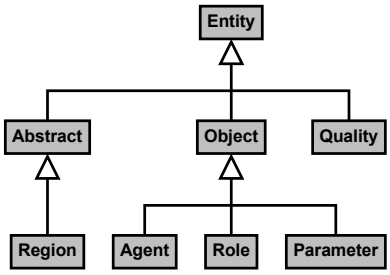
[9] `http://www.cisco.com/c/en/us/products/routers/3945-integrated-services-router-isr/index.html`, last accessed: 01/21/16

[10] `http://ontologydesignpatterns.org/ont/dul/ontopic.owl`, last accessed: 01/21/16

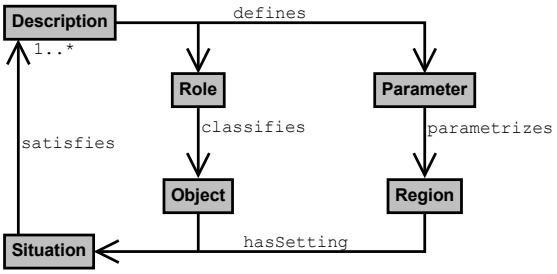
gives an overview of the InFO pattern system. Subsequently, each pattern is explained in more detail.

3.3.1. Modeling Methodology and Reused Ontologies

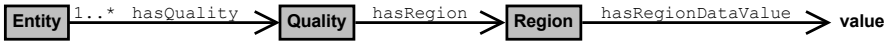
InFO uses DOLCE+DnS Ultralite (DUL) [119] as a foundational ontology since it provides a rich axiomatization based on several design patterns. In addition, the use of DUL has proven to be a good design choice [266]. Figure 3.2 depicts the main classes and reused design patterns of DUL. The ontology defines the class `Entity` and its subclasses `Object`, `Quality`, and `Abstract`, which are depicted in Figure 3.2a. `Object`s are entities that exist in time and space such as `Agent`s. They are either physical objects such as natural persons or social objects such as `Role`s. A `Quality` describes a feature of an `Entity` whose feature value is specified by a `Region`. `Abstract` refers to entities that do neither have spatial nor temporal features. `Region`s are `Abstract`s and represent data value spaces such as time intervals. The relations between the three classes `Entity`, `Quality`, and `Region` are covered by DUL's qualities and quality region pattern [119] which is depicted in Figure 3.2c. The pattern distinguishes between a feature of an entity and its corresponding feature value. While a feature is inextricably linked to the entity that has the feature, its value can also exist independently. Apart from this pattern, InFO also uses other design patterns from DUL which are the collection pattern, the participation pattern, the sequence pattern, the information realization pattern, and the description and situation (DnS) pattern.



(a) Basic class hierarchy



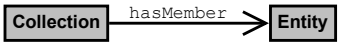
(b) Description and situation pattern



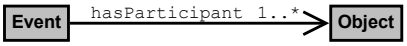
(c) Qualities and quality region pattern (cf. [265])

Figure 3.2.: Important classes and patterns of DOLCE+DnS Ultralite (DUL).

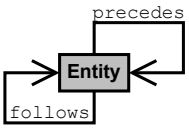
The collection pattern defines the property `hasMember` which can be used for describing the relationship between a collection and its elements. As depicted in Figure 3.2d, the property links a particular collection to each of its members. The participation pattern models the relationship between an event and all involved objects. The pattern is depicted in Figure 3.2e and defines the property `hasParticipant` to link an event to any social object or physical object. The sequence pattern depicted in Figure 3.2f defines the two properties `follows` and `precedes`. These properties are inverse to each



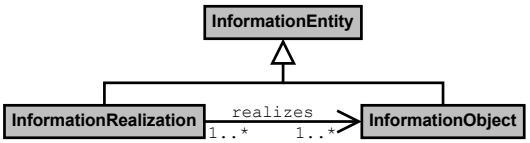
(d) Collection pattern



(e) Participation pattern



(f) Sequence pattern



(g) Information realization pattern

Figure 3.2.: Important classes and patterns of DOLCE+DnS Ultralite (DUL). *Continued from previous page.*

other and allow to describe a relative order between two entities. A sequence consisting of multiple entities can be created by using the properties for modeling a relationship between all pairs of consecutive entities. The information realization pattern [44] is shown in Figure 3.2g. It defines the classes `InformationObject` and `InformationRealization` as well as the property `isRealizedBy`. An `InformationObject` is an abstract piece of information and is realized by a physical object or a digital resource. The entity realizing the abstract piece of information is called an `InformationRealization`. An example of an `InformationObject` is Shakespeare's *Hamlet*. Hamlet is an abstract work which can be encoded as a written book or performed as a play in theater. The book and the performed play both correspond to two different `InformationRealization`s of the same abstract piece of information. The DnS pattern [120] is a central design pattern of DUL and models n-ary relationships between arbitrary entities. The pattern strictly distinguishes between a conceptual view of the relationships and a specific state of affairs. The conceptual view is modeled as a `Description` and defines the context and functions of all involved entities. Functions can be modeled as `Role`s or `Parameter`s. The state of affairs corresponds to a `Situation` which acts as the setting for all entities of the relation. A `Situation` satisfies a `Description` by mapping all abstract functions to specific entities such as `Object`s or `Region`s. A simplified depiction of the DnS pattern is shown in Figure 3.2b.

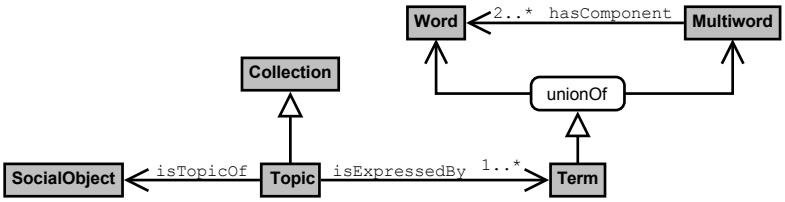


Figure 3.3.: Topic pattern of the Ontopic ontology.

InFO also imports the Ontopic core ontology which is an extension of DUL. The ontology defines the class `Topic` and a corresponding design pattern, which is depicted in Figure 3.3. A `Topic` is a collection of semantically related social objects such as `InformationObject`s or `Role`s. A specific topic is expressed using one or more descriptive

`Term`s. A term is either a single word or a multiword consisting of several single words. Multiwords can be used for modeling compound words such as the word “neo-Nazi”. An example `Topic` is “neo-Nazi propaganda” which is basically a collection of all neo-Nazi propaganda material. This topic can be expressed using the single word “propaganda” and a multiword consisting of the two single words “neo” and “Nazi”.

3.3.2. Overview of the pattern system

An overview of the InFO pattern system is depicted in Figure 3.4. The overview distinguishes between foundational ontologies, core ontologies, and domain ontologies. Foundational ontologies do not focus on a specific use case or application domain. Instead, they define a broad set of generic concepts and axioms for describing the world [266]. Foundational ontologies can be used in various fields and form a basis for creating other ontologies such as core ontologies or domain ontologies [222]. Core ontologies further specify a particular field by providing more detailed axioms and concepts [266]. They

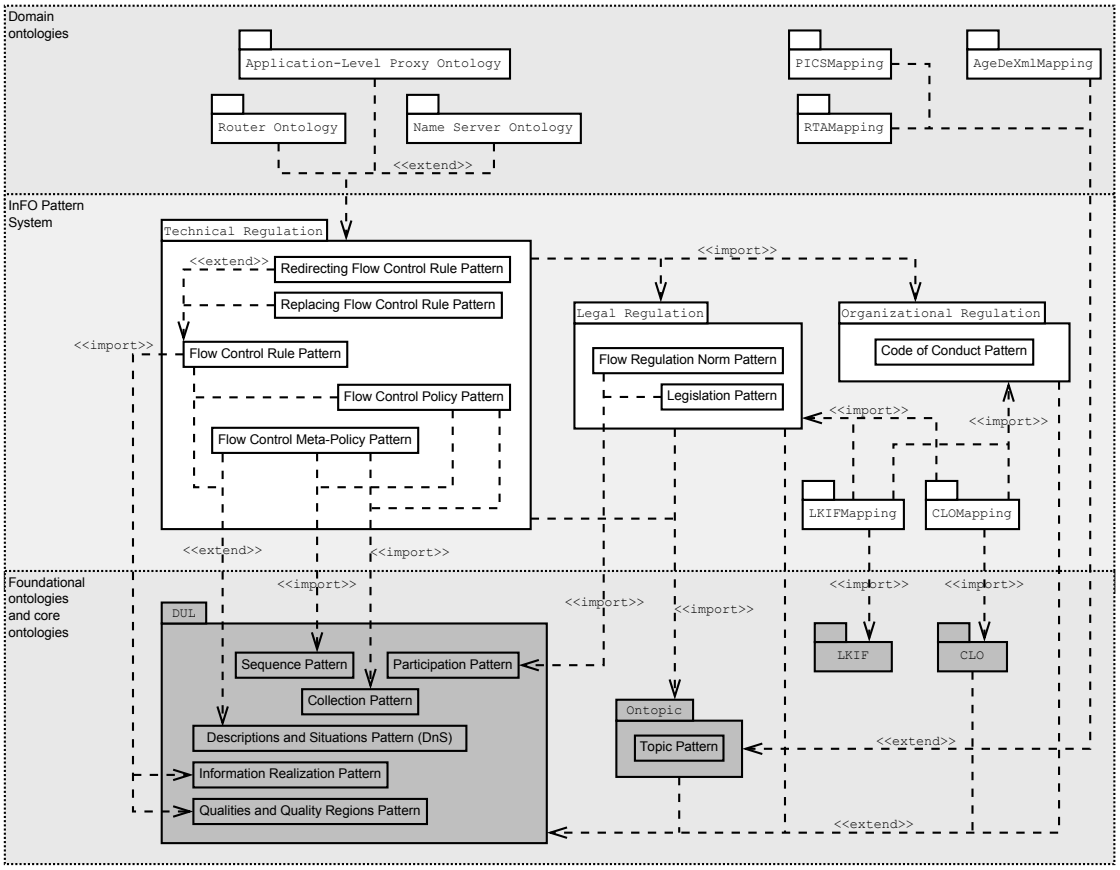


Figure 3.4.: Overview of the InFO pattern system. Dark gray elements are external ontologies reused by InFO whereas white elements are patterns of InFO or their domain-specific extensions.

are usually built upon foundational ontologies and serve as the modeling basis for domain ontologies [266]. Thus, core ontologies cover various domains within a particular field [222]. Finally, domain ontologies focus on a particular domain within a specific field [222]. They provide concepts and axioms which are only relevant to the particular domain and are not used in other domains [102]. Domain ontologies can be based on foundational ontologies and core ontologies [222].

The InFO pattern system represents a core ontology for modeling regulations of Internet communication. It is based on the foundational ontology DUL and on the core ontology Ontopic. Its patterns are subdivided into the Technical Regulation patterns, the Organizational Regulation patterns, and the Legal Regulation patterns. Each category of patterns implements a specific aspect of information flow control. The Technical Regulation patterns cover the description of all technical regulation details which are InFO's main focus. The Organizational Regulation patterns and the Legal Regulation patterns enrich the technical policies with human-readable descriptions. Domain specific extensions of the InFO pattern system are provided for routers, proxy servers, and name servers as corresponding domain ontologies. As depicted in Figure 3.4, the InFO pattern system also allows to integrate existing content labeling schemes such as PICS [183], the RTA label, or age-de.xml [264].

In detail, the Technical Regulation consists of five different patterns which are based on DUL's DnS pattern depicted in Figure 3.2b. Each pattern models a different flow control aspect which defines the context of the involved entities. The DnS pattern is used since it allows entities such as computer systems to participate in a specific context by fulfilling a specific function. Policies are basically descriptions of regulations and thus are modeled as subclasses of `Description`. Their implementation leads to a `Situation` where each concept defined by the policy is fulfilled by a corresponding entity. The Flow Control Rule Pattern describes a flow control rule which covers the technical regulation details for a particular communication flow. The pattern describes whether or not this flow of communication is to be allowed or denied and thus implements requirements **RA.F.3.1** and **RA.F.3.2**. The regulation details include an identifier (**RA.F.6**) and a classification (**RA.F.7**) of the content to be regulated as well as the location of the user accessing the content (**RA.F.8**). All technical regulation details are provided by an agent who acts as the rule data provider (**RA.F.5.2**). The Flow Control Rule Pattern is further extended by the Redirecting Flow Control Rule Pattern and the Replacing Flow Control Rule Pattern, which allow the creation of more complex rules for prohibiting a communication flow. Several flow control rules sharing the same purpose are combined to form a flow control policy, which is provided by the Flow Control Policy Pattern. In order to further describe the rules' purpose, a flow control policy is linked to an organizational code of conduct and/or a legal foundation. Therefore, the Flow Control Policy Pattern imports the Organizational Regulation patterns and the Legal Regulation patterns. A flow control policy also covers the enforcing party (**RA.F.5.1**) and the enforcing system in form of routers (**RA.F.1.1**), proxy servers (**RA.F.1.3**), and name servers (**RA.F.1.2**) which implement the flow control. Possible conflicts between rules of one policy (**RA.F.4.1**) or multiple policies (**RA.F.4.2**) are resolved by using

a meta-policy described with the Flow Control Meta-Policy Pattern. A meta-policy is basically a collection of all flow control policies which are enforced by the same system.

The Organizational Regulation patterns and the Legal Regulation patterns enrich the Technical Regulation with human-readable descriptions. The Code of Conduct Pattern fulfills requirement **RA.F.9.1** by defining concepts for describing an organizational code of conduct as well as its legal background. The Flow Regulation Norm Pattern defines the legality of a particular communication flow and implements requirement **RA.F.9.2**. The Legislation Pattern pattern allows to model how the legal norm conceptualized in the Flow Regulation Norm Pattern was actually created. This corresponds to a legislative procedure and allows to specify the legislator of the norm to fulfill requirement **RA.F.5.3**. The Organizational Regulation patterns and the Legal Regulation patterns do not define all details for covering human-readable descriptions themselves. Instead, the patterns define generic concepts which can be integrated into existing legal ontologies such as the Legal Knowledge Interchange Format (LKIF) [146, 147] or the Core Legal Ontology (CLO) [123, 118]. This allows to reuse all concepts defined in these ontologies together with the Technical Regulation of InFO. An external ontology is integrated by using a corresponding mapping ontology like LKIFMapping or CLOMapping as shown in Figure 3.4.

The Technical Regulation patterns only define the basic structure of the technical regulation details. This structure is independent of any particular enforcing node. Thus, policies for particular types of enforcing nodes are described using domain ontologies which are specialized from the Technical Regulation patterns. Policies for IP-based regulation are described using the Router Ontology (**RA.F.1.1**), policies for the Domain Name System use the Name Server Ontology (**RA.F.1.2**), and policies for proxy servers are based on the Application-level Proxy Ontology (**RA.F.1.3**). Each domain ontology provides concepts and axioms for precisely specifying all parameters required for implementing the flow control (**RA.F.2**) on a specific type of enforcing node. For example, the Router Ontology contains concepts of IP-based rules which must be enforced by routers, the Name Server Ontology provides concepts and axioms for modeling rules based on domain names, and the Application-level Proxy Ontology provides concepts for modeling URL-based rules which are enforced by application-level proxy servers. In addition to these three domain ontologies, it is also possible to create new domain ontologies by extending the Technical Regulation patterns.

The fulfillment of all functional requirements is summarized in Table 3.1. The non-functional requirements standards compliance **RA.N.1**, modularity **RA.N.2** and extensibility **RA.N.3** cannot be mapped to a particular pattern, since they affect the InFO pattern system as a whole. Requirement **RA.N.1** is achieved by using OWL as the modeling language. Requirement **RA.N.2** is addressed by InFO's modular design and its use of DUL as modeling basis. Requirement **RA.N.3** is supported by allowing the creation of new domain ontologies besides the already existing ones. Furthermore, it is also possible to import other legal ontologies as the foundation for describing the legal and organizational background. In summary, InFO covers all functional as well as non-functional requirements defined in Section 3.2. A detailed discussion of these requirements and a comparison with the state of the art and related work is provided in

Table 3.1.: Functional requirements and their implementations by the InFO patterns.

	Requirement	Implementation
RA.F.1.1	Support for routers as enforcing nodes	Router Ontology
RA.F.1.2	Support for name servers as enforcing nodes	Name Server Ontology
RA.F.1.3	Support for proxy servers as enforcing nodes	Application-level Proxy Ontology
RA.F.2	Operationalization of policies	Router Ontology, Application-level Proxy Ontology, Name Server Ontology
RA.F.3.1	Support for allowing rules	Flow Control Rule Pattern
RA.F.3.2	Support for denying rules	Flow Control Rule Pattern
RA.F.4.1	Rule conflict resolution for single policies	Flow Control Meta-Policy Pattern
RA.F.4.2	Rule conflict resolution for multiple policies	Flow Control Meta-Policy Pattern
RA.F.5.1	Identification of regulation enforcer	Flow Control Policy Pattern
RA.F.5.2	Identification of regulation provider	Flow Control Rule Pattern
RA.F.5.3	Identification of regulation legislator	Legislation Pattern
RA.F.6	Identification of regulated content	Flow Control Rule Pattern
RA.F.7	Classification of regulated content	Flow Control Rule Pattern
RA.F.8	User's access location	Flow Control Rule Pattern
RA.F.9.1	Organizational background	Code of Conduct Pattern
RA.F.9.2	Legal background	Flow Regulation Norm Pattern

Section 3.6. In the following, each pattern of InFO is described in more detail. First, the different Technical Regulation patterns are explained. Afterwards, the Organizational Regulation patterns and the Legal Regulation patterns are described. Finally, the integration of existing legal ontologies and content labeling schemes into the InFO pattern system is covered.

3.3.3. Flow Control Rule Pattern

The Technical Regulation patterns cover three different patterns for expressing flow control rules. These patterns are the Flow Control Rule Pattern, the Redirecting Flow Control Rule Pattern, and the Replacing Flow Control Rule Pattern. All three patterns define several communication aspects such as the communicating end points and the transmitted content. Each communication aspect is modeled using the same basic structure which is depicted in Figure 3.5. This structure defines a communication aspect as a `Role` which is played by an instance of the class `Object` or one of its subclasses. Example objects are client systems, web servers, or web pages. An object is identified by its features which are described as a quality of the object. According to DUL's qualities and quality region pattern depicted in Figure 3.2c, the actual values of these features are modeled as subclasses of `Region`. Possible identifiers for client systems and web servers are IP addresses and domain names, possible identifiers for web pages are URLs, and possible identifiers for communication channels are protocol names such as HTTP or FTP. Each communication aspect is further specified by a corresponding `Specifier` which parametrizes its `Region`.

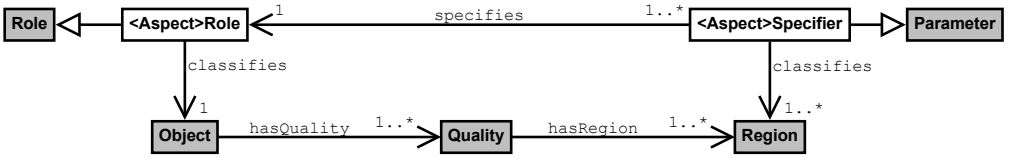


Figure 3.5.: Basic structure of a communication aspect in the Flow Control Rule Pattern. This structure uses DUL's qualities and quality region pattern. `<Aspect>` is a placeholder for `Sender`, `Receiver`, `Content`, and `Channel`. For example, a `SenderSpecifier` `specifies` a `SenderRole`.

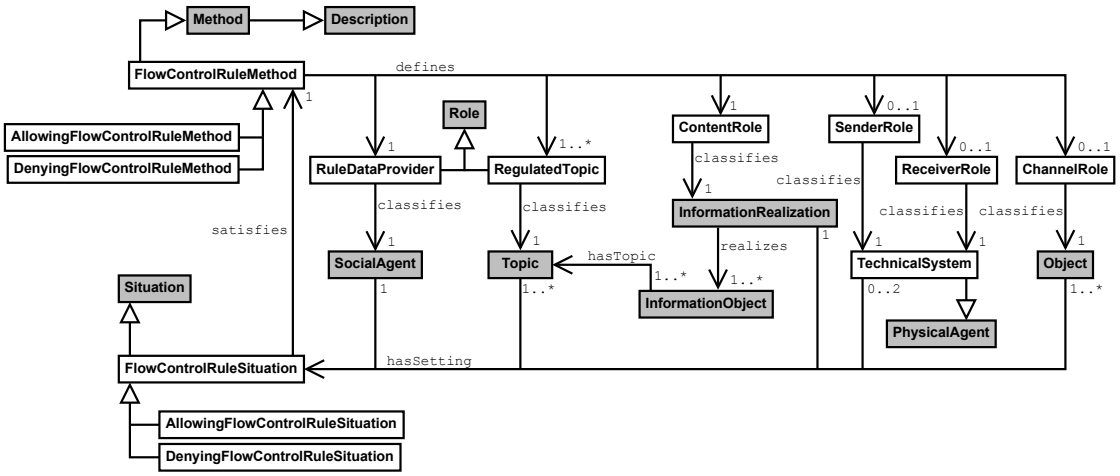


Figure 3.6.: Flow Control Rule Pattern.

The main class of the Flow Control Rule Pattern is `FlowControlRuleMethod` which is modeled as a subclass of DUL's `Method`. A flow control rule regulates if the establishment of a particular Internet communication is to be allowed or denied. `FlowControlRuleMethod` itself does not specify whether the described flow control shall be allowed or prohibited. Instead, this is expressed by its subclasses `AllowingFlowControlRuleMethod` and `DenyingFlowControlRuleMethod`. The Flow Control Rule Pattern allows to describe such a regulation by associating the regulating rule with four different aspects of an Internet communication. These aspects are defined according to Shannon's communication model [278] and cover the sender and receiver of the communication as well as the transmitted content and the communication channel. All four aspects are modeled using the basic structure depicted in Figure 3.5 by replacing the placeholder `<Aspect>` with `Sender`, `Receiver`, `Content`, and `Channel`, respectively. Such a generic solution allows the Flow Control Rule Pattern to cover almost any arbitrary type of information flow. For reasons of simplicity, the depiction of this pattern provided in Figure 3.6 only shows the aspect's role as well as the classified object. As a flow control rule regulates the establishment of a technical communication flow, the sender and receiver of this communication are modeled as `TechnicalSystem`s. The content is modeled as an instance of `InformationRealization` since the rule operates on a specific digital data

file and not on an abstract piece of information. If a flow control rule does not specify one of the four communication aspects, it will be evaluated for all possible aspects. For example, if a rule does not explicitly define a sender, it will be evaluated for all senders.

Besides the four communication aspects, the Flow Control Rule Pattern also defines the provider of the rule as well as a regulated topic. `RuleDataProvider` represents the party who creates a flow control rule by providing all information for technically enforcing it. This includes the identifiers of all communication aspects such as IP addresses, domain names, or URLs. Possible `RuleDataProvider`s are given in the scenario in Section 2.1 and include the BKA and KACST. The regulated topic describes the content whose transmission is regulated by the rule. Example topics are neo-Nazi propaganda or pornography. The content's topic is described using the topic pattern of the Ontopic ontology depicted in Figure 3.3. For simplicity reasons, Figure 3.6 only shows the class `Topic` and not the complete topic pattern. Since a topic is independent from any specific encoding and primarily associated with abstract piece of information, the Ontopic ontology associates a topic with an `InformationObject`. In order to be able to associate a topic with an `InformationRealization`, the Flow Control Rule Pattern includes an `InformationObject` as an indirect link between the two classes. If additional descriptions of the content are desired, the `InformationObject` can be further described by using additional content labeling schemes such as age-de.xml and their integration into the InFO pattern system. The general process of such an integration is presented in Section 3.3.8 and the details are covered in Appendix B.2.

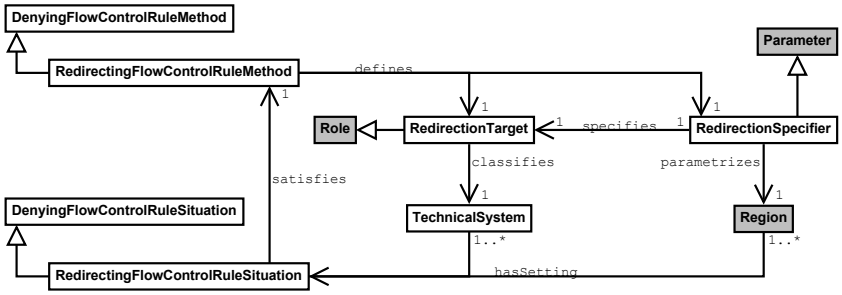


Figure 3.7.: Redirecting Flow Control Rule Pattern.

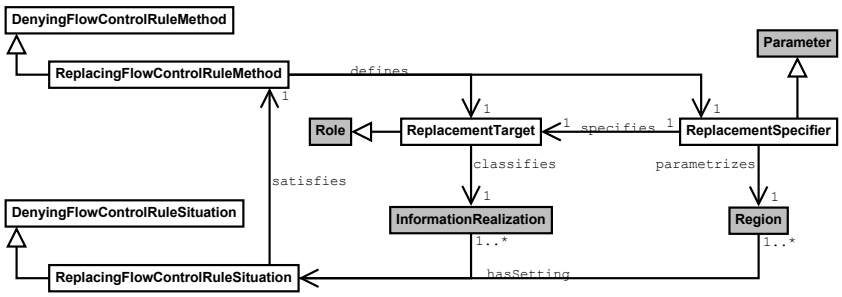


Figure 3.8.: Replacing Flow Control Rule Pattern.

The Flow Control Rule Pattern is extended by the Redirecting Flow Control Rule Pattern and the Replacing Flow Control Rule Pattern. The Redirecting Flow Control Rule Pattern allows to deny a particular communication flow by replacing the original receiver with a different, predefined receiver. The intended communication flow is therefore not possible. The pattern may be useful if the sender of the communication shall be redirected to another server which provides additional background information about the regulation. The Redirecting Flow Control Rule Pattern is depicted in Figure 3.7. It extends the Flow Control Rule Pattern by adding a `RedirectionTarget` which is modeled according to the basic structure for communication aspects as shown in Figure 3.5. The Replacing Flow Control Rule Pattern depicted in Figure 3.8 is similar to the Redirection Flow Control Rule Pattern. It also denies a particular communication flow by replacing one of its four basic communication aspects with a predefined value. More specifically, the Replacing Flow Control Rule Pattern replaces the intended content with a replacement content. The pattern extends the Flow Control Rule Pattern by defining the class `ReplacementTarget` which is also modeled according to the basic structure of communication aspects depicted in Figure 3.5.

3.3.4. Flow Control Policy Pattern

A flow control policy is a collection of multiple flow control rules sharing the same purpose. The Flow Control Policy Pattern depicted in Figure 3.9 allows to define such collections and associates them with a legal norm and/or code of conduct. Both the legal norm and the code of conduct express the purpose of a flow control policy and all of its rules in a human-readable form. Their usage and modeling is further described in Section 3.3.6. The Flow Control Policy Pattern also associates a flow control policy with one enforcing party and one technical enforcing system. The party is represented by a `SocialAgent` and acts as a `ResponsibleOperator`. Possible types of operators are natural persons and organizations. The system which technically implements the flow control is modeled as a `TechnicalSystem` in the role of an `EnforcingSystem`. Example systems are routers, name servers, and application-level proxy servers.

In order to resolve conflicting rules, the pattern provides two optional conflict resolution algorithms. The `LocalNonApplicabilityAlgorithm` covers the handling of such flow control rules which cannot be fully implemented on the enforcing system. An example of such rules is described below in Section 3.3.5. The `LocalConflictResolutionAlgorithm` defines how conflicts between two contradicting flow control rules of the same policy are resolved. Such conflicts occur when rules of the same policy share the same specifiers of their aspects but differ in the actual handling of the communication flow. For example, one rule is allowing the specified communication flow while another rule is prohibiting it. Before the algorithm is evaluated, all rules of the policy are ordered according to the rule priority algorithm of the policy's meta-policy which is further described in the next section. Both types of local conflict resolution algorithms are also provided as global algorithms in a policy's meta-policy. If a flow control policy does not specify a local conflict resolution algorithm, existing conflicts are resolved by using the corresponding algorithms of the meta-policy. The local algorithms are optional and are,

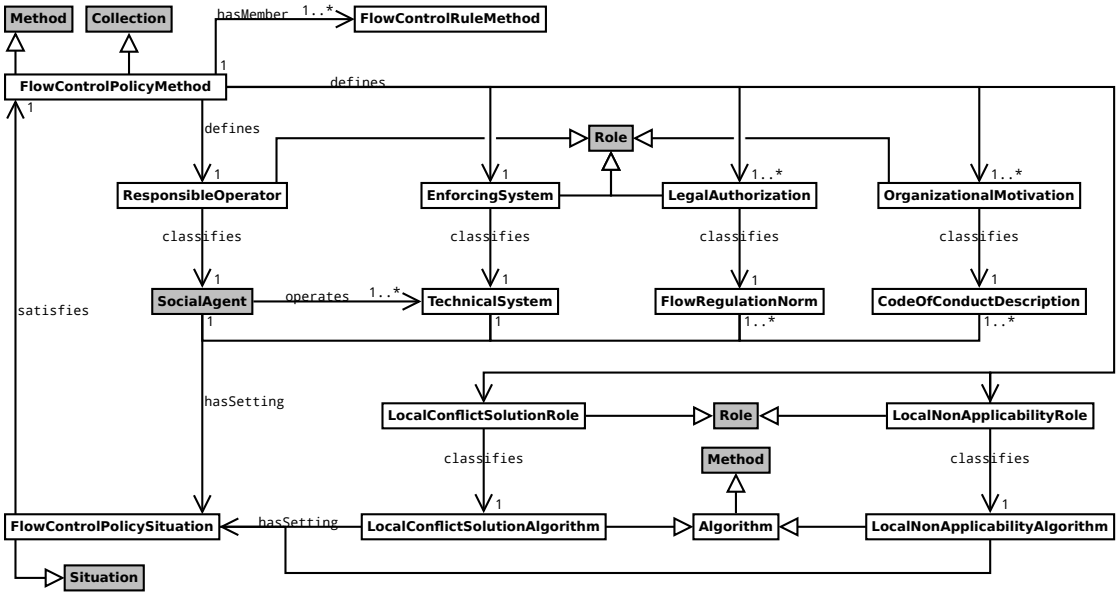


Figure 3.9.: Flow Control Policy Pattern.

if existing, applied before their respective global algorithm. This allows to overwrite the global algorithms within a particular policy.

3.3.5. Flow Control Meta-Policy Pattern

A flow control meta-policy provides several algorithms for resolving conflicts of flow control rules. These algorithms either resolve conflicts between two contradicting rules of one or more flow control policies or between a rule and its enforcing system. The Flow Control Meta-Policy Pattern depicted in Figure 3.10 provides a conceptualization for a meta-policy. It defines the class `FlowControlMetaPolicyMethod` as a collection of several flow control policies and four different conflict resolution algorithms. These algorithms are the `PolicyPriorityAlgorithm`, the `RulePriorityAlgorithm`, the `GlobalConflictResolutionAlgorithm`, and the `GlobalNonApplicabilityAlgorithm`. The algorithms are inspired by the policy languages XACML [205], DEN-ng [286], the Ontology-Based Policy Translator (OPoT) [21], Ponder [88], and ODRL [156, 157] as further described in Section 3.6. However, InFO provides a more fine-grained and flexible approach for solving conflicts than these policy-languages. Each algorithm covers a specific aspect of the conflict resolution process which is further described below. Unlike to the optional conflict resolution algorithms of a flow control policy, all global algorithms of a meta-policy are mandatory. The behavior of a particular algorithm is specified via a corresponding subclass of the algorithm type. For example, possible `GlobalConflictResolutionAlgorithms` are `IgnoreAffectedRulesAlgorithm` and `IgnoreAffectedPoliciesAlgorithm`. The former algorithm only discards the conflicting rules but leaves other rules of the same policy unchanged. The latter algorithm dis-

cards the whole policies which contain the conflicting rules. Additional algorithms are described in Appendix A. Apart from these algorithms, a flow control meta-policy also defines the enforcing system's default behavior via a `DefaultRule`. Each flow control rule covers a specific communication flow. If no rule can be applied to a particular communication, the `DefaultRule` will be used instead. This rule does not define any specific sender, receiver, content, or channel. Instead, it only covers those parameters necessary for the rule's implementation, e. g., redirection targets or replacement targets. A default rule will be evaluated for every communication as long as there is no other flow control rule which already covers that communication. Similar to the Flow Control Policy Pattern, the Flow Control Meta-Policy Pattern associates a meta-policy with one enforcing party and one technical enforcing system. Each enforcing system is related to exactly one flow control meta-policy and can implement multiple flow control policies and corresponding rules.

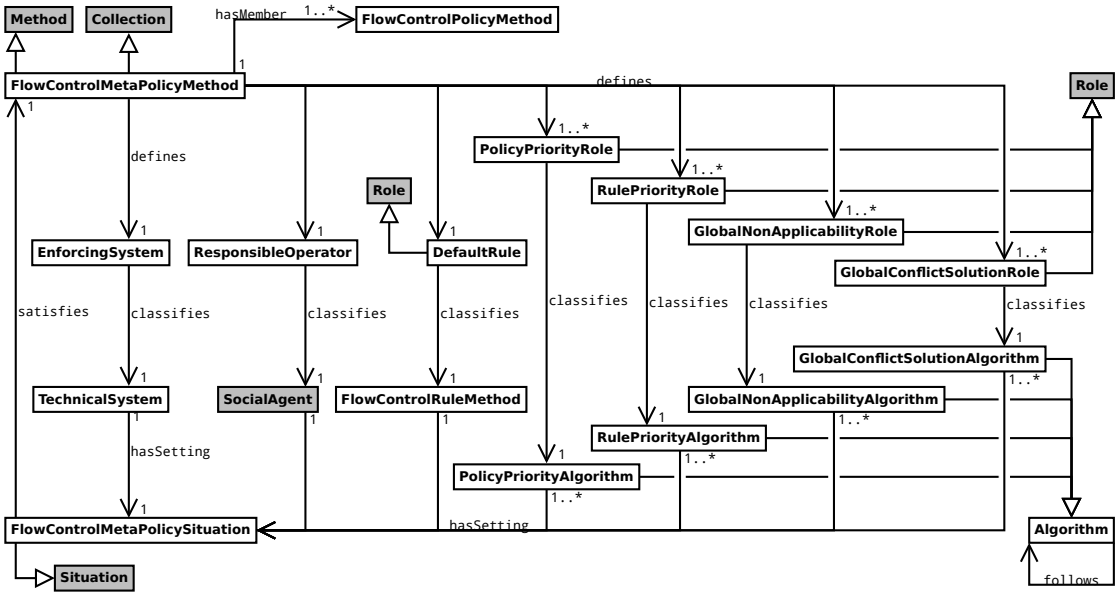


Figure 3.10.: Flow Control Meta-Policy Pattern.

Lupu and Sloman distinguish between two different categories of possible conflicts of rules which are *modality conflicts* and *application specific conflicts* [192]. Modality conflicts occur between two rules when the establishment of a particular flow of communication is allowed by one rule and prohibited by the other rule. InFO resolves modality conflicts with the `PolicyPriorityAlgorithm`, the `RulePriorityAlgorithm`, the `GlobalConflictResolutionAlgorithm`, and the optional `LocalConflictResolutionAlgorithm`. Application specific conflicts occur between a flow control rule and its enforcing system and correspond to an incompatibility between the two. Such an incompatibility exists if the rule uses ontological concepts which are unknown to the enforcing system. In this case, the enforcing system does not know the meaning of the unknown concepts and cannot implement the corresponding rule. Resolving applica-

tion specific conflicts is important as InFO's open design allows to define new types of rules by creating a corresponding subclass of `FlowControlRuleMethod`. However, if the enforcing system does not understand this new rule type, it cannot interpret it. For example, a new rule type `ReplaceWordOnWebPage` may be defined in order to replace offensive words on web pages by an application-level proxy server. If the enforcing proxy server does not know the semantics of this rule, it cannot replace any offensive word. Application-specific conflicts like these are handled by the `GlobalNonApplicability-Algorithm` and the optional `LocalNonApplicabilityAlgorithm` which are evaluated before applying any other conflict solution algorithm. All algorithms are evaluated by an enforcing system in the following order:

1 Apply the `LocalNonApplicabilityAlgorithm` to the rules of each flow control policy which defines such an algorithm. If a policy does not define a local non-applicability algorithm, this step is skipped.

2 Apply the `GlobalNonApplicabilityAlgorithm` to the rules of all other flow control policies associated with the enforcing system's meta-policy.

3 Order all flow control policies according to the meta-policy's `PolicyPriorityAlgorithm`.

4 Order the rules of each flow control policy according to the meta-policy's `Rule-PriorityAlgorithm`.

5 Apply the optional `LocalConflictResolutionAlgorithm` to all rules which are in conflict with each other and are part of the same policy. Skip this step if a policy does not define a local conflict resolution algorithm.

6 Apply the `GlobalConflictResolutionAlgorithm` to all remaining rules which are in conflict with each other.

Steps **1** and **2** resolve all application-specific conflicts. After these steps, every flow control policy only contains such rules which can be completely interpreted by and implemented on their enforcing system. Modality conflicts which can be resolved by defining different priorities of the conflicting rules are eliminated by applying steps **3** and **4**. Rules with a low priority that are in conflict with a rule of higher priority are ignored by the enforcing system. Any modality conflict which still remains after steps **3** and **4** is resolved during steps **5** and **6**. In order to achieve this, the `Global-ConflictResolutionAlgorithm`s are designed to remove all contradicting rules or their corresponding policies in the final step **6**. A flow control meta-policy must define at least one algorithm for each type. If there is more than one algorithm per type, the property `follows` from DUL's sequence pattern defines the order of their application. Evaluating the six steps above ensures that all remaining rules can completely be interpreted by the enforcing system. However, if rules or policies are removed during this process, manual intervention may be necessary to further analyze and eliminate the actual cause of the conflict.

3.3.6. Organizational Regulation and Legal Regulation Patterns

The Organizational Regulation patterns and the Legal Regulation patterns allow to associate human-readable background information with a flow control policy. The patterns are designed to integrate existing legal ontologies such as LKIF [146, 147] or CLO [123, 118] into InFO by using corresponding mapping ontologies. This flexible design allows to reuse different external ontologies with variable expressiveness in different scenarios. The Organizational Regulation defines the Code of Conduct Pattern and the Legal Regulation defines the Flow Regulation Norm Pattern and the Legislation Pattern. This section describes these three patterns. The integration of existing legal ontologies into InFO is discussed in Section 3.3.7.

The Code of Conduct Pattern depicted in Figure 3.11 allows to describe the organizational code of conduct on which a technical flow control implementation is based. A code of conduct is represented by the pattern's main concept `CodeOfConductDescription`. It is created by the party acting as the `CodeOfConductCreator` and based on at least one legal foundation such as a legal norm or a law. The legal foundation can define the boundaries of a code of conduct by stating that the code must not violate any legal norm. A code of conduct is expressed by a `CodeOfConductText` which is a subclass of `InformationObject` and describes the code in a human-readable form. The class `CodeOfConductDescription` is used in the Flow Control Policy Pattern depicted in Figure 3.9 to link a particular flow control policy to its organizational motivation. The code of conduct then holds for all flow control rules of the associated policy.

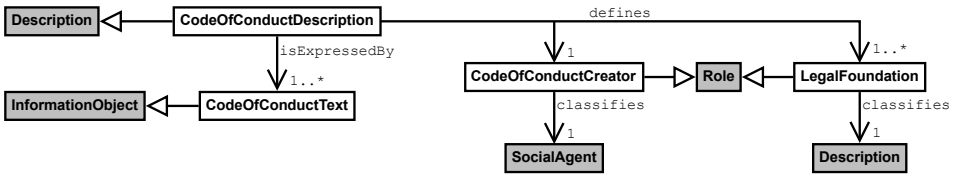


Figure 3.11.: Code of Conduct Pattern.

The Flow Regulation Norm Pattern is depicted in Figure 3.12 and models the legal state of a particular communication flow. It defines whether a technical communication flow is permitted or prohibited by using a corresponding subclass of the pattern's main concept `FlowRegulationNorm`. The pattern can be considered as a legal view on the technical Flow Control Rule Pattern described in Section 3.3.3. The Flow Regulation Norm Pattern models a particular communication flow as an event by using DUL's participation pattern as depicted in Figure 3.2e. The participants of this event are both communicating parties, the transmitted content, and the content's topic. The communicating parties are distinguished between the content provider and the content consumer. Both parties are represented by their technical communication system such as a web server or a web browser and the agent who uses that system. Possible agents include organizations which may operate a web server and natural persons which may use a web browser. In contrast to the Flow Control Rule Pattern, the Flow Regulation Norm Pattern does not specify all details of a technical communication system. Instead, the

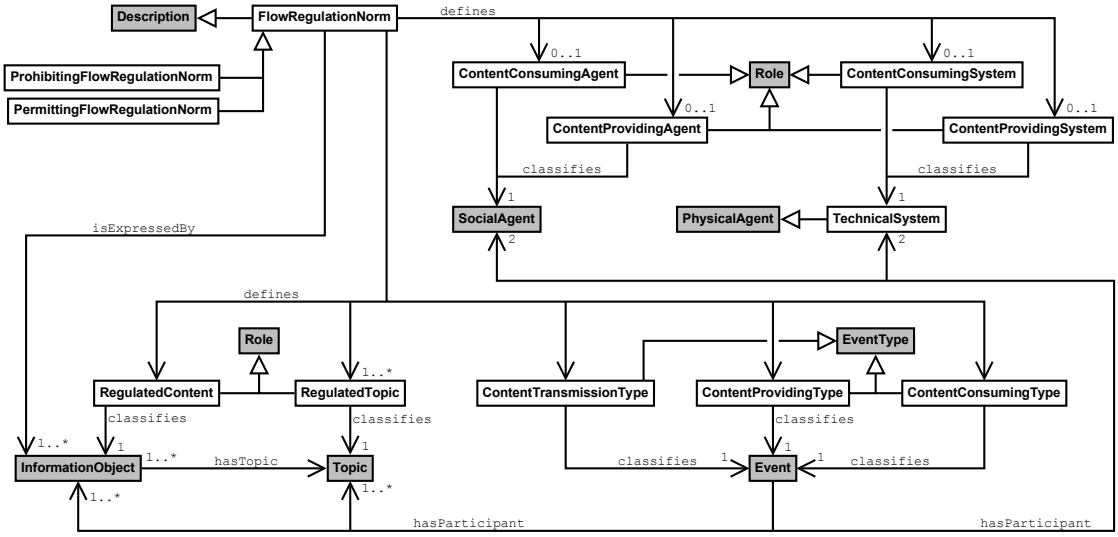


Figure 3.12.: Flow Regulation Norm Pattern.

legal view modeled by this pattern focuses on the system's type such as a router or a name server. The type of the event defines the specific aspect of the communication flow that is actually regulated. `ContentTransmissionType` states that the legal norm regulates the content's transmission between two communicating parties and affects both parties. If the event type corresponds to a `ContentProvidingType`, the legal norm mainly affects the content provider whereas a `ContentConsumingType` primarily regulates the actions of the content consumer. For example, §86 of the German Criminal Code [62] regulates the distribution of neo-Nazi propaganda and could be modeled using the event type `ContentProvidingType`. Similar to a code of conduct, a legal norm is also expressed by an `InformationObject` which models a human-readable representation of the norm. The specific relations between all entities of the Flow Regulation Norm Pattern depend on the integrated legal ontology. The general procedure of this integration is discussed in Section 3.3.7 and possible mappings are provided in Appendix B.1. A `FlowRegulationNorm` is associated with a flow control policy by using the Flow Control Policy Pattern described in Section 3.3.4. This pattern models the norm as the policy's legal foundation, thereby linking it to all of the policy's flow control rules as well.

The Legislation Pattern is depicted in Figure 3.13 and complements the Flow Regulation Norm Pattern. It models the altering process of a legal norm and provides further background information about how the current state of a legal norm was achieved. This altering process is considered as a `LegislationAct` which is performed by a `Legislator` who is responsible for the process. The Legislation Pattern has a similar design as the Flow Regulation Norm Pattern. Its main concept `LegislationNorm` is associated with all concepts relevant for passing or modifying a legal norm. The particular alteration of a legal norm is modeled as a subclass of `AlteredNorm` and may be the creation, the suspension, or the modification of a norm.

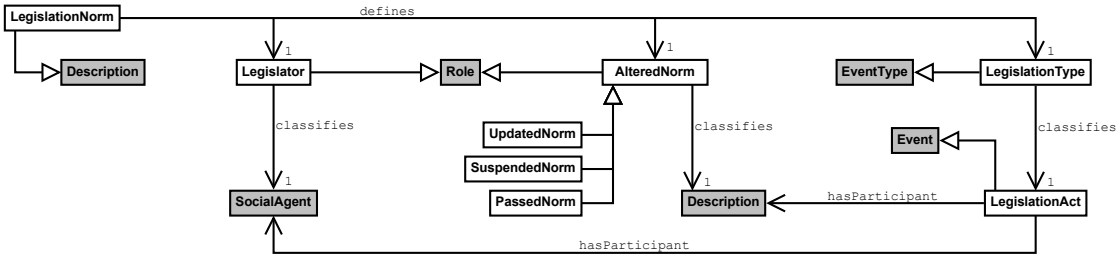


Figure 3.13.: Legislation Pattern.

3.3.7. Integration of Existing Legal Ontologies into InFO

The Organizational Regulation patterns and the Legal Regulation patterns are designed for using existing legal ontologies together with InFO by integrating them into InFO's pattern system. The main goal of this integration is the reuse of the legal ontologies without modifying or refactoring them beforehand. Statements of the integrated existing legal ontologies must be valid in both InFO and in the original legal ontology. The integration of these legal ontologies is done in four steps and based on the alignment method by Scherp et al. [265]. In the first step, the structure and design of a legal ontology is analyzed and the core concepts and properties are identified. The analysis is based on reviewing the ontology model and/or any additional documentation. In the second step, existing groups of concepts and properties are identified. Such groups may be explicitly modeled by using, e. g., ontology design patterns or only described in external documents. The third step corresponds to the actual integration and is based on creating a mapping ontology for each legal ontology. As depicted in Figure 3.4, such a mapping ontology imports the Organizational Regulation and the Legal Regulation of InFO as well as the legal ontology to be integrated. A mapping ontology does not define any new classes or properties. Instead, it only defines subclass and subproperty relationships between the concepts and properties of InFO and the legal ontology. However, in contrast to Scherp et al. [265], modifying the original ontology by, e. g., removing concepts or axioms is not intended. Therefore, the internal structure of both InFO and the integrated legal ontology remains intact. The fourth step is the validation of the mappings and can be done using an ontology reasoner. The validation ensures that the mapping is correct and does not contain any modeling errors.

Figure 3.4 shows how the legal ontologies CLO [123, 118] and LKIF [146, 147] are integrated into InFO. They can be used for both describing the organizational background and the legal background of an information flow regulation. The actual integration is done with the mapping ontologies `LKIFMapping` and `CLOMapping`. Both mapping ontologies import the Organizational Regulation patterns and the Legal Regulation patterns as well as their respective legal ontology and define additional statements for the integration. The details of the mapping ontologies are provided in Appendix B.1.

3.3.8. Integration of Existing Content Labeling Schemes into InFO

The Flow Control Rule Pattern introduced in Section 3.3.3 uses the topic pattern of the Ontopic ontology for describing the topic of the transmitted content. Although the descriptions created with this pattern are sufficient for many scenarios, it is also possible to further describe the transmitted content by importing content labeling schemes such as the RTA label, age-de.xml [264], or PICS [248] into InFO. These schemes are designed to be used by child protection software for prohibiting minors from accessing adult web content. Thus, integrating these schemes into InFO can achieve more compatibility between the InFO pattern system and other implementations for regulating access to web pages such as Net Nanny and the Jugendschutzprogramm. Figure 3.4 shows how the RTA label, age-de.xml, and PICS are integrated into InFO. The integration of the content labeling schemes is done in a similar way as the integration of the legal ontologies described in the previous section. The mapping uses the corresponding mapping ontologies RTAMapping, AgeDeXmlMapping, and PICSMapping. The details of the three mapping ontologies are provided in Appendix B.2.

3.3.9. Summary

The pattern system InFO consists of several ontology design patterns which cover specific aspects for describing the regulation of information flow on the Internet. These aspects are either of technical, of organizational, or of legal issue. The main focus of InFO is the technical regulation of Internet communication. The Organizational Regulation and the Legal Regulation are designed to be used together with existing legal ontologies.

3.4. Applications and Use Cases

This section demonstrates how the InFO policy language is applied for implementing the scenarios provided in Section 2. The first scenario covers the regulation of Internet communication and is described in Sections 3.4.1 to 3.4.4. Afterwards, Section 3.4.5 describes the second scenario for securing the access to medical data.

3.4.1. Example Policies for Regulating Internet Communication

Figure 3.14 depicts a subnetwork of the example network described in Section 2.1 including more technical details such as the addresses of the communication nodes and the URLs of the web content. These details are required for precisely defining a set of flow control policies. The policies are created using three different domain ontologies that extend InFO's Technical Regulation patterns. These domain ontologies are the Router Ontology, the Name Server Ontology, and the Application-Level Proxy Ontology. Each ontology defines specific flow control rules which are designed to be implemented on their respective enforcement systems. For example, the Router Ontology defines the classes `IPAddressBlockingRuleMethod` and `IPAddressRedirectingRuleMethod` as subclasses

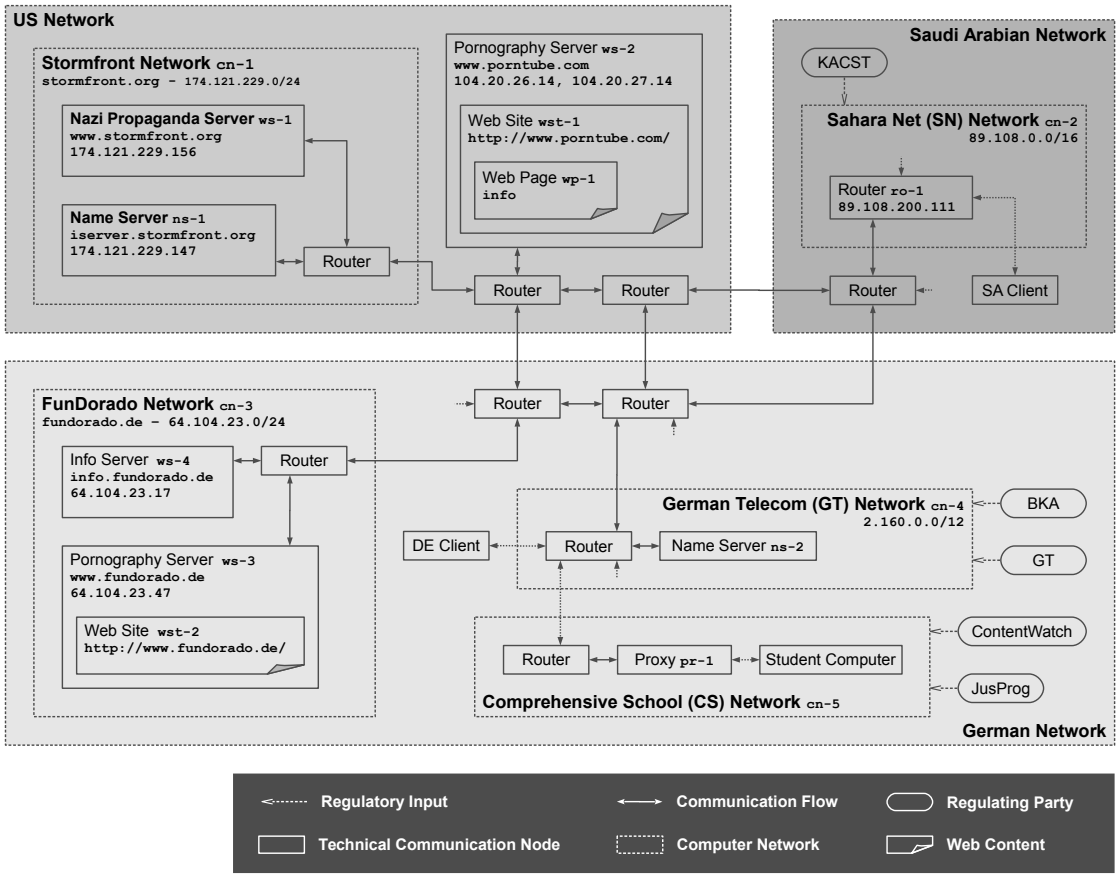


Figure 3.14.: Regulated web servers and web content of the example policies. The depicted topology is a subnetwork of Figure 2.1 extended with more technical details such as IP addresses and domain names.

of the generic classes `DenyingFlowControlRuleMethod` and `RedirectingFlowControl-RuleMethod`. The ontology also provides additional axioms which reduce the possibility of creating invalid flow control rules and flow control policies. Flow control rules based on IP addresses require at least one IP address for the sender and/or the receiver of a communication. If such a flow control rule does not contain at least one IP address, the rule cannot be enforced by a router and is therefore considered as invalid. Similarly, the Name Server Ontology defines classes and axioms for flow control rules based on domain names and the Application-Level Proxy Ontology covers classes and axioms based on URLs. Further details of the classes and axioms of the three domain ontologies are provided in Appendix A.

The following subsections present three example flow control regulations for each of the three domain ontologies. The example policies cover the regulation of the computer networks `cn-1` and `cn-3` as well as the web server `ws-2`. The network `cn-1` provides neo-Nazi material whose access shall be prohibited for all German users. The flow control

rules implementing this access regulation are based on domain names and are enforced by the name server `ns-2` of the German Telecom. The network `cn-3` located in Germany and the web server `ws-2` located in the USA provide pornographic web content. Access to this content shall be prohibited for all users in Saudi Arabia and for all students of the German comprehensive school. In Saudi Arabia, the access regulations are based on IP addresses and enforced by the router `ro-1` whereas the German comprehensive school regulates the access using its proxy server `pr-1` which operates on URLs. For illustration purpose, each flow control regulation consists of two or three `FlowControlRuleMethod`s, one `FlowControlPolicyMethod` that contains these rules, and one `FlowControlMeta-PolicyMethod`. All rules of the same policy share the same regulated topic and sender. The sender of each rule and hence the requester of the regulated content is modeled as a whole computer network rather than a single computer.

In general, a communication between a content consumer and a content provider can be regulated in two different ways. The first option is to regulate the content consumer's request before it is sent to the content provider. In this case, the server acting as the content provider does not receive any message from the client which acts as the content consumer. The second option is to regulate the content provider's response after it has processed the content consumer's request. Implementing this option allows the server to receive messages from the client but prohibits the transmission of messages in the opposite direction. InFO generally supports both types of regulation by defining the sender and receiver of a communication accordingly. Choosing a particular type of regulation may depend on technical, on organizational and/or on legal factors and must be decided before creating particular regulation policies. For example, the first option results in a faster regulation as an enforcing system can immediately regulate a request without having to wait for a corresponding response. This also reduces the amount of transmitted data and may result in a faster Internet connection of the enforcing node. On the other hand, the legal foundation of a regulation may allow requesting particular content but not the transmission of the content itself. In this case, the second option might be used for implementing the regulation. In addition, there are also technical constraints when choosing between the two types of regulation. Although routers and proxy servers can regulate the flow of communication in both directions, name servers can only regulate requests from a client system. This is due to the use of the domain name system when initiating an Internet communication [212]. If a client wants to contact a server, it first maps the domain name of the server to its IP address. The IP addresses of the client and the server are then included in the messages which are exchanged between the two systems. When sending a response, the server already has the IP address of the client and does not need to contact a name server. Thus, prohibiting the communication between two parties is only possible by implementing the regulation on the client's name server. In the following, all example rules are designed in such a way that the initial request of a client system is regulated by the enforcing system.

For reasons of brevity, the following depictions of flow control rules, policies, and meta-policies only show their most important aspects although the actual regulation is still complete. All three meta-policies define a global conflict resolution algorithm and a global non-applicability algorithm as these algorithms are mandatory for the

conflict resolution process. In contrast, the three flow control policies do neither define a local conflict resolution algorithm nor a local non-applicability algorithm. Local conflict resolution algorithms are only evaluated if a policy contains two or more contradicting rules. Since the flow control rules used in the following examples do not provoke any conflicts, the algorithm is omitted for simplicity reasons. On the other hand, local non-applicability algorithms are only evaluated if an enforcing system cannot implement a particular flow control rule. The flow control rules used in the examples are basic blocking and allowing rules. The following subsections provide three examples of flow control rules, flow control policies, and flow control meta-policies to be enforced on a router, a name server, and a proxy server. A detailed workflow of creating and distributing these regulations is provided in Section 4.7.1 as part of the graph signing framework Siggi.

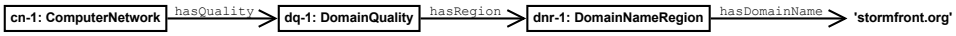
3.4.2. Applying the Name Server Ontology

In the first example regulation, the German Telecom regulates the access to the Stormfront network, which provides an online platform hosting neo-Nazi material [95, 319]. The regulation prohibits the clients of the German Telecom to access any neo-Nazi material available in the network. The regulation is implemented using the Name Server Ontology and enforced by a name server of the German Telecom. The Stormfront network is a real-world example of regulating Internet communication and was the target of several regulations in the past. The network is still regulated in some way, e. g. the French[11] and the German[12] versions of the Google search engine exclude the website from their search results [196]. A detailed discussion of the regulation of Stormfront is provided in [95, 319]. As depicted in Figure 3.14, the Stormfront network is identified as `cn-1` and its domain name is `stormfront.org`. The network contains a name server represented by the individual `ns-1` and a web server represented by the individual `ws-1`. While `ns-1` is a regular name server managing the domain names of the domain `stormfront.org`, `ws-1` corresponds to the web server providing the Stormfront web forum. The domain name of the web server is `www.stormfront.org` and the name server can be accessed by its domain name `iserver.stormfront.org`. The example policy for regulating the Stormfront network `cn-1` only blocks access to those parts of the network that can be directly associated with neo-Nazi material. At the same time, the regulations allow access to other network nodes such as the name server `ns-1`. The name server only provides a mapping between domain names and IP addresses and does not host any web content of neo-Nazi material itself.

Figure 3.15 depicts the general definitions used for the example flow control rules shown in Figure 3.16. Figure 3.15a shows an ontological representation of the Stormfront network `cn-1` and Figure 3.15b depicts the ontological representation of the network `cn-4` of the German Telecom. Both representations use DUL's qualities and quality region pattern to associate a computer network with its network address and its domain name. The computer network of the German Telecom is identified as `cn-4` and its network address is `2.160.0.0/12`. The postfix `/12` of the network address denotes the

[11] `http://www.google.fr`, last accessed: 01/21/16
[12] `http://www.google.de`, last accessed: 01/21/16

CIDR notation [114] of the network's subnet mask. `12` corresponds to the subnet mask `255.240.0.0`. Figure 3.15c shows how the topic pattern of the Ontopic ontology is used for modeling a topic representing neo-Nazi propaganda. The topic is identified as `nnpt-1` and consists of one multiword and one regular word.



(a) Representation of the Stormfront network `cn-1`. DUL's qualities and quality region pattern depicted in Figure 3.2c is used to model the network's domain name.

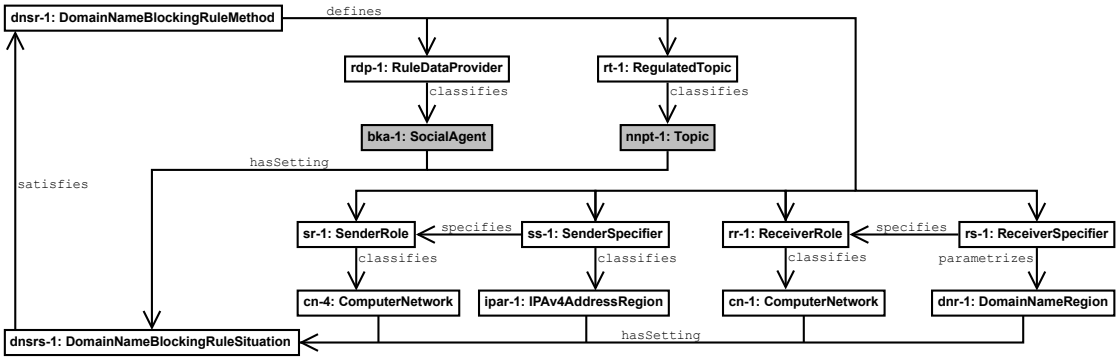
(b) Representation of the computer network `cn-4` of the German Telecom. The network is associated with its IP address by using DUL's qualities and quality region pattern.

(c) Representation of the topic `nnpt-1` describing neo-Nazi material. The topic is modeled using the topic pattern of the Ontopic ontology depicted in Figure 3.3.

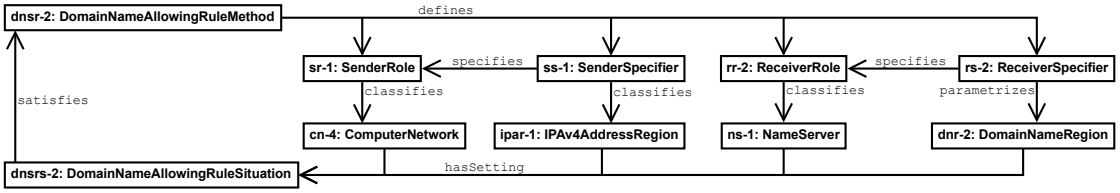
Figure 3.15.: General definitions used in the example regulation of the Name Server Ontology.

The example regulation consists of two flow control rules, one flow control policy, and one flow control meta-policy. Both flow control rules cover the same sender, the same regulated topic, and the same rule data provider. The sender of each rule and thus the requester of the regulated content corresponds to the computer network `cn-4` of the German Telecom. The topic of the regulated content is `nnpt-1` and corresponds to neo-Nazi material. The rule data provider of each rule is `bka-1` and corresponds to the BKA which is in charge of creating flow control rules and sending them to the German Telecom. The first rule `dnsr-1` depicted in Figure 3.16a states that any German client connected to the network `cn-4` shall be prevented from establishing a connection to the Stormfront network `cn-1`. The intention of this rule is to prevent users of the German Telecom from accessing neo-Nazi material hosted within the Stormfront network. However, the network also includes servers which do not provide neo-Nazi material such as the name server `ns-1`. Thus, the second flow control rule `dnsr-2` depicted in Figure 3.16b allows to access this name server. `dnsr-2` shares the same rule data provider and sender network as rule `dnsr-1`. For reasons of brevity, not all of these details are depicted in the figure.

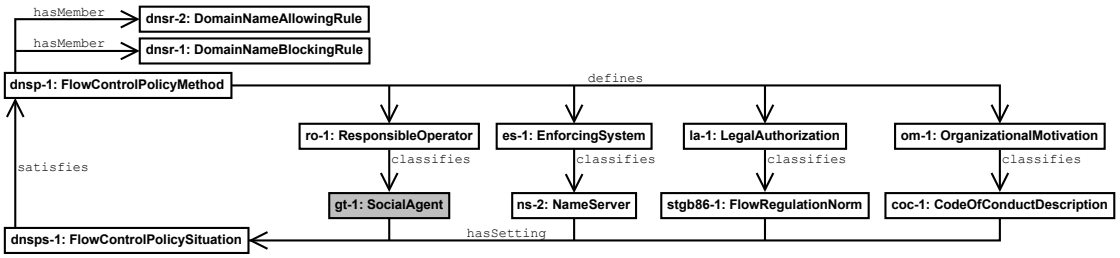
The flow control policy `dnsp-1` of the flow control rules `dnsr-1` and `dnsr-2` is depicted in Figure 3.16c. It associates both rules with their enforcing system and their responsible operator. The responsible operator `gt-1` represents the German Telecom and the enforcing system `ns-2` corresponds to the name server depicted in Figure 3.14. For reasons of brevity, the name server is not further specified. However, it is also possible to further describe the name server by using DUL's qualities and quality region pattern



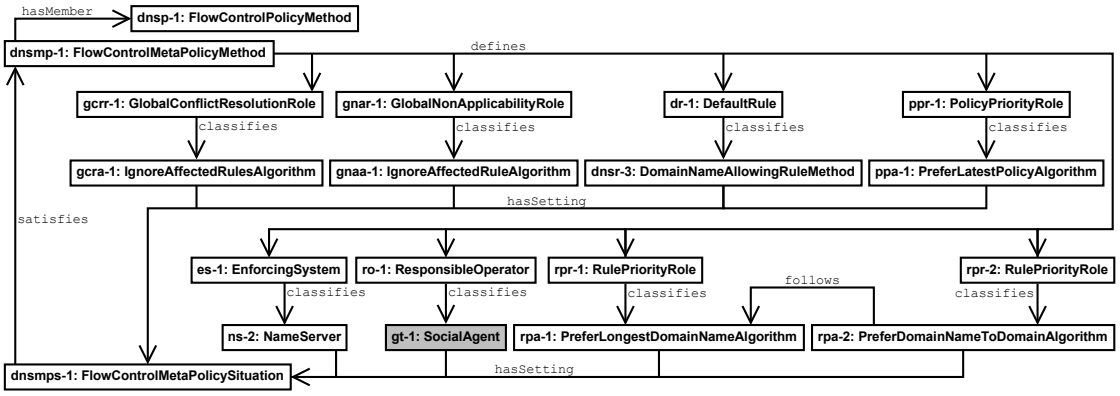
(a) First example flow control rule of the Name Server Ontology



(b) Second example flow control rule of the Name Server Ontology



(c) Example flow control policy of the Name Server Ontology



(d) Example flow control meta-policy of the Name Server Ontology

Figure 3.16.: Example regulating using the Name Server Ontology.

for, e. g., associating it with its IP address. The flow control policy `dnsp-1` also links the two rules `dnsr-1` and `dnsr-2` to their respective legal authorization and organizational motivation. In the case of the Stormfront network, the legal authorization is §86 of the German Criminal Code [62] which is identified as `stgb86-1`. The code of conduct of the German Telecom [91] is used as an organizational motivation and represented as `coc-1`.

The flow control meta-policy `dnsmp-1` of the of the name server `ns-2` is depicted in Figure 3.16d. As depicted, it shares the same enforcing enforcing system and responsible operator as the flow control policy `dnsp-1`. Furthermore, it defines a default rule and several conflict resolution algorithms. The default rule is identified as `dnsr-3` and states that any communication which is not explicitly covered by another rule is to be allowed. This type of regulation corresponds to a blacklisting approach in which all communication is to be allowed as long as it is not explicitly forbidden by a particular rule. As depicted in Figure 3.16c, the flow control policy `dnsp-1` does not define any local conflict resolution algorithm or local non-applicability algorithm. Therefore, all non-applicable rules and conflicts between contradicting rules are handled by the meta-policy `dnsmp-2`. The global non-applicability algorithm `gnaa-1` states that all rules which cannot be enforced by the name server `ns-2` are to be ignored. As both rules `dnsr-1` and `dnsr-2` only use standard concepts of the InFO policy language, they are not affected by this algorithm. The policy priority algorithm `ppa-1` states that newer policies have higher priority than older policies. Since the example regulation only uses one policy, this algorithm does not affect the flow control policy `dnsp-1`. The meta-policy defines two different rule priority algorithms which are `rpa-1` and `rpa-2`. The algorithms decide which of the two flow control rules `dnsr-1` and `dnsr-2` are used for regulating access to the name server `ns-1`. Both rules cover the domain name `dnr-1` of the name server and can therefore generally be applied for regulating its access. DUL's sequence pattern is used to define that the algorithm `rpa-1` has a higher priority than `rpa-2` and must therefore be applied first. The algorithm `rpa-1` states that longer domain names shall be preferred to shorter ones. In this case, the domain name `iserver.stormfront.org` of the name server `ns-1` is longer that the domain name `stormfront.org` of the whole computer network `cn-1`. Thus, the enforcing system `ns-2` applies the rule `dnsr-2` to the name server `ns-1` and the rule `dnsr-1` to all other servers of the same domain. This results in the refusal of any communication attempts to any server in the network `stormfront.org` except for the name server with the domain name `iserver.stormfront.org`. Finally, the global conflict resolution algorithm `gcra-1` states that all conflicting rules which are still in conflict with each other after having applied all other algorithms are to be ignored by the enforcing system. The conflict between the two rules `dnsr-1` and `dnsr-2` is resolved after having applied the rule priority algorithm `rpa-1`. Thus, these rules are not affected by the global conflict resolution algorithm `gcra-1`.

3.4.3. Applying the Router Ontology

In the second example regulation, the Saudi Arabian access provider Sahara Net prohibits its users from accessing pornographic web content. The regulation to this web content is enforced by a router of Sahara Net and implemented using the Router On-

tology. The regulated web content is provided by a web server located in the USA and by the FunDorado network located in Germany. The US web server is identified as `ws-2` and has the two IP addresses `104.20.26.14` and `104.20.27.14`. The FunDorado network is identified as `cn-3` and has the network address `64.104.23.0/24`. It contains the two web servers `ws-3` and `ws-4`. Similar to the web server `ws-2`, the web server `ws-3` also provides pornographic content. On the other hand, the web server `ws-4` hosts the website of FunDorado GmbH, the company managing the FunDorado network. The company's web site does not contain any pornographic content and only provides information about the company and its services. Thus, access to the web server `ws-4` with the IP address `64.104.23.17` shall not be blocked. Figure 3.17 shows how the IP addresses of the web server `ws-2` are defined. The network address of the FunDorado network and the IP addresses of its web servers are defined similarly and are not included for reasons of brevity.

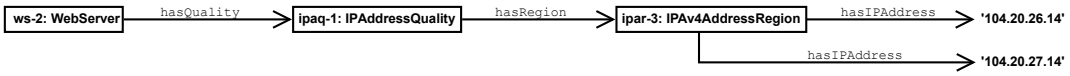
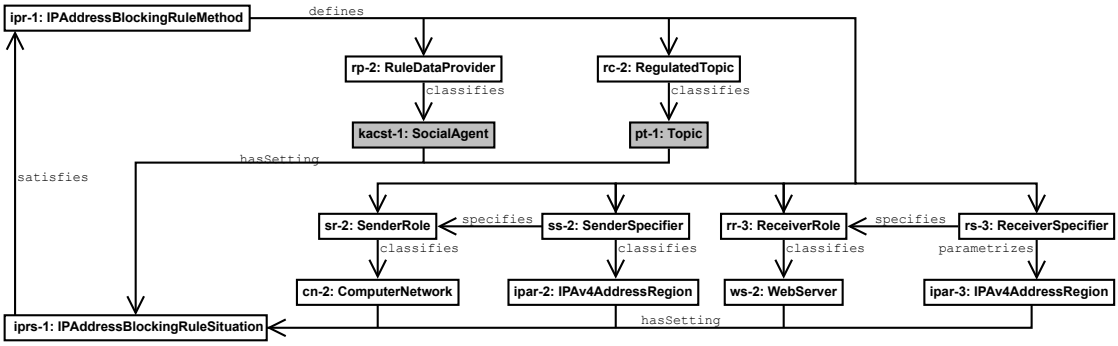


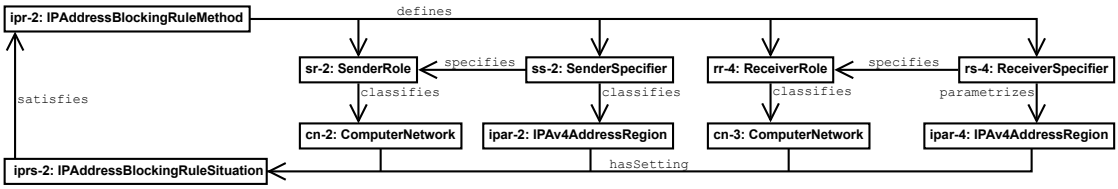
Figure 3.17.: Representation of the US web server `ws-2` and its two IP addresses.

The example regulation is depicted in Figure 3.18 and consists of three flow control rules, one flow control policy, and one flow control meta-policy. All flow control rules cover the same sender, the same rule data provider, and the same regulated topic. The sender of each rule corresponds to the network `cn-2` of Sahara Net. All rules are created by the KACST which is identified as `kacst-1`. The rule data provider is responsible for regulating all Internet communication in Saudi Arabia and sends the created rules to Sahara Net. The topic of the regulated web content is identified as `pt-1` and represents pornography. The first flow control rule `ipr-1` is depicted in Figure 3.18a. It states that any client system of the network `cn-2` shall be prevented from establishing a connection to the web server `ws-2`. Similarly, the second flow control rule `ipr-1` depicted in Figure 3.18b blocks the access for all users of the computer network `cn-2` to the FunDorado network `cn-3`. This network mainly contains web servers such as `ws-3` which provide pornographic web content which is to be blocked. However, the network also covers the web server `ws-4` which does not provide such content. Thus, the flow control rule `ipr-3` depicted in Figure 3.18c allows the access to this web server. The flow control rules `ipr-2` and `ipr-3` share the same rule data provider and regulated topic as rule `ipr-1`. For reasons of brevity, this is not shown in Figures 3.18b and 3.18c.

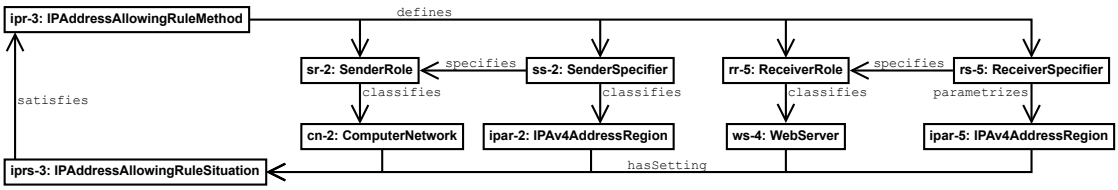
The flow control policy `ipp-1` depicted in Figure 3.18d associates the flow control rules `ipr-1`, `ipr-2`, and `ipr-3` with their enforcing system and their responsible operator as well as their legal authorization. The enforcing system is the Router `ro-1`. It is operated by Sahara Net which is identified as `sn-1`. The legal authorization of the flow control rules is §6 of the Saudi Arabian Anti-Cyber Crime Law [179] and represented as `accl6-1`. An organizational motivation for the regulations is not provided in the policy. Since the flow control policy `ipp-1` does not define any local conflict resolution algorithm, all conflicts between conflicting rules are resolved by the policy's meta-policy `ipmp-1`



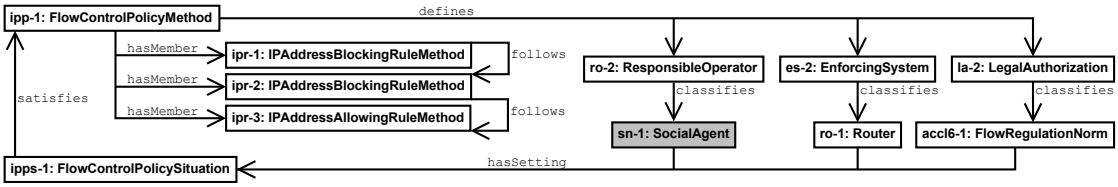
(a) First example flow control rule of the Router Ontology



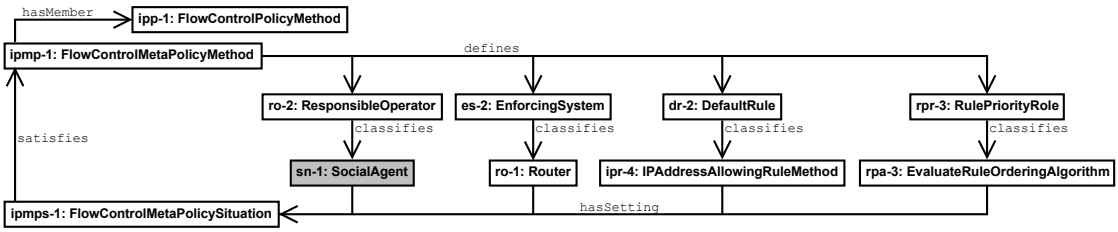
(b) Second example flow control rule of the Router Ontology



(c) Third example flow control rule of the Router Ontology



(d) Example flow control policy of the Router Ontology



(e) Example flow control meta-policy of the Router Ontology

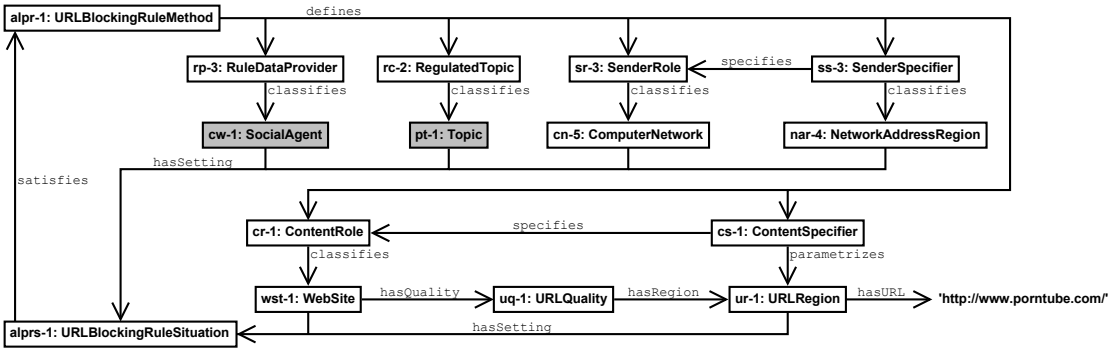
Figure 3.18.: Example usage of the Router Ontology.

depicted in Figure 3.18e. The meta-policy contains the flow control policy `ipp-1` and shares the same responsible operator and enforcing system. Its default rule `ipr-4` states that any communication is allowed as long as it is not explicitly forbidden by another rule. For reasons of brevity, Figure 3.18e does not depict all conflict resolution algorithms of the meta-policy. Instead, it only shows a rule priority algorithm for resolving the conflict between the two flow control rules `ipr-2` and `ipr-3`. Both rules cover the same IP address region `ipar-5` which is associated with the web server `ws-4`. Thus, they can in general be applied to this server. In order to decide which of the two rules are to be used, the enforcing router `ro-1` must apply the rule priority algorithm `rpa-3`. This algorithm requires an explicit ordering of flow control rules with the property `follows`. As depicted in Figure 3.18d, the rule `ipr-3` has a higher priority than `ipr-2` in order to allow access to the web server `ws-4`. Thus, the router `ro-1` applies the rule `ipr-3` to the web server `ws-4` and the rule `ipr-2` to all other servers of the computer network `cn-2`. This results in the refusal of any communication attempts to any server in the network `64.104.23.0/24` except for the web server with the IP address `64.104.23.17`. As the flow control rule `ipr-1` is not in conflict with the other flow control rules, it is not affected by the meta-policy's rule priority algorithm.

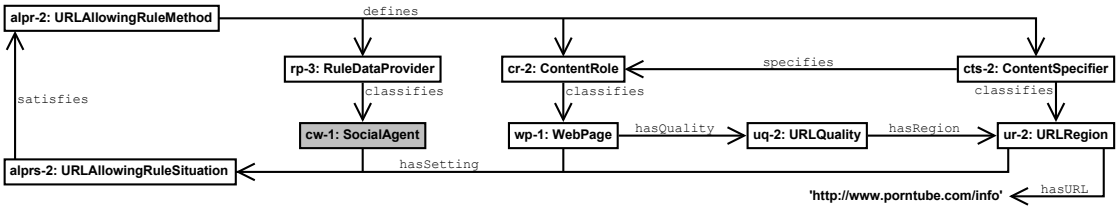
3.4.4. Applying the Application-Level Proxy Ontology

In the third example regulation, the German comprehensive school `cs-1` prohibits its students from accessing pornographic web content by using the Application-Level Proxy Ontology. The regulation is enforced by the school's proxy server `pr-1` which serves as a gateway for all student computers. The regulated web content corresponds to the websites `wst-1` and `wst-2` which are hosted by the web servers `ws-2` and `ws-3`, respectively. The example regulation consists of three flow control rules, one flow control policy, and one flow control meta-policy as depicted in Figure 3.19. The first flow control rule `alpr-1` is depicted in Figure 3.19a. It states that any user of the school's network `cn-5` shall be prevented from accessing the website `wst-1`. The rule is provided by the US company ContentWatch which is represented as `cw-1`. The regulated topic corresponds to pornography and is identified as `pt-1`. The website `wst-1` consists of several web pages including the web page `wp-1`. Although most of these web pages provide pornographic content, the web page `wp-1` does not. Instead, it only provides textual information about the website such as its terms of services and its privacy policy. Thus, the second flow control rule `alpr-2` depicted in Figure 3.19b allows the student computers to access the web page `wp-1`. The third flow control rule `alpr-3` depicted in Figure 3.19c blocks the access to the FunDorado website `wst-2`. The rule is created by JusProg which is a registered society located in Germany and identified as `jp-1`. General information about the website `wst-2` such as its terms of services is provided by a separate website which is hosted by the web server `ws-4`. Access to this website is not covered by any flow control rule and therefore not regulated in any way.

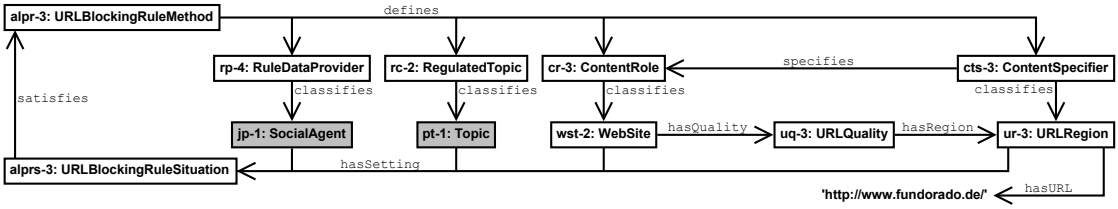
The flow control policy `alpp-1` combines the three rules `alpr-1`, `alpr-2`, and `alpr-3` and is depicted in Figure 3.19d. The policy states that the rules are enforced by the proxy server `pr-1` which is operated by the school itself. Although the rules are created



(a) First example flow control rule of the Application-Level Proxy Ontology



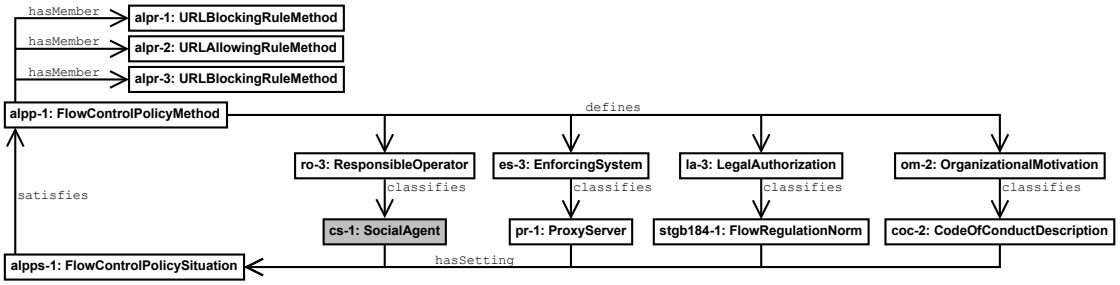
(b) Second example flow control rule of the Application-Level Proxy Ontology



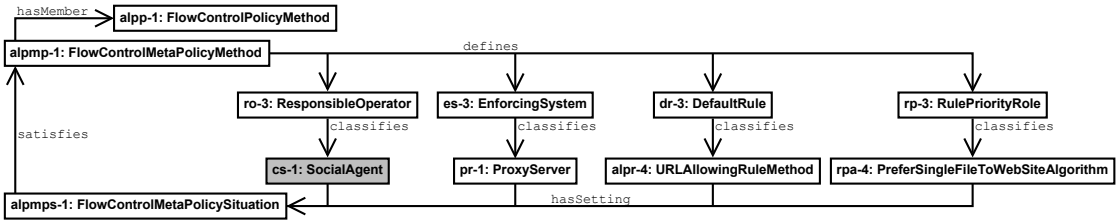
(c) Third example flow control rule of the Application-Level Proxy Ontology

Figure 3.19.: Example usage of the Application-Level Proxy Ontology.

by different rule data providers which even operate in different countries, they cover the same topic. Access to this topic is to be regulated according to §184 of the German Criminal Code [61]. Thus, the flow control policy defines this article as a legal foundation for all flow control rules. As an organizational motivation, the policy uses the school's code of conduct `coc-2`. Again, `alpp-1` does neither define a local conflict resolution algorithm nor a local non-applicability algorithm itself and relies on its corresponding flow control meta-policy `alpmp-1` for resolving conflicts between contradicting rules and handling non-applicable rules. The flow control meta-policy `alpmp-1` is depicted in Figure 3.16d and shares the same enforcing system and responsible operator as the policy `alpp-1`. It defines the rule priority algorithm `rpa-4` which is an instance of the class `PreferWebPageToWebSiteAlgorithm`. This algorithm states that rules associated with single web pages shall be preferred to rules with whole websites. The website `wst-1` covered by the rule `alpr-1` contains the web page `wp-1`. In applying the algorithm `rpa-4`, the proxy server `pr-1` uses the rule `alpr-2` for allowing access to the web page `wp-1`



(d) Example flow control policy of the Application-Level Proxy Ontology



(e) Example flow control meta-policy of the Application-Level Proxy Ontology

Figure 3.19.: Example usage of the Application-Level Proxy Ontology. *Continued from previous page.*

while blocking access to all other web pages of the website `wst-1`. The flow control rule `alpr-3` is not affected by this algorithm since it covers a different website than the other two rules.

3.4.5. Example Policies for Securing the Exchange of Medical Data

The scenario for securing medical data records introduced in Section 2.2 covers a medical case which involves the transmission of such records between different care delivery organizations (CDOs). Ensuring the confidentiality of medical records is required by law as these records contain sensitive personal information. Thus, the transmission of medical data records between different CDOs must be protected as well. In general, the secure transmission of data can be achieved by using secure communication protocols

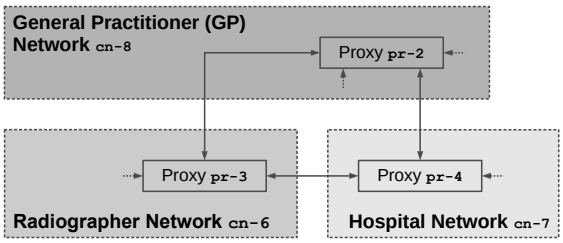
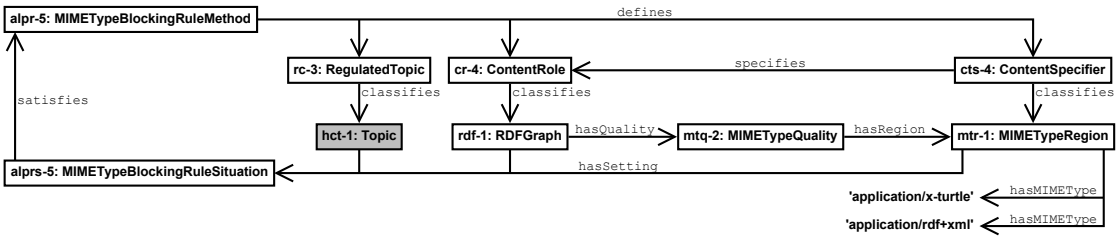
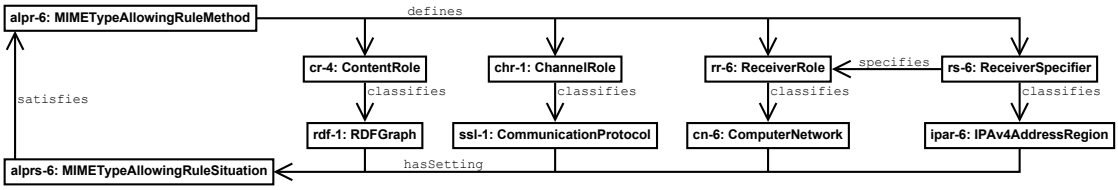


Figure 3.20.: Example computer network connecting three different CDOs. The proxy server of each network serves as a gateway to the other networks.

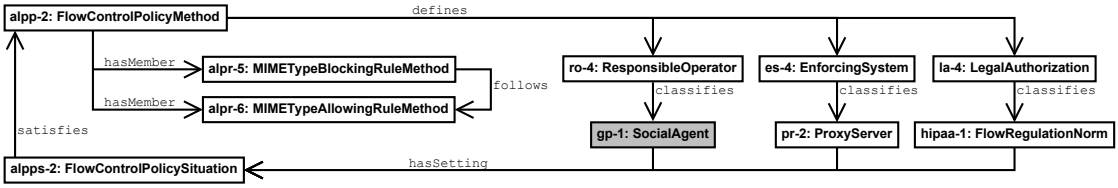
such as SSL [113]. InFO policies are applied to ensure that medical records can only be transmitted between CDOs via a secure SSL connection and that any other transmission of the records is prohibited. Figure 3.20 shows a simplified depiction of three computer networks which correspond to the three CDOs of the scenario. These CDOs are the general practitioner (GP), the radiographer, and the hospital. Each computer network includes a proxy server which serves as a gateway to the other networks and regulates the transmission of all data. For example, the computer network `cn-8` contains the proxy server `pr-2` which is operated by the GP. The server implements the flow control regulation depicted in Figure 3.21 which consists of two flow control rules, one flow control policy, and one flow control meta policy. For reasons of brevity, the figure only



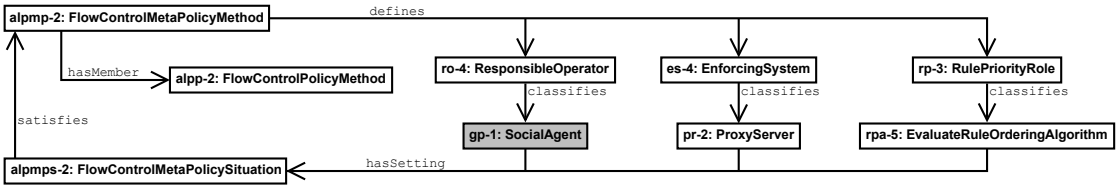
(a) Flow control rule for blocking the transmission of all RDF graphs.



(b) Flow control rule for allowing the transmission of RDF graphs via an SSL connection to the computer network `cn-6`.



(c) Flow control policy containing two flow control rules.



(d) Flow control meta policy.

Figure 3.21.: Regulating the transmission of medical data records.

depicts the most important parts of the regulation. The complete regulation is similar to the example of the application-level proxy ontology presented in Section 3.4.4. The first flow control rule `alpr-5` is shown in Figure 3.21a and blocks the transmission of all RDF graphs which are associated with the topic `hct-1`. This topic is used for classifying all health care related data and is modeled using the topic pattern of the Ontopic ontology as depicted in Figure 3.3. Instead of regulating the transmission of a particular RDF graph by using a URI, the rule uses two different MIME types in order to cover all RDF graphs. Multipurpose Internet Mail Extensions (MIME) [112] support the transmission of arbitrary data including binary data over the Internet. The format of the transmitted data is defined by its MIME type which consists of a type and a subtype. The flow control rule `alpr-5` uses the MIME types `application/rdf+xml` and `application/x-turtle` to identify RDF graphs which are encoded with the formats RDF/XML [26] or Turtle [27], respectively. The second flow control rule `alpr-6` is shown in Figure 3.21b and allows the transmission of RDF graphs containing medical data to the computer network `cn-6` as long as they are transmitted via an SSL connection. As depicted in Figure 3.20, the computer network `cn-6` corresponds to the network of the radiographer. Thus, the rule `alpr-6` allows the GP to securely transmit medical data records to the radiographer. If transmitting such records to the hospital's computer network `cn-7` shall be possible as well, an additional flow control rule must be created.

The flow control policy `alpp-2` depicted in Figure 3.21c contains the two rules `alpr-5` and `alpr-6` and associates them with their responsible operator, enforcing system, and legal authorization. The legal authorization for the regulation is HIPAA [297] which defines security requirements for transmitting medical records in the USA. The flow control meta policy `alpmp-2` is depicted in Figure 3.21d and defines `EvaluateRuleOrderingAlgorithm` as a rule priority algorithm. This algorithm creates an explicit order of the flow control rules `alpr-5` and `alpr-5` by evaluating the property `follows` as used in the flow control policy `alpp-2`. The algorithm ensures that the proxy server `pr-2` applies the rule `alpr-6` before the rule `alpr-5`. In doing so, the GP can only send medical data records encoded as RDF graphs via a secure SSL connection to the radiographer while all other transmissions of such graphs are blocked. If the other CDOs depicted in Figure 3.20 shall be able to exchange medical data records as well, additional regulations must be provided. These regulations are created similar to the regulation depicted in Figure 3.21 and are enforced by the proxy servers `pr-3` and `pr-4`.

3.5. Prototypical Implementation of the InFO Pattern System

The pattern system InFO and its three domain-specific extensions for routers, proxy servers, and name servers have been implemented on three prototypical enforcing systems. All systems share the same basic design which is shown in Figure 3.22. Each system consists of three different parts which are the *preparation module*, the *rule storage*, and the *regulation module*. The preparation module is used by the enforcing system's operator to import new InFO policies, which are provided as RDF data. The preparation module resolves any existing conflicts in the policies and transforms the remaining

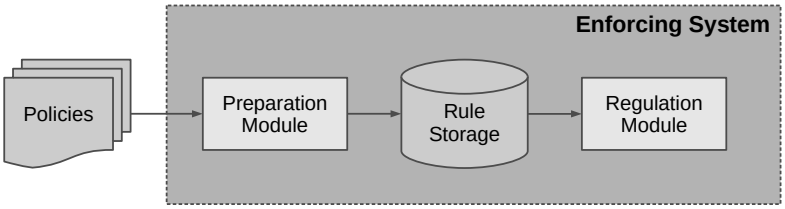


Figure 3.22.: Basic architecture of the prototypical implementations of the three enforcing systems. The arrows indicate the direction of the flow of regulation data.

policy rules to a simpler data structure. This data structure is stored in the rule storage and can be directly interpreted by the enforcing system. The regulation module performs the actual regulation by applying the transformed rules of the rule storage. The module interacts with the users that are affected by the regulation. All three enforcing systems are based on a common preparation module which is implemented in Java and uses the Jena triple store[13]. In contrast, the implementation details of the rule storage and the regulation module depend on the corresponding enforcing system. The following subsections describe the prototypical implementations of the three enforcing systems in more detail.

3.5.1. Example Name Server Implementation

The Name Server Ontology has been implemented as a prototypical, modified name server. The preparation module of this name server transforms the InFO policies into a set of DNS resource records [202]. Resource records are the data format of the domain name system and store information about domain names and IP addresses. A resource record contains a domain name, a type, and a value. The value of a resource record depends on the record's type and can be, e. g., an IP address, an alternative domain name, or a textual description. Resource records are stored in zone files which are basically collections of resource records of the same domain. In the prototypical name server, these zone files correspond to the server's rule storage and are directly used by the regulation module. The regulation module operates as a name server and answers IP address requests from users. Such requests contain a domain name and ask for the corresponding IP address. If the domain name is not regulated, the regulation module returns the correct IP address. Otherwise, the result depends on the type of regulation and can contain a wrong IP address or no IP address at all. The implementation of the regulation module is based on the Java name server EagleDNS[14] and described in more detail in [204].

[13] `http://jena.apache.org`, last accessed: 01/21/16
[14] `http://www.unlogic.se/projects/eagledns`, last accessed: 01/21/16

An example IP address request for the domain `stormfront.org` using the Unix tool dig[15] looks like: `dig +tcp stormfront.org`. This request asks the name server to retrieve the IP address for the given domain name. As the example flow control rule `dnsr-1` depicted in Figure 3.16a states, access to this domain name is to be blocked by the name server `ns-2` of the German Telecom. The name server's response is depicted in Listing 3.1. As shown in line 5, the server `REFUSED` answering the request and did not return any IP address. Additional background information about the flow control rule is returned as several `TXT` records which are shown in lines 10 to 14. Resource records of type `TXT` are generally used for associating a domain name with textual descriptions [202]. The prototypical name server uses `TXT` records to store human-readable background information about a regulation. It returns this information when a user requests the IP address of a regulated domain name. In order to better describe the content of a particular `TXT` record, the name server inserts two additional prefixes into its value. The first prefix shown in line 10 has the value `ID` which indicates that the `TXT` record contains the URI of the used flow control rule. The second prefix of this record has the value `0` and corresponds to a local identifier of this URI. This identifier is used for grouping all `TXT` records which are associated with the same rule. Since the `TXT` records shown in lines 10 to 14 all share the same local identifier, they are all based on the same flow control rule. If there is more than one rule which covers the same domain name, this number is used for distinguishing between these rules. The name server's response also contains further information about the flow control rule including its organizational motivation (line 11), its legal authorization (line 12), its rule data provider (line 13), and its regulated topic (line 14). While the topic is directly embedded into the name server's response, further information about the other regulation details can be obtained by dereferencing the URI provided in the corresponding `TXT` records.

```
1  ; <<>> DiG 9.10.1-P1 <<>> +tcp stormfront.org
2  ; (1 server found)
3  ;; global options: +cmd
4  ;; Got answer:
5  ;; ->>HEADER<<- opcode: QUERY, status: REFUSED, id: 44811
6  ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 9
7  ;; QUESTION SECTION:
8  ;stormfront.org. IN A
9  ;; ADDITIONAL SECTION:
10 stormfront.org. 3600 IN TXT "ID:0:http://..uni-koblenz.de/../dnsPolicy01.owl#dnsr-1"
11 stormfront.org. 3600 IN TXT "PH:0:http://..uni-koblenz.de/../TelekomCoC.owl#coc-1"
12 stormfront.org. 3600 IN TXT "LW:0:http://..uni-koblenz.de/../StGB.owl#stgb86-1"
13 stormfront.org. 3600 IN TXT "DP:0:http://..uni-koblenz.de/../dnsPolicy01.owl#bka-1"
14 stormfront.org. 3600 IN TXT "TO:0:neo-Nazi propaganda"
```

Listing 3.1: Example blocking result of a name server.

[15] `http://www.isc.org/software/bind`, last accessed: 01/21/16.

3.5.2. Example Router Implementation

The Router Ontology has been implemented as a set of routers, which are configured via a dedicated administration node. This node serves as the preparation module which transforms all InFO policies into a configuration script for the Linux firewall software iptables[16]. The administration node sends the configuration script to all connected routers via an encrypted SSL connection [113]. The iptables software runs on all routers. After having received the configuration script, each router applies the script in order to update the current configuration of its local iptables installation. The iptables software consists of a packet filter and several tables which store the active firewall rules. The tables correspond to the rule storage and the packet filter is directly used as the regulation module. Thus, an additional implementation of the regulation module is not required. If a user wants to access a prohibited IP address, she receives an Internet Control Message Protocol (ICMP) [236] message that informs about the regulation. ICMP is specifically designed for exchanging information messages and error messages between IP-based communication nodes. The ICMP message is encapsulated in an IP message and send back to the original requester. The sender of this message is the router implementing the flow control. All routers and their administration node are implemented in Java and described in more detail in [311].

Listing 3.2 shows an ICMP message after a user with the IP address `89.108.23.155` has tried to access the server with the IP address `104.20.26.14`. The message was captured with the packet analyzing software Wireshark[17] and slightly modified for illustration. The user is connected to the Internet via the Saudi Arabian access provider Sahara Net. The server is located in the USA and provides pornographic content. According to the example flow control rule `ipr-1` shown in Figure 3.18a, access to this server is to be blocked for all users of Sahara Net. As depicted in the example network in Figure 3.14, the users of Sahara Net communicate with the Internet through the gateway router `ro-1` with the IP address `89.108.200.111`. This router acts as the enforcing system which implements the example regulation shown in Figure 3.18. Lines 1 to 5 of Listing 3.2 show the header of the IP packet which encapsulates the ICMP message. Line 2 states that the sender of the ICMP message was the enforcing router and line 3 indicates that the receiver was the user of Sahara Net. Lines 6 to 18 cover the actual ICMP message. The meaning of an ICMP message is generally defined by its type and code. The type defines the category of the message and the code further specifies the particular reason for sending the message. Line 7 states that the server with the IP address `104.20.26.14` was unreachable due to administrative filtering as explained in line 8. The ICMP message also contains the header of the original request which is shown in lines 9 to 15. The request consists of the IP header and the TCP header. As depicted in the example network in Figure 3.14, the IP address `104.20.26.14` corresponds to the server hosting the website `http://porntube.com/`. The lines 12 and 15 further describe this server as a web server since such a server typically uses the port number `80` and the transport layer protocol TCP. The data section

[16] `http://www.netfilter.org/projects/iptables/`, last accessed: 01/21/16
[17] `http://www.wireshark.org/`, last accessed: 01/21/16

of the ICMP message depicted in lines 16 to 18 is used to provide further background information about the regulation. Line 17 shows an encoded version of the hyperlink `http://icp.it-risk.iwvi.uni-koblenz.de/policies/ipPolicy01.owl#ipr-1`. This hyperlink refers to the regulation details including their legal authorization and organizational motivation.

```
1  Internet Protocol Version 4
2  |--Source: 89.108.200.111 (89.108.200.111)
3  |--Destination: 89.108.23.155 (89.108.23.155)
4  |--Protocol: ICMP (1)
5  |--Options: (28 bytes)
6  +--Internet Control Message Protocol
7     |--Type: 3 (Destination unreachable)
8     |--Code: 13 (Communication administratively filtered)
9     +--Internet Protocol Version 4
10       |--Source: 89.108.23.155 (89.108.23.155)
11       |--Destination: 104.20.26.14 (104.20.26.14)
12       |--Protocol: TCP (6)
13       |--Transmission Control Protocol
14       |  |--Source port: 32517 (32517)
15       |  +--Destination port: 80 (80)
16       +--Data
17          |--Data: 436f6d6d756e69636174696f6e2068617320626565e20626c6f636b65642e...
18          +--[Length: 126 Bytes]
```

Listing 3.2: Example blocking result of a router.

3.5.3. Example Proxy Server Implementation

The Application-level Proxy Ontology has been implemented as a prototypical proxy server. The preparation module of this proxy server resolves any existing conflicts of flow control rules and stores the remaining rules in a relational database. This database serves as the proxy server's rule storage. Whenever the proxy server's regulation module receives a request for a particular URL, the URL is looked up in the rule storage. If the URL is not found, the request is allowed. Otherwise, the proxy server performs a corresponding regulation. The regulation module of the proxy server is implemented in Java and uses standard Java libraries. A detailed description of its implementation is provided in [18].

An example request for the website `http://www.porntube.com/` using the Unix tool cURL[18] looks like: `curl -v http://www.porntube.com/`. The website provides pornographic content. According to the example flow control rule `alpr-1` depicted in Figure 3.19a, access to this website is to be blocked by the proxy server `pr-1` of the German comprehensive school. The regulation affects all student computers of the school's computer network `cn-5` shown in Figure 3.14. The response of the proxy server is depicted in Listing 3.3. Line 1 contains the status code returned by the server. The status code 451 indicates that the access to the requested website was denied due to legal reasons [48].

[18] `https://curl.haxx.se/`, last accessed: 01/21/16

```
1 < HTTP/1.1 451 Unavailable For Legal Reasons
2 < Content-Length: 406
3 < Content-Type: text/html; charset=iso-8859-1
4 <
5 <html>
6   <head>
7     <title>Unavailable For Legal Reasons</title>
8   </head>
9   <body>
10    <h1>Unavailable For Legal Reasons</h1>
11    <p>The web page you are trying to access is not accessible due to legal reasons.
12    For more information about the regulation see <a
13    href="http://icp.it-risk.iwvi.uni-koblenz.de/policies/proxyPolicy01.owl#alpr-1"
14    >the regulation details</a>.</p>
15  </body>
16 </html>
```

Listing 3.3: Example blocking result of a proxy server.

Along with the status code, the proxy server also returns a short web page providing a human-readable explanation about the regulation's background. The web page is shown in lines 5 to 16 and contains a hyperlink which refers to the flow control rule that initiated the regulation.

3.6. Evaluation and Comparison with Existing Approaches

This section evaluates how the related work discussed in Section 3.1 and the InFO policy language fulfill the requirements introduced in Section 3.2. The results of this assessment are shown in Table 3.2. Most of the reviewed policy languages and content labeling schemes focus on a particular application and do not fulfill all requirements. In the following, the related work is analyzed regarding the functional requirements **RA.F.1** to **RA.F.9** as well as the non-functional requirements **RA.N.1** to **RA.N.3**.

3.6.1. Evaluating the Functional Requirements

Access control systems require the user accessing a digital resource to be authenticated first. As this authentication process is usually done by an application layer protocol, access control languages such as AMO, Common Policy, WebAccessControl, EPAL, and XACML exclude routers as possible enforcing systems (**RA.F.1.1**). These systems operate on lower layers of the OSI model such as the network layer and possibly the transport layer as well. EPAL and XACML are designed to be used separately from the server providing the regulated content while AMO, Common Policy, and WebAccessControl require a more close integration with the server. This allows EPAL and XACML policies to be enforced by proxy servers (**RA.F.1.3**), while policies created with AMO, Common Policy, or WebAccessControl can only be enforced by the content providing server. Although the flow control languages DEN-ng, OPoT, and the firewall metamodel

Table 3.2.: Comparison of different policy languages and content labeling schemes with the requirements introduced in Section 3.2. Rows correspond to the different approaches and columns to requirements. Requirements **RA.F.1.1** to **RA.F.9.2** are functional, while **RA.N.1** to **RA.N.3** are non-functional. The letter y represents a complete fulfillment of the requirement, l stands for a partial fulfillment, and n corresponds to no fulfillment of the requirement.

	RA.F.1.1: Routers as enforcing nodes	**RA.F.1.2**: Name servers as enforcing nodes	**RA.F.1.3**: Proxy servers as enforcing nodes	**RA.F.2**: Operationalization of policies	**RA.F.3.1**: Allowing modality	**RA.F.3.2**: Denying modality	**RA.F.4.1**: Conflict resolution for one policy	**RA.F.4.2**: Conflict resolution for multiple policies	**RA.F.5.1**: Regulation enforcer	**RA.F.5.2**: Regulation provider	**RA.F.5.3**: Regulation legislator	**RA.F.6**: Content identification	**RA.F.7**: Content classification	**RA.F.8**: Access location	**RA.F.9.1**: Organizational background	**RA.F.9.2**: Legal background	**RA.N.1**: Standards Compliance	**RA.N.2**: Modularity	**RA.N.3**: Extensibility
Access Control																			
AMO [58]	n	n	n	n	y	n	n	n	n	n	n	y	n	n	n	n	y	n	y
CommonPolicy [276]	n	n	n	n	y	n	n	n	n	n	n	n	y	n	n	n	y	n	y
EPAL [15]	n	n	y	l	y	y	l	n	n	y	n	n	y	y	n	n	y	n	y
WebAccessControl	n	n	n	y	y	n	n	n	n	n	n	y	n	n	n	n	y	n	y
XACML [205]	n	n	y	l	y	y	y	y	n	y	n	y	y	y	n	n	y	n	y
Flow Control																			
Cuppens et al. [85]	y	n	n	y	y	n	n	n	n	n	n	n	n	y	y	n	y	n	n
DEN-ng [286]	y	n	n	y	y	y	l	l	n	n	n	n	n	y	n	n	n	n	y
OPoT [21]	y	n	n	y	y	y	l	l	n	n	n	n	n	y	y	n	y	n	n
Usage Control																			
ccREL [3]	n	n	n	n	y	y	n	n	n	n	n	y	n	n	n	l	y	n	l
LDR [253, 254]	n	n	n	n	y	y	n	n	n	n	n	y	y	n	n	l	y	n	y
ODRL [156, 157]	n	n	n	n	y	y	n	l	n	y	n	y	n	y	n	n	y	l	l
METSRights	n	n	y	n	y	n	n	n	n	y	n	y	l	n	n	n	y	n	l
MPEG-21 REL [306]	n	n	n	n	y	n	n	n	n	y	n	y	n	n	n	n	y	l	l
PLUS	n	n	n	n	y	l	n	n	n	y	n	y	l	y	l	n	l	n	n
General purpose																			
KAoS [298]	n	n	y	n	y	y	y	n	n	y	n	n	n	y	y	n	y	y	y
Rei [166]	n	n	y	n	y	y	l	n	n	y	n	y	y	y	n	n	y	y	y
Ponder [88]	y	n	y	y	y	y	y	n	n	n	n	y	y	y	n	n	n	n	y

Continued on next page.

Table 3.2.: Comparison of the related work with InFO. *Continued from previous page.*

	R.A.F.1.1: Routers as enforcing nodes	**R.A.F.1.2**: Name servers as enforcing nodes	**R.A.F.1.3**: Proxy servers as enforcing nodes	**R.A.F.2**: Operationalization of policies	**R.A.F.3.1**: Allowing modality	**R.A.F.3.2**: Denying modality	**R.A.F.4.1**: Conflict resolution for one policy	**R.A.F.4.2**: Conflict resolution for multiple policies	**R.A.F.5.1**: Regulation enforcer	**R.A.F.5.2**: Regulation provider	**R.A.F.5.3**: Regulation legislator	**R.A.F.6**: Content identification	**R.A.F.7**: Content classification	**R.A.F.8**: Access location	**R.A.F.9.1**: Organizational background	**R.A.F.9.2**: Legal background	**R.A.N.1**: Standards Compliance	**R.A.N.2**: Modularity	**R.A.N.3**: Extensibility
Content labeling																			
age-de.xml [264]	n	n	y	l	y	y	l	n	n	y	n	y	l	l	n	n	y	n	l
PICS [183, 103]	n	n	y	n	y	y	l	n	n	y	n	y	y	n	l	l	l	l	l
RTA	n	n	y	n	n	y	n	n	n	l	n	n	l	n	n	n	y	n	n
InFO	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

focus on managing low-level enforcing nodes such as routers (**RA.F.1.1**), they are not designed to define policies enforced by proxy servers or name servers. Usage control policies also require an enforcement at the application layer. These policies are generally enforced at the user's site as this requires the user's actions to be monitored. Thus, ccREL, LDR, ODRL, MPEG-21 REL, and PLUS do not fulfill requirements **RA.F.1.1** to **RA.F.1.3**. However, METS is designed to be used within closed library environments making METSRights suitable to be enforced at the library's proxy server (**RA.F.1.3**). KAoS and Rei focus on rather abstract behavioral policies which are also designed to be enforced by application-layer systems. This makes it possible to enforce their policies by proxy servers. On the other hand, Ponder allows to define policies which can be enforced by almost arbitrary communication nodes including end user systems, content-providing servers, or intermediary communication nodes such as routers. However, neither Ponder nor any other of the evaluated policy languages support name servers as enforcing nodes (**RA.F.1.2**). Name servers are not part of the communication path between a content provider and a content consumer. Instead, they only provide a means for establishing this communication path, which is not covered by any of the languages depicted in Table 3.2. Content labeling schemes such as age-de.xml, PICS, and the RTA label allow to annotate web content which is processed at the application layer together with its annotations. Thus, these schemes do not support an enforcement by routers

or name servers and do not fulfill requirements **RA.F.1.1** and **RA.F.1.2**. As both the web content and their annotations can be interpreted by proxy servers, content labeling schemes fulfill requirement **RA.F.1.3**. InFO supports different enforcing systems including routers, name servers, and proxy servers. Each enforcing system is supported by a specific domain ontology such as the Router Ontology, the Name Server Ontology, and the Application-Level Proxy Ontology.

Many of the examined policy languages and content labeling schemes define rather abstract rules whose enforcement cannot be directly mapped onto the enforcing systems' capabilities. Instead, the enforcement requires additional parameters and a further interpretation of how to interpret the policy's actual meaning. These parameters are sometimes not included in the policy directly (**RA.F.2**) and must therefore be added through a different process. Although policies created with AMO or Common Policy contain a reference to those users who are allowed to access a specific piece of information, they do not define how the users shall be authenticated. EPAL and XACML provide such a description but do not explicitly define the rest of the enforcement procedure. WebAccessControl requires user identification via the WebID authentication process. Usage control only describes on an abstract level what a user may do with a digital resource. However, usage control languages do not define how the permitted or prohibited actions shall actually be regulated. For example, it is unclear how a permission to print a text document is to be technically enforced. Thus, the evaluated usage control policies do not fulfill requirement **RA.F.2**. Since the general purpose languages Rei and KAoS also define rather abstract policies and not their specific enforcement, they also do not fulfill requirement **RA.F.2**. However, Ponder's low-level language can be directly used for enforcing mechanisms. Although content labeling schemes are designed to block the access to web content based on their annotations, most schemes do not define how the blocking procedure shall actually be implemented. age-de.xml allows to redirect all requests to another website instead of just blocking the access. However, it does not support a precise definition of how the redirection or the default blocking behavior shall be technically implemented. On the other hand, flow control languages are specifically designed for a direct enforcement of policies. Each created policy already contains enough information to be enforced without requiring any additional interpretation or parameters. Similarly, InFO is also designed for enforcing particular policies. Support for a precise description of all enforcement parameters are provided by the domain extensions of InFO such as the Router Ontology, the Name Server Ontology, and the Application-Level Proxy Ontology.

All policy languages for access control, usage control, and flow control as well as all general purpose languages shown in Table 3.2 support allowing rules and thus fulfill requirement **RA.F.3.1**. Allowing rules are also supported by PICSRules. age-de.xml supports allowing rules by annotating web content with the *all ages* category. In contrast, the RTA label does not fulfill requirement **RA.F.3.1** as it represents a single age category which results in the blocking of the annotated content. In order to ease the creation of specific policies, some languages such as AMO, Common Policy, WebAccessControl, and the firewall metamodel do not support denying rules (**RA.F.3.2**) and focus on allowing rules only. In doing so, these languages completely avoid potential conflicts between

two or more contradicting rules. Thus, they do not provide any means for resolving such conflicts (requirements **RA.F.4.1** and **RA.F.4.2**). Similarly, the RTA label only provides a means for denying access and does not require any conflict resolution algorithm as well. PLUS only supports allowing rules and expresses denying rules as constraints on such a rule. Consequently, the support for denying rules is only limited and the language does not support any conflict resolution mechanism. InFO allows to create both allowing and denying rules as well as specific types of denying rules. The different types of rules are provided by the Flow Control Rule Pattern as well as the Redirecting Flow Control Rule Pattern and the Replacing Flow Control Rule Pattern described in Section 3.3.3.

The usage control policy languages ccREL and LDR assume that there is only one policy for each regulated good which is created by its owner. Thus, the languages do not provide any means for resolving conflicts between contradicting rules. EPAL, Rei, and PICSRules only provide the order of rules as a mechanism for resolving conflicts between rules of the same policy (**RA.F.4.1**). Since all three languages support only one active policy, they do not provide any means for resolving conflicts between rules of different policies (**RA.F.4.2**). Similarly, age-de.xml uses the order of the defined age categories for a single web page to resolve any conflicts between contradicting categories. DEN-ng also uses the order of rules for resolving conflicts between them. Additionally, policies can also be ordered to resolve conflicts between rules of different policies. OPoT is only able to detect conflicting rules of one or more policies and shows them to the policy's creator. The actual conflict resolution must be performed manually by the creator of the policy. ODRL assumes that a single policy does not contain any conflicting rules. Since each policy is created by a single party, the party must pay attention when creating the policy. On the other hand, resolving conflicts between rules of different policies is supported by ODRL. This is done by either preferring the allowing or the denying rule of two contradicting rules. However, the preferred rule modality is defined within each policy. Thus, ODRL's conflict resolution algorithm only works if all affected policies prefer the same modality. Otherwise, conflicts cannot be resolved [24]. XACML and Ponder allow to define specific conflict resolution algorithms which provide a much greater flexibility than a simple order of rules. These algorithms can be used for resolving conflicts between rules of one or more policies. KAoS also provides algorithms for resolving conflicts. Since KAoS only supports one active policy, these algorithms can only be used for contradicting rules of one policy. The conflict resolution mechanisms of InFO are inspired by XACML. Similar to XACML and Ponder, InFO allows to resolve conflicts between contradicting rules of one or multiple policies based on predefined or user-defined algorithms. The Flow Control Meta-Policy Pattern described in Section 3.3.5 is especially designed for expressing such algorithms and thus for resolving conflicts as well. The pattern also splits the whole conflict resolution process of XACML and Ponder into four different steps and assigns a particular algorithm to each step. This achieves a greater flexibility as some algorithms can be reused for different enforcing systems while others must be replaced with more specific algorithms.

Most of the discussed policy languages do not distinguish between a policy's creator (i. e. the provider) and its enforcer. Languages like EPAL, XACML, ODRL, METS-Rights, MPEG-21 REL, PLUS, KAoS, Rei, and PICSRules, which allow naming a pol-

icy's provider within the policy itself (**RA.F.5.2**), do not allow to name a separate enforcer (**RA.F.5.1**). Similarly, the content labeling scheme age-de.xml only supports providers of age categories. However, InFO explicitly requires such a distinction as outlined in the scenario for regulating Internet communication of Section 2.1. Support for a regulation's provider is given as the rule data provider defined in the Flow Control Rule Pattern and a means for defining the regulation's enforcer is given in the Flow Control Policy Pattern described in Section 3.3.4.

Most of the reviewed policy languages are not able to link a policy to its legal background (**RA.F.9.2**). Consequently, they do not allow to specify the legislator of the policy's legal background (**RA.F.5.3**). Both ccREL and LDR allow policies to be linked to their jurisdiction. This jurisdiction defines the circumstances under which a policy is valid and may even add additional permissions or prohibitions. However, the jurisdiction only refers to a country's legislation and not to particular laws. Identifying the creators of this legislation is neither supported by ccREL nor by LDR. InFO supports both relating a flow control to its legal background and the definition of a legislator as well. The legal background is described by the Flow Regulation Norm Pattern introduced in Section 3.3.6 and the legislator is part of the Legislation Pattern. By linking technical policies to their legal background, InFO allows a better comparison between different policies of various enforcing systems.

The identification and classification of content (**RA.F.6**) is only supported by those policy languages which are directly able to regulate the processing of particular information documents rather than whole systems or services only. Such languages include access control and usage control languages as well as most general purpose languages. Both AMO and WebAccessControl require the explicit identification of the document to be protected by using an URI (**RA.F.6**). Content classification is neither supported by AMO nor by WebAccessControl. On the other hand, both Common Policy and EPAL only support classes of documents (**RA.F.7**) but do not allow a more precise identification of the content. Data classification with Common Policy can be achieved by using the *sphere* constraint whereas EPAL provides *data categories*. XACML allows to regulate access based on either the contents' ID or its topic. Usage control languages are designed to control the consumption of digital resources. Thus, they all support a precise identification of the content to be controlled. METSRights also supports a simple classification of the content according to its licensing status such as *copyrighted* or *licensed*. However, an actual content description is not supported. Similarly, PLUS and the RTA label only support the classification of adult content but do not provide more precise content descriptions. Flow control languages only focus on regulating communication between complete systems and thus do not fulfill requirements **RA.F.6** and **RA.F.7**. age-de.xml uses URIs to associate web content with its respective age category and thus fulfills requirement **RA.F.6**. However, the precise topic of the content cannot be described. On the other hand, the RTA label is directly embedded into the labeled web content and does not require any content identifier. PICS supports precise content descriptions by associating the content's URI with arbitrary labels and thus fulfills requirements **RA.F.6** and **RA.F.7**. Although InFO also focus on regulating flow control,

it considers the topic of the regulated content as well. The Flow Control Rule Pattern associates each particular flow regulation with such a topic.

The location of the user who wants to access a regulated resource can be implemented in different ways. EPAL, XACML, ODRL, KAoS, Rei, Ponder, and PLUS support constraints regarding the applicability of their rules. These constraints also cover location constraints which directly implement requirement **RA.F.8**. All evaluated flow control languages are able to regulate network communication using IP addresses. Since these addresses can be mapped to a geographical location, the flow control languages fulfill requirement **RA.F.8** as well. age-de.xml fulfills requirement **RA.F.8** by supporting ISO 3166-1 alpha-2 country codes [161] such as `DE` for defining the user's location. InFO also uses IP addresses to refer to a requesting user's country and thus supports requirement **RA.F.8**.

Organizational background information corresponds to the enforcer's motivation to implement a specific policy. Although some of the analyzed policy languages provide a *purpose* constraint, this property does not correspond to an actual explanation of a policy's meaning and function. Instead, it only restricts the applicability of allowing and denying rules to a specific use case. The firewall metamodel and OPoT are designed for mapping high-level organizational security policies to their technical representation. Such a design also allows to link policies created with one of these languages to their corresponding security policy. In doing so, the policies are enriched by a human-readable description and thus implement requirement **RA.F.9.1**. KAoS and PICSRules follow a different approach by directly embedding human-readable descriptions into a created policy using the properties `hasDescription` and `description`, respectively. As these properties are designed for annotating arbitrary strings, they can be used for describing both organizational and legal background information. PLUS provides several attributes for embedding human-readable conditions and restrictions into the policy. However, these attributes are not sufficient for describing the complete organizational background of a particular policy as each of the attributes only covers a specific part of it. InFO allows to express a regulation enforcer's code of conduct with the Code of Conduct Pattern described in Section 3.3.6.

3.6.2. Evaluating the Non-Functional Requirements

Most of the policy languages and content labeling schemes are based on standard formats such as XML, RDFS, or OWL and thus support requirement **RA.N.1**. DEN-ng uses UML for describing policies. Although UML is a well-known standard, it cannot be directly used for implementing policies. Instead, policies modeled as UML diagrams have to be mapped to other formats which can natively be interpreted by an enforcing system. As DEN-ng does not provide such a mapping, it does not fulfill requirement **RA.N.1**. Although PLUS uses an RDFS ontology, many essential parts of a PLUS policy are encoded as structured string values. Since these string values have their own proprietary format, PLUS does not completely fulfill requirement **RA.N.1**. Ponder uses its own proprietary format which is not compatible with other formats such as XML or OWL. Although PICS and PICSRules also use their own format, a suggestion for mapping

PICS labels to a lightweight RDFS ontology is provided in [51]. However, the ontology does not cover PICSRules and only focuses on PICS labels. InFO is modeled as an OWL ontology and thus fulfills requirement **RA.N.1**.

None of the evaluated access control languages has a modular design and thus do not fulfill requirement **RA.N.2**. Instead, their specification consists of a single document which cannot be further partitioned into different sections. However, the main entities defined by the languages can still be extended with additional terms such as new actions or new roles (**RA.N.3**). The examined flow control languages solely focus on network management and already provide a sufficient vocabulary for expressing corresponding policies. Due to their restricted use case and their straightforward design, they do not provide a modular structure or a broad extensibility. However, based on its open design on UML, DEN-ng is still able to be extended with additional language elements. Based on their lightweight design, neither ccREL, LDR, nor PLUS have a modular structure. Although both ccREL and LDR can be extended with additional terms, using such terms in a ccREL policy may result in a policy which no longer corresponds to a Creative Commons license. In contrast, PLUS cannot be extended with additional terms. ODRL and MPEG-21 REL define an REL and a separate RDD. Since the default RDD is not mandatory and can be replaced with a user-defined one, the separation between the REL and the RDD can be considered as limited modularity. However, the REL of both languages itself is not modular. The extendability of both ODRL and MPEG-21 REL is limited to defining new vocabulary terms for their corresponding RDD such as new actions or constraints. Adding new entities to their REL's model is not possible. METSRights does not define separate specifications for an REL and an RDD and thus cannot be considered as modular. However, it is still possible to add new terms to the language's vocabulary. KAoS and Rei are based on OWL. The concepts of these languages are separated into different ontologies, each of which covers a specific aspect of them. For example, both languages define an ontology for describing actions and a separate ontology for policies. Since the languages are based on OWL, they also fulfill requirement **RA.N.3** by supporting user-defined extensions. Although Ponder also supports the definition of new language entities such as new rule types, its proprietary representation format does not permit a modular design (**RA.N.2**). Due to their simple design, neither age-de.xml nor the RTA label have a modular design. However, age-de.xml can be extended with new XML elements. The PICS framework consists of the PICS labeling scheme and PICSRules which are two separate formats. Since both formats can also be used separately from each other, PICS supports a limited modularity. The PICS labels can also be extended with arbitrary new labels, similar to the RDD of a rights expression language. However, PICSRules cannot be extended with new types of rules. InFO's modular design consists of several ontology design patterns (**RA.N.2**). Many of these patterns can be extended with new concepts such as introducing new rule types as subclasses of `FlowControlRuleMethod`. Furthermore, InFO is specifically designed to be extended with domain-specific ontologies that cover concepts relevant for particular use cases.

3.6.3. Summary

None of the evaluated policy languages and content labeling schemes can be used for regulating information flow control in open and distributed networks such as the Internet. However, InFO reuses some of their concepts such as meta-policies and different conflict resolution algorithms. InFO's extendability is inspired by Ponder which allows to model arbitrary types of communication flow. XACML's flexible conflict resolution algorithms are also adopted by InFO. Since InFO is a pattern system which covers a core ontology, ontological languages such as AMO, WebAccessControl, KAoS, and Rei may be aligned as domain specific extensions. Other domain specific extensions are also possible and cover the integration of content labeling schemes such as PICS or age-de.xml into InFO. Such extensions can be used to further describe the content of a regulated Internet communication.

3.7. Limitations and Possible Extensions

As demonstrated in the previous section, the InFO policy language fulfills all functional and non-functional requirements defined in Section 3.2. However, InFO still has some limitations when applying a particular policy to an enforcing system. This section first describes these limitations and their causes. Afterwards, possible extensions to InFO are discussed which provide additional features.

3.7.1. Enforcing InFO Policies

InFO primarily focuses on providing a policy language for precisely describing how a regulation shall be technically implemented. However, the policy language itself does not to provide any means for ensuring that an enforcing system interprets and implements a regulation correctly. It is also possible that the enforcing system's behavior does not comply with the intended meaning of a policy and thus implements a regulation which differs from the provided flow regulation policies. In order to reduce the possibility of such effects, each enforcing system should be tested according to its conformance to the InFO policy language. Such a conformance test should be conducted by an independent party in order to reduce the chance of manipulating the test's procedure and outcome. Systems which pass the test should be certified accordingly and the resulting certificate should be provided to all parties involved in a regulation. Similarly, the responsible operator of the enforcing system should also be tested and certified in order to eliminate any organizational mistakes in the regulation's implementation.

3.7.2. Legal Background

InFO provides a solution for a technical regulation of Internet communication without requiring any manual interaction. Although each policy is associated with its legal and/or organizational background, this background is primarily used as the policy's human-readable explanation. It is expressed using external ontologies such as LKIF [146, 147]

or CLO [123, 118] which are integrated into InFO. However, even existing legal ontologies may not be able to completely replace every human intervention. As Brown and Greenberg [55] have demonstrated, not all legal cases are formally decidable and require manual interaction instead. Thus, InFO considers the mapping from an organizational background and/or a legal background to a technical regulation to be a manual process as well. The process may supported by KORA (Konkretisierung rechtlicher Anforderungen; concretizing of legal requirements) [139], a methodology for deriving technical requirements from legal requirements.

In addition to this creation process, some legal regulations also contain exceptions regarding the affected users. For example, the German Criminal Code contains several norms related to computer crimes including §202c [63]. §202c prohibits the creation and distribution of software tools which can be used for conducting computer crimes. However, §202c does not apply to security experts who use the software tools for assessing the security of a computer system [277]. The treatment of such exceptions generally requires human intervention and is usually done by courts [277]. Thus, a completely automatic assessment of a particular situation is not always possible. Even a security expert may violate §202c if she uses software tools to deliberately sabotage a computer system without having a proper authorization. Although InFO can easily be extended with additional roles representing the intervening parties, the actual interpretation of the exceptions would still require a manual intervention.

3.7.3. Consistency Between Different Layers

An InFO flow control policy associates several flow control rules with their organizational motivation and/or legal authorization. As described in the previous section, deriving a particular flow control policy from an organizational code of conduct and/or legal norm is considered to be a manual process. Similarly, ensuring that the technical regulation details comply with its organizational and legal background is a manual process as well. Although the Flow Regulation Norm Pattern described in Section 3.3.6 can be considered as a legal view on the technical Flow Control Rule Pattern described in Section 3.3.3, InFO does not provide any means for checking the consistency between both patterns. Instead, the patterns can only be used as a basis for evaluating whether or not a technical flow control policy complies with its organizational and legal background. The actual evaluation must be conducted via a manual process. However, this process may be supported by ontological reasoning on the InFO policies and by evaluating additional rules such as RIF expressions [178].

3.7.4. Supporting Child Protection Software

Child protection software aims at prohibiting minors from accessing adult web content. Most software applications focus on regulating the Internet access of a local home environment and can be installed on local proxy servers, home routers, or client computers. Although some child protection software such as NetNanny[19] and the Jugendschutzpro-

[19] see `https://www.netnanny.com/support/changelog/`, last accessed: 01/21/16

gramm[20] evaluate third-party formats such as the RTA label or age-de.xml as well, their regulation is mainly based on a proprietary rule syntax created via the software's user interface. Since child protection software focuses on a similar use case as the InFO policy language, it is possible to configure the software by using corresponding InFO policies. This would achieve a greater interoperability between different software products and also between different installations of the same product. Regulation policies created with InFO could then be imported into any child protection software without having to reconfigure any specific regulation. This could also be used for separating the developers of child protection software from the providers of regulation rules. A particular software installation could then be configured with regulation rules from different providers. Supporting the configuration of child protection software with InFO can be achieved by providing corresponding domain ontologies.

3.7.5. Integration into Software Defined Networking

Software Defined Networking (SDN) [225] defines a generic architecture for flexible and dynamic management of closed networks which are centrally administrated by a single organization. SDN generally distinguishes between the controller plane and the data plane. The controller plane is a central control node which manages and configures other network nodes such as routers and switches. These network nodes correspond to the data plane and carry out the actual packet forwarding. The configuration of these nodes is done via a specific protocol. This protocol and the distinction between the controller plane and the data plane are the main components of the SDN architecture. Since SDN defines a generic architecture, a particular implementation of all three components is not provided. Instead, different protocols and even different routers and switches can be used. The only requirement is that all three components of the architecture are compatible with each other. InFO can be a used as part of the SDN protocol for exchanging regulation information between the controller plane and the data plane. However, as InFO does not provide such a protocol itself, designing and implementing a complete protocol for both the controller plane and the data plane would still be necessary. A first step towards this integration has been conducted by implementing policy-based regulation rules on routers which is described in Section 3.5.2 and further explained in [311]. The implementation also uses a central administration node for configuring a set of routers with InFO regulations.

3.8. Summary

This chapter has presented InFO, a policy language for regulating information flow in open and distributed networks such as the Internet. Regulations expressed with InFO can be implemented on different types of enforcing systems such as routers, application-level proxy servers, and name servers. Additionally, other types of enforcing systems can also be supported due to InFO's modular and extensible design. Each regulation

[20] See `http://www.jugendschutzprogramm.de/faq6.php`, last accessed: 01/21/16

policy consists of several rules which contain precise technical details for implementing a particular regulation. Any conflicts between two or more rules are eliminated by InFO's conflict resolution mechanism. The purpose of a policy is described by attaching human-readable background information such as a regulation's legal foundation and the organizational motivation. InFO achieves compliant availability and thus answers research question **RQ.4** by restricting the availability of information to authorized parties only. Parties are considered to be authorized as long as they comply with the InFO regulations.

Chapter 4.

Siggi: A Framework for Iterative Signing of Graph Data

This chapter presents Siggi, a generic framework for iterative signing of Semantic Web graph data. Signing graph data is a security mechanism for achieving integrity and authenticity of the data [28]. Thus, the graph signing framework Siggi answers the research questions **RQ.2** and **RQ.3**. The framework is independent from any particular algorithm or software implementation. It can be configured to achieve different features such as minimum signature overhead or minimum runtime. The framework divides the signing process into multiple steps, each of which can be implemented with different algorithms. Due to its generic design, the framework also serves as a guideline for creating new algorithms. It provides various features such as signing Named Graphs, signing multiple graphs at once, and iterative signing of graph data. Iteratively signing graph data is the process of signing already signed graph data again. This can be used for provenance tracking which documents the data flow of the signed data. A signature created with the graph signing framework Siggi is independent of the graph's encoding. The signature only covers the actual contents of the graph's triples but not their syntactical representation. This allows it to change the order of the triples within a graph or rename the local identifiers of the graph's blank nodes without invalidating the signature. Prior versions of the graph signing framework were published in [168, 169, 170, 173]. This chapter is based on these publications but rephrases, consolidates, and extends them with additional aspects.

The remainder of this chapter is organized as follows: The state of the art and related work for achieving integrity and authenticity of graph data is summarized in Section 4.1. Based on this section and on the scenarios introduced in Chapter 2, Section 4.2 defines the functional and non-functional requirements for the graph signing framework Siggi. The formal specification of Siggi is provided in Section 4.3. Section 4.4 presents four example configurations of the signing framework. These configurations are further analyzed in Sections 4.5 and 4.6 which conduct a detailed cryptanalysis and performance analysis, respectively. Section 4.7 demonstrates how the graph signing framework is used for signing different types of graph data. Section 4.8 assesses the state of the art and related work and compares it with the framework. Limitations and possible improvements of Siggi are discussed in Section 4.9 before the chapter is concluded.

4.1. State of the Art and Related Work

The signature framework Siggi divides the process of signing graph data and verifying their signatures into different steps as depicted in Figure 4.1. These steps are based on the XML signature syntax and processing standard [20] for signing and verifying XML documents. Each of these steps can be implemented by different algorithms making them interchangeable with each other. Thus, the framework forms a basis for different graph signing implementations. After having loaded the graph data into memory, a *canonicalization function* κ_N [193] normalizes the data to a unique form. Second, a *serialization function* ν_N transforms the canonicalized data into a sequential representation. Third, a *hash function for graphs* λ_N computes a cryptographic hash value on the serialized data. Fourth, a *signature function* φ combines the data's hash value with a private signature key [268]. The results of the first four functions are combined to the *graph signing function* σ_N. Fifth, an *assembly function* α_N creates a signature graph containing all data for verifying the graph's integrity and authenticity including the signature value and an identifier of the signature verification key. The actual verification is conducted in the last step by the *verification function* δ_N.

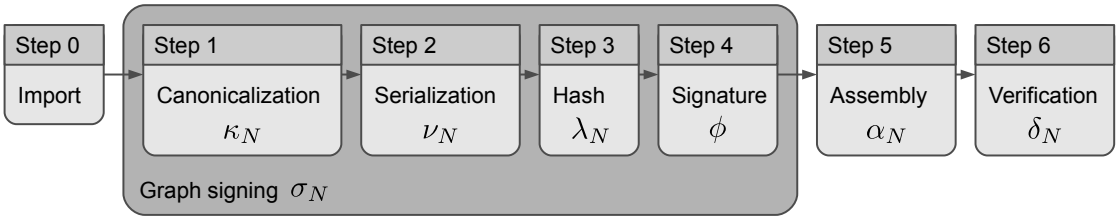


Figure 4.1.: The general process of signing and verifying graph data (cf. [20]).

This section presents the state of the art and related work on signing graphs along the individual graph signing sub-functions as depicted in Figure 4.1. For each sub-function, its runtime complexity and space complexity is discussed as well. Subsequently, existing assembly functions are presented. Verification functions operate similarly to graph signing functions and use the same sub-functions or their inverse. Thus, they are not discussed in more detail. This section concludes with a discussion of alternative approaches for achieving integrity and authenticity of graph data. A formalization of all functions is provided in Section 4.3. Table 4.1 summarizes the complexity of different implementations of the four sub-functions. In the table, n refers to the number of triples to be signed and b corresponds to the number of blank nodes in the graph. A detailed comparison with the related work on graph signing functions and the signing framework Siggi is given in Section 4.8.

4.1.1. Graph Signing Functions

Tummarello et al. [295] present a graph signing function for fragments of RDF graphs. These fragments are minimum self-contained graphs (MSGs) and are defined over triples. An MSG of a triple t is the smallest subgraph of the complete RDF graph that contains t

and the triples of all blank nodes associated directly or recursively with t. Triples without blank nodes are an MSG on their own. The graph signing function of Tummarello et al. is based on Carroll's canonicalization function and hash function [72] described in Section 4.1.2. A signature is stored as six triples, which are linked to the signed MSG via RDF statement reification [142] of one of the MSG's triples. The approach of Tummarello et al. only supports signing one MSG at a time. Signing a full graph with multiple MSGs requires multiple signatures. Thus, the graph signing process depicted in Figure 4.1 has to be applied to each MSG in the graph. This creates a large overhead of six signature triples per MSG. Furthermore, sining arbitrary sets of triples which do not correspond to complete MSGs is not supported by the approach.

Signing a graph can also be accomplished by signing a document containing a particular serialization of the graph [262]. For example, a graph can be serialized using an XML-based format such as RDF/XML [26] or OWL/XML [210]. This results in an XML document which can be signed using the XML signature standard [20]. If the graph is serialized using a plain text-based format such as the triple-based serialization formats N-Triples [25] or Turtle [27], also standard text document signing approaches may be used [268]. However, these approaches inextricably link the signature with a concrete encoding of the graph [262]. Consequently, such a signature can only be verified as long as the very specific serialization of the graph contained in the document is provided.

4.1.2. Canonicalization Functions for Graphs

A *canonicalization function* κ_N deterministically normalizes a graph in such a way that its syntactical representation does not affect its hash value and its signature. A canonicalization function ensures that a graph's signature only covers its semantics and not its syntax [193]. As depicted in Figure 4.1, a cryptographic hash function is used for computing a hash value of a serialized and canonicalized graph. Such a hash function operates on string values and outputs different hash values for different input strings. Blank node identifiers of a graph also influence the graph's hash value and thus its signature as well. These blank node identifiers can be consistently renamed within the whole graph without modifying the graph's semantics. This means that a particular blank node identifier can be renamed to another identifier without destroying the graph's meaning as long as this identifier is renamed in all its occurrences. However, renaming blank node identifiers changes the graph's syntactical representation and thus its hash value and signature as well. A canonicalization function prohibits blank node renaming from invalidating a graph's signature. A formal specification for canonicalization functions is provided in Section 4.3.3.

Hogan [148] canonicalizes an RDF graph by computing all its isomorphic graphs, sorting them, and selecting the first graph as canonical graph. A single isomorphic graph is computed by replacing all blank nodes in the graph with the same identifier, computing the hash values of all triples in the graph, and computing the hash values of all blank nodes by using the hash values of the triples in which they occur. If several blank nodes share the same hash value, one of the blank nodes is replaced by a new identifier and the hash values are computed again. This process is repeated until the hash values of

Table 4.1.: Complexity of the sub-functions used by the graph signing function σ_N for signing a single graph with n triples and b blank nodes.

Function	Example	Runtime	Space
Canonicalization κ_N	Hogan [148]	$\mathcal{O}(b^2 b!)$	$\mathcal{O}(b!)$
	Carroll [72]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Fisteus et al. [110]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Sayers and Karp [261]	$\mathcal{O}(n)$	$\mathcal{O}(b)$
	Kuhn and Dumontier [184]	$\mathcal{O}(n)$	$\mathcal{O}(b)$
Serialization ν_N	N-Triples [25]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	Turtle [27]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	N3 [34]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	TriG [38]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	TriX [75]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	RDF/XML [26]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	OWL/XML [210]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Hash λ_N	Melnik [200]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Carroll [72]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Fisteus et al. [110]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
	Sayers and Karp [261]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Signature φ	ElGamal [97]	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	RSA [251]	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	DSA [214]	$\mathcal{O}(1)$	$\mathcal{O}(1)$

all blank nodes are unique. All isomorphic graphs are computed by replacing different blank nodes with a unique identifier in each run. Creating a single isomorphic graph renames b blank nodes which is done in at most $b - 1$ iterations. This corresponds to a runtime complexity of $\mathcal{O}(b^2)$. Since there are $b!$ different isomorphic graphs for a graph with b blank nodes, the runtime complexity for computing all isomorphic graphs is $\mathcal{O}(b^2 b!)$. Sorting all $b!$ isomorphic graphs with a sorting algorithm such as merge sort [241] results in a overall space complexity of $\mathcal{O}(b!)$. Carroll [72] presents a canonicalization function for RDF graphs that replaces all blank node identifiers with uniform place holders, sorts all triples of the graph based on their N-Triples [25] representation, and renames the blank nodes according to the order of their triples. If this results in two blank nodes having the same identifier, additional triples are added for these blank nodes. Carroll's canonicalization function uses a sorting algorithm with a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$ [72]. Fisteus et al. [110] perform a canonicalization of blank node identifiers based on the hash values of a graph's triples. First, all blank nodes are associated with the same identifier. Second, a triple's hash

value is computed by multiplying the hash values of its subject, predicate, and object with corresponding constants and combining all results with XOR modulo a large prime. If two triples have the same hash value, new identifiers of the blank nodes are computed by combining the hash values of the triples in which they occur either directly or transitively. This process is repeated until there are no collisions left. Colliding hash values are detected by sorting them. Using merge sort as sorting algorithm leads to a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$. Sayers and Karp [261] provide a canonicalization function for RDF graphs which stores the identifier of each blank node in an additional triple. If the identifier is changed, the original one can be recreated using this triple. Since this does not require sorting the triples, the runtime complexity of the function is $\mathcal{O}(n)$. In order to detect already processed blank nodes, the function maintains a list of additional triples created so far. This list contains at most b entries with b being the total number of additional blank node triples. Thus, the space complexity of the function is $\mathcal{O}(b)$. Finally, Kuhn and Dumontier [184] canonicalize an online available RDF graph by transforming all its blank nodes to URIs. In order to prohibit name clashes with similar URIs stored in other graphs, all created URIs are prefixed with the same string based on the graph's designated web address. As this renaming process does not require sorting, it can be implemented with a runtime complexity of $\mathcal{O}(n)$. To achieve a consistent renaming of blank nodes, the canonicalization function requires a list of already transformed blank nodes and their respective URIs. This list contains at most b entries, resulting in a space complexity of $\mathcal{O}(b)$.

4.1.3. Serialization Functions for Graphs

A *serialization function* ν_N transforms an RDF graph into a sequential representation such as a set of bit strings. A formal definition of serialization functions is given in Section 4.3.4. The sequential representation is encoded in a specific format such as triple-based N-Triples [25] and Turtle [27] or XML-based RDF/XML [26] and OWL/XML [210]. TriX [75] and TriG [38] are formats for expressing Named Graphs. While TriX is based on XML, TriG has a triple-based syntax built upon Turtle. Notation3 (N3) [34] is another superset of Turtle for expressing RDF graphs as well as RDF rules. When signing RDF graphs, triple-based formats are often preferred to XML-based notations due to their simpler structure. If a serialization function processes each triple in the graph individually, it can be implemented with a runtime complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(1)$. Some canonicalization functions such as [72] directly operate on a serialized graph. Such canonicalization functions also include a serialization function and output a canonical serialization of the graph.

4.1.4. Hash Functions for Graphs

Applying a *hash function for graphs* λ_N is often based on computing the hash values of the graph's triples and combining them into a single value. Computing a triple's hash value can be done by using a basic hash function λ such as MD5 [250] or SHA-2 [218]. A formalization of basic hash functions and hash functions for graphs functions is provided

in Section 4.3.5. Melnik [200] uses a simple hash function for RDF graphs. A triple's hash value is computed by concatenating the hash value of its subject, predicate, and object and hashing the result. The hash values of all triples are sorted, concatenated, and hashed again to form the hash value of the entire RDF graph. Using a sorting algorithm like merge sort, the function's runtime complexity is $\mathcal{O}(n \log n)$ and its space complexity is $\mathcal{O}(n)$. Carroll [72] uses a graph-hashing function which sorts all triples, concatenates the result, and hashes the resulting bit string using a basic hash function such as SHA-2 [218]. As the function uses a sorting algorithm with a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$, the runtime complexity and the space complexity of the canonicalization function are the same. Fisteus et al. [110] suggest a hash function for N3 [34] datasets. The triples' hash values are computed with the canonicalization function of the same authors described in Section 4.1.2. The hash value of a graph is computed by incrementally multiplying the hash values of its triples modulo a large prime. Since this operation is commutative, sorting the triples' hash values is not required. Thus, the runtime complexity of the hash function is $\mathcal{O}(n)$. Due to the incremental multiplication, the space complexity is $\mathcal{O}(1)$. Finally, Sayers and Karp [261] compute a hash value of an RDF graph similar to the approach of Fisteus et al. First, the triples are serialized as single bit string and then hashed. Second, the incremental multiplication is conducted. Thus, the runtime complexity of this approach is $\mathcal{O}(n)$ and the space complexity is $\mathcal{O}(1)$.

4.1.5. Signature Functions

A *signature function* φ computes the actual graph signature by combining the graph's hash value with a secret signature key. Signature functions are formalized in Section 4.3.7. Examples of existing signature functions are DSA [214], RSA [251], and ElGamal [97]. Since the graph's hash value is independent from the number of triples, the signature is as well. Thus, the runtime complexity and the space complexity of all signature functions are $\mathcal{O}(1)$.

4.1.6. Assembly Function

An *assembly function* α_N creates a detailed description of how a graph's signature can be verified. This description may then be added to the signed graph data or be stored at a separate location. Section 4.3.8 provides a formal definition of assembly functions. Tummarello et al. [295] present a simple assembly function which adds additional triples to each signed MSG containing the signature value and a URL to the public key used for verifying the signature value. Information about the graph signing function and its sub-functions is not provided. Once the URL to the signature verification key is broken, i. e., the key is not available anymore at this URL, the signature can no longer be verified. Even if a copy of the verification key is still available at a different location, the verifier has no proof that this copy is really a legitimate copy of the original public key. Since the issuer and other identifying metadata of the signature verification key are not provided by the triples of the assembly function, the verifier cannot check the true

authenticity of the copied key. The assembly function described in the XML signature standard [20] provides an XML schema for covering all information about the signature's creation and verification. This includes the names of the used canonicalization function, the hash function, and the signature function used for computing the signature value. The XML schema also provides a unique identifier of the signature key issuer, the key's serial number, and further information.

4.1.7. Alternative Approaches for Achieving Integrity of Graph Data

RDF graph data is usually stored at a data provider located in the web [39]. In order to access the graph data, a recipient has to first download it from such a provider. Thus, ensuring integrity of RDF graph data can also be accomplished by securing this download process. The Transport Layer Security (TLS) protocol [92] and its predecessor, the Secure Sockets Layer (SSL) protocol [113], can be used to create a secure communication channel between a recipient and a data provider. Such a secure communication channel can be used for achieving both integrity and authenticity of the transmitted graph data. When the recipient downloads the graph from the data provider, the communication channel is digitally signed by the data provider. Any unauthorized modification of the data during its transmission can be identified by the recipient. Furthermore, the digital signature also associates the transmitted graph with the data provider, achieving authenticity of the graph data. However, secure communication channels based on TLS or SSL only protect the transmitted data during the transmission process. They cannot be used for permanently achieving integrity and authenticity of the data. Once a secure communication channel is closed after the graph data has been transmitted, the recipient cannot verify the graph data again without retrieving it once more from the data provider. Additional verifications might be necessary if the data is stored on untrusted devices such as cloud storage services or when it is transmitted further to other parties which need to verify the graph data themselves. Furthermore, secure communication channels require the data provider to be completely trusted. If the data provider is different from the graph's original creator, it may not be considered as completely trustworthy. In this case, the data provider can modify the graph at any time and therefore violate the graph's integrity. The recipient of the graph cannot notice such modifications as the secure communication channel originates from the data provider and not from the graph's creator.

Kuhn and Dumontier [184] present trusty URIs, an approach for achieving integrity of RDF graphs without digitally signing them. A trusty URI is a web address of an RDF graph which contains the graph's cryptographic hash value. After having downloaded a graph, the recipient computes the graph's hash value again and compares it with the hash value encoded into the trusty URI. If both hash values are identical, the graph's integrity is confirmed. Publishing a graph on the web requires the data provider to first compute the graph's hash value. In order to remove the influence of the graph's blank nodes on the computed hash value, all blank nodes are transformed into preliminary URIs by using the canonicalization function of the same authors described in Section 4.1.2. In this case, the prefix used for the preliminary URIs is the graph's designated web address

without its hash value. After having transformed the blank nodes, the graph's hash value is computed by using the hash function of Carroll [72] as described in Section 4.1.4. Finally, the computed hash value is used for creating the trusty URI of the graph. The security of trusty URIs is based on the connection between a graph's hash value and its web address. Any modifications of a graph can be identified as the hash value of the modified graph differs from the hash value encoded into the graph's web address. However, trusty URIs are only capable of protecting the integrity of a graph if the graph is downloaded from the web. After this process has been completed, verifying the graph's integrity is no longer possible without having to download the graph again. Furthermore, trusty URIs do not create a connection between a data provider and the provided graph data. Thus, trusty URIs are not able to protect a graph's authenticity and are vulnerable to spoofing attacks in which an attacker masquerades as the data owner. An attacker may fake a trusty URI for a self-created graph which contains the web address of the data owner, thereby claiming that this data owner has published the graph. The attacker then publishes the trusty URI on the web. If a recipient wants to download the graph from the data owner, the attacker uses a spoofing attack such as IP spoofing [291] or ARP cache poisoning [1] to redirect and intercept the communication. The attacker then sends the self-created graph to the recipient, pretending that the recipient is actually communicating with the data provider. As trusty URIs do not use any mechanisms for achieving the graph's authenticity, a recipient cannot verify the identity of the respective communication partner and thus cannot distinguish between an attacker and a legitimate data owner. Such attacks can be prohibited by establishing a secure communication channel based on a protocol such as TLS or SSL. However, as such connections also achieve integrity of the transmitted data, the use of trusty URIs would become obsolete in this case.

4.2. Requirements for a Graph Signing Framework

The graph signing framework Siggi formally defines a process for signing arbitrary graph data. Due to the framework's flexible design, each step in this process can be implemented by a different algorithm. Based on these general objectives, this section defines the functional (**RI.F.***) and non-functional (**RI.N.***) requirements for the graph signing framework Siggi. As defined in Section 3.2, functional requirements define the services and functions that a system must provide [282]. On the other hand, non-functional requirements define the overall design of a system and describe how functional requirements are implemented. The following requirements are based on the scenario for regulating Internet communication presented in Section 2.1 and on the related work for signing graph data summarized in Section 4.1. The graph signing framework Siggi must support the following functional requirements:

RI.F.1: Signing different types of graph data

The graph signing framework must allow a party to sign different types of graph data such as RDF(S) graphs (**RI.F.1.1**), OWL graphs (**RI.F.1.2**), and Named

Graphs (**RI.F.1.3**). This allows the framework to be used in heterogeneous environments such as Linked Open Data [39]. In the workflow for exchanging regulation policies introduced in the scenario in Section 2.1.2, the BKA provides OWL ontology design patterns, ContentWatch provides Named Graphs, and JusProg provides its regulation data as RDF graphs.

RI.F.2: Signing at different levels of granularity
The graph signing framework must support signing graph data at different levels of granularity including single triples (**RI.F.2.1**), arbitrary sets of triples (**RI.F.2.2**), MSGs (**RI.F.2.3**), and entire graphs (**RI.F.2.4**). This results in a most flexible use of the framework. In the scenario, the BKA signs ontology design patterns and the German Telecom signs its entire regulation graph.

RI.F.3: Signing T-box and A-box knowledge
The graph signing framework must allow a party to sign both assertional (A-box) knowledge (**RI.F.3.2**) and terminological (T-box) knowledge (**RI.F.3.1**). A-box knowledge corresponds to factual knowledge whereas T-box knowledge represents the knowledge encoded in the ontological model, i. e., the schema knowledge [152]. This enables the framework to be used for signing vocabularies issued by, e. g., standardization bodies. In addition, parties publishing their own instance data using those vocabularies can sign their assertional knowledge as well. In the scenario, the BKA signs both its ontologies and its technical regulation details.

RI.F.4: Signing graph data iteratively
The graph signing framework must support signing graph data which is already signed, i. e., iterative signing of graph data. An iterative signature may cover signed graph data from a previous signing step as well as newly added graph data. Signing already signed graph data again without adding any additional triples can be used for countersigning the data. Other applications of iterative signing are provenance tracking where each party receives signed data from a party, sings the data again, and sends the result to another party. This allows it to re-create the flow of the signed data. In the scenario, the German Telecom receives signed graph data from the BKA and signs it again in order to track the provenance of the exchanged regulation details.

RI.F.5: Signing multiple and distributed graphs
The graph signing framework must allow a party to sign multiple graphs at the same time which are distributed over different locations. In the scenario, the German comprehensive school retrieves two different graphs from the ContentWatch and JusProg and signs both graphs at once.

In addition to these functional requirements, the graph signing framework Siggi must also support the following non-functional requirements:

RI.N.1: Creating encoding-independent signatures
The signatures created with the graph signing framework must not rely on a particular encoding or serialization of the graph. It must be possible to modify the

graph's syntactical representation after having signed the graph without invalidating the created signature. For example, it must be possible to transform a signed graph stored as an RDF/XML document into an N-Triples file while still being able to verify the graph's signature. Changing the order of the triples of a signed graph or renaming its blank node identifiers must also be possible without destroying the graph's signature. As these two modifications do not influence the graph's semantics, they are also considered to be plain syntactical modifications and are subsumed by this requirement as well.

RI.N.2: Supporting flexible configurability
The graph signing framework must support different configurations for signing graph data. A configuration implements each step of the general signing process depicted in Figure 4.1 with a particular algorithm. Possible configurations of the framework must at least cover already existing approaches for signing graph data as discussed in the related work in Section 4.1. Furthermore, the design of the framework should also support future configurations.

RI.N.3: Supporting a modular design
The graph signing framework must have a modular design which allows to combine different sub-functions for signing graphs as one framework configuration. The framework must support interchangeable implementations for each sub-function and allow the creation of new configurations by using these sub-functions as building blocks. For example, it must be possible to use the canonicalization function of one author and combine it with a hash function for graphs from another author.

RI.N.4: Separating the signing process from software implementations
The graph signing framework must not rely on a particular software implementation or programming language. Instead, it must be possible to sign a graph with a particular software implementation and verify the created signature with another implementation.

Integrity and authenticity of graph data are not listed as specific requirements for the graph signing framework Siggi. Instead, integrity and authenticity of graph data correspond to the research questions **RQ.2** and **RQ.3**, respectively, which are answered by the graph signing framework Siggi. The fulfillment of all functional and non-functional requirements by the the graph signing framework Siggi is summarized in Section 4.3.10 and a comparison with the related work is given in Section 4.8.

4.3. Formalization of the Graph Signing Framework Siggi

Based on the state of the art and the related work discussed in Section 4.1 as well as on the requirements defined in Section 4.2, this section provides a formal specification of the graph signing framework Siggi. This specification covers the input and output of all functions which are part of the graph signing process as depicted in Figure 4.1. The specification can be used as a guideline to create new functions for each step in the process.

As the formal specification is independent from any particular software implementation, the created signatures can be verified using any available implementation.

4.3.1. Definition of Graphs

An RDF graph G is an unordered, finite set of RDF triples t. The set of all RDF triples is defined as follows with the pairwise disjoint sets of resource URIs $\mathbb{R}$, blank nodes $\mathbb{B}$, and literals $\mathbb{L}$[1]:

$$\mathbb{T} := (\mathbb{R} \cup \mathbb{B}) \times \mathbb{R} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}) \tag{4.1}$$

It is $t = (s, p, o)$ with $s \in \mathbb{R} \cup \mathbb{B}$ being the subject of the triple, $p \in \mathbb{R}$ being the predicate, and $o \in \mathbb{R} \cup \mathbb{B} \cup \mathbb{L}$ being the object [13]. An OWL graph can be mapped to an RDF graph [230]. Thus, in the following only RDF graphs are denoted while OWL graphs are included by mapping them to RDF graphs. The set of all possible RDF graphs is defined as follows:

$$\mathbb{G} := \mathcal{P}(\mathbb{T}) = \mathcal{P}((\mathbb{R} \cup \mathbb{B}) \times \mathbb{R} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L})) \tag{4.2}$$

A Named Graph extends the notion of RDF graphs and associates a unique name in form of a URI to a single RDF graph [73] or set of RDF graphs. This URI can be described by further triples, which form the so-called *annotation graph*. Consequently, the original RDF graph is also called the *content graph*. A Named Graph $NG \in \mathbb{G}_N$ is defined as $NG = (a, A, \{C_1, C_2, \ldots, C_l\})$ with $a \in \mathbb{R} \cup \{\varepsilon\}$ being the name of the graph, $A \in \mathbb{G}$ being the annotation graph, and $C_i \in \mathbb{G}_N$ being content graphs with $i = 1 \ldots l$ and $l \in \mathbb{N}$. If a Named Graph does not explicitly specify an identifier, ε is used as its name. This corresponds to associating a blank node with the graph. In this case, the annotation graph A is empty, i. e., $A = \emptyset$. Any RDF graph $G \in \mathbb{G}$ can be defined as Named Graph C using the notation above as $C = (\varepsilon, \emptyset, G)$. The set of all Named Graphs $\mathbb{G}_N$ is recursively defined as follows:

$$\mathbb{G}_N := (\mathbb{R} \times \mathbb{G} \times \mathcal{P}(\mathbb{G}_N)) \cup (\{\varepsilon\} \times \{\emptyset\} \times \mathbb{G}) \tag{4.3}$$

4.3.2. Graph Signing Function σ_N

The *graph signing function* σ_N consists of several sub-functions and creates a signature value s of a set of Named Graphs. These sub-functions are the canonicalization function κ_N, the serialization function ν_N, the hash function for graphs λ_N, the combining function for graphs ϱ_N, and the signature function φ, which are formally defined below. The combining function for graphs ϱ_N allows the graph signing function to sign multiple graphs at once. The assembly function α_N and the verification function γ_N are not part of the graph signing function σ_N and are applied afterwards. The graph signing function σ_N requires a secret key $k_s \in \mathbb{K}_s$ as input with $\mathbb{K}_s$ being the set of all secret keys. Additionally, the function requires a set of m Named Graphs NG with

[1] Please note that this thesis uses the symbol $\mathbb{R}$ for representing resources and not real numbers. The set of real numbers is not used at all in this work.

$NG_i = (a_i, A_i, \{C_{1_i}, \ldots, C_{l_i}\})$, $i = 1, \ldots, m$, and $m \in \mathbb{N}$. The resulting signature value s is a bit string of length $d' \in \mathbb{N}$, i. e., $s \in \{0,1\}^{d'}$. The graph signing function σ_N is defined as follows:

$$\sigma_N : \mathbb{K}_s \times \mathcal{P}(\mathbb{G}_N) \to \{0,1\}^{d'}, \quad \sigma_N(k_s, \{NG_1, \ldots, NG_m\}) := s \tag{4.4}$$

$$s := \varphi(k_s, \varrho_N(\lambda_N(\nu_N(\kappa_N(NG_1))), \ldots, \lambda_N(\nu_N(\kappa_N(NG_m))))) \tag{4.5}$$

The graph signing function σ_N first canonicalizes all Named Graphs using the canonicalization function κ_N. Each canonicalized graph is then serialized using the serialization function ν_N and transformed into a bit string using the hash function for graphs λ_N. The combining function for graphs ϱ_N is applied to these bit strings in order to create a single bit string which can then be signed with the signature function φ.

4.3.3. Canonicalization Function for Graphs κ_N

The *canonicalization function* κ transforms a graph $G \in \mathbb{G}$ into its canonical form $\hat{G} \in \hat{\mathbb{G}}$ with $\hat{\mathbb{G}} \subset \mathbb{G}$ being the set of all canonical graphs. Example implementations of the function κ are discussed in Section 4.1.2. The function is defined as follows:

$$\kappa : \mathbb{G} \to \hat{\mathbb{G}}, \quad \kappa(G) := \hat{G} \tag{4.6}$$

For Named Graphs, the canonicalization function κ_N is recursively defined. It computes a canonical representation of a Named Graph $NG = (a, A, \{C_1, \ldots, C_l\})$ by computing the canonical representations $\hat{A}$ and $\hat{C}_i$ of its annotation graph A and its content graphs C_i with $i = 1, \ldots, l$ and $l \in \mathbb{N}$. The result is a canonical representation $\hat{NG} \in \hat{\mathbb{G}}_N$ with $\hat{\mathbb{G}}_N \subset \mathbb{G}_N$ being the set of all canonical Named Graphs. Using the function κ, the canonicalization function κ_N is defined as follows:

$$\kappa_N : \mathbb{G}_N \to \hat{\mathbb{G}}_N, \quad \kappa_N(NG) := \hat{NG} \tag{4.7}$$

$$\hat{NG} := \begin{cases} (\varepsilon, \emptyset, \kappa(G)) & \text{if } NG = (\varepsilon, \emptyset, G),\ G \in \mathbb{G} \\ (a, \kappa(A), \{\kappa_N(C_1), \ldots, \kappa_N(C_l)\}) & \text{if } NG = (a, A, \{C_1, \ldots, C_l\}) \end{cases} \tag{4.8}$$

4.3.4. Serialization Function ν_N

The *serialization function* ν transforms a graph $G \in \mathbb{G}$ into a set $\overline{G}$ of bit strings $b \in \{0,1\}^*$ with $\overline{G} \in \mathcal{P}(\{0,1\}^*)$. A single bit string usually represents a triple in the graph G. The concrete characteristics of the bit strings in $\overline{G}$ depend on the used serialization format. Example formats of the serialization function ν are discussed in Section 4.3.4. The serialization function ν is defined as follows:

$$\nu : \mathbb{G} \to \mathcal{P}(\{0,1\}^*), \quad \nu(G) := \overline{G} \tag{4.9}$$

The serialization function ν can be extended to the serialization function ν_N for Named Graphs $NG \in \mathbb{G}_N$. The result of the function ν_N is a set $\overline{NG}$ of o bit strings $b_i \in \{0,1\}^*$ with $\overline{NG} = \{b_1, b_2, \ldots, b_o\}$, $i = 1, \ldots, o$, and $o \in \mathbb{N}$. The particular value of o depends

on the used implementation of the serialization function ν_N and on the serialized Named Graph NG. Using the function ν, the serialization function ν_N is recursively defined as follows:

$$\nu_N : \mathbb{G}_N \to \mathcal{P}(\{0,1\}^*), \quad \nu_N(NG) := \overline{NG} \tag{4.10}$$

$$\overline{NG} := \begin{cases} \nu(G) & \text{if } NG = (\varepsilon, \emptyset, G),\ G \in \mathbb{G} \\ \{a\} \cup \nu(A) \cup \nu_N(C_1) \cup \ldots \cup \nu_N(C_l) & \text{if } NG = (a, A, \{C_1, \ldots, C_l\}) \end{cases} \tag{4.11}$$

4.3.5. Hash Function for Graphs λ_N

The *basic hash function* λ computes a hash value h of an arbitrary bit string $b \in \{0,1\}^*$. The resulting hash value h has a fixed length $d \in \mathbb{N}$, i. e., $h \in \{0,1\}^d$. The function λ is defined as follows:

$$\lambda : \{0,1\}^* \to \{0,1\}^d, \quad \lambda(b) := h \tag{4.12}$$

The *hash function for graphs* λ_N computes a hash value $h_N \in \{0,1\}^d$ of a set of bit strings $b \in \{0,1\}^*$. It is built upon the basic hash function λ. The function λ_N directly operates on bit strings and can be used for computing hash values of graphs $G \in \mathbb{G}$ and hash values of Named Graps $NG \in \mathbb{G}_N$. Example implementations of the basic hash function λ and the hash function for graphs λ_N are presented in Section 4.3.5. The hash function for graphs λ_N is defined as follows:

$$\lambda_N : \mathcal{P}(\{0,1\}^*) \to \{0,1\}^d, \quad \lambda_N(\overline{NG}) := h_N \tag{4.13}$$

4.3.6. Combining Function for Graphs ϱ_N

The *combining function for graphs* ϱ_N combines a set of hash values $h \in \{0,1\}^d$ of equal length $d \in \mathbb{N}$ into a single hash value $h_M \in \{0,1\}^d$. The function ϱ_N allows the graph signing function σ_N to sign multiple graphs at once. Example combining functions of graphs ϱ_N are discussed in [261]. The combining function for graphs ϱ_N is defined as follows:

$$\varrho_N : \mathcal{P}(\{0,1\}^d) \to \{0,1\}^d, \quad \varrho_N(\{h_1, h_2, \ldots\}) := h_M \tag{4.14}$$

4.3.7. Signature Function φ

A *signature function* φ computes the signature value of a set of graphs based on the set's hash value $h_N \in \{0,1\}^d$ and a cryptographic key. The keyspace, i. e., the set of all asymmetric, cryptographic keys is defined as $\mathbb{K}_a = \mathbb{K}_p \times \mathbb{K}_s$ with $\mathbb{K}_p$ as the set of public keys and $\mathbb{K}_s$ as the set of secret keys. For computing signatures, a secret key $k_s \in \mathbb{K}_s$ is used. Possible implementations for the signature function φ are presented in Section 4.1.5. Using $s \in \{0,1\}^{d'}$ as identifier for the resulting bit string, the signature function is defined as follows:

$$\varphi : \mathbb{K}_s \times \{0,1\}^d \to \{0,1\}^{d'}, \quad \varphi(k_s, b) := s \tag{4.15}$$

4.3.8. Assembly Function α_N

The *assembly function* α_N is applied after the graph signing function σ_N. It creates a signature graph $S \in \mathbb{G}$ and includes it in a new Named Graph $NG_S \in \mathbb{G}_N$. The content and structure of S depend on the implementation of the assembly function α_N. The graph S provides information about how to verify the signature of a set of a graphs. This includes all sub-functions of the graph signing function σ_N, the public key k_p of the used secret key k_s, the identifiers a_i of the signed Named Graphs, and the signature value s. A possible structure of a signature graph is shown in the examples in Section 4.7. Additional examples of an assembly function are presented in Section 4.1.6. The Named Graph NG_S contains the signature graph S as its annotation graph and the signed graphs NG_i with $i = 1, \ldots, m$ and $m \in \mathbb{N}$ as its content graphs. In order to support iterative signing of Named Graphs, the result of the assembly function α_N is also a Named Graph. The function is defined as follows:

$$\alpha_N : \mathbb{K}_p \times \{0,1\}^{d'} \times \mathcal{P}(\mathbb{G}_N) \to \mathbb{G}_N, \quad \alpha_N(k_p, s, \{NG_1, \ldots, NG_m\}) := NG_S \tag{4.16}$$

$$NG_S := (a_S, S, \{NG_1, \ldots, NG_m\}) \tag{4.17}$$

4.3.9. Verification Function γ_N

The verification of a signature is similar to its creation. A *verification function* γ_N requires a canonicalization function κ_N, a serialization function ν_N, and a hash function λ_N for graphs. It also requires a signature verification function δ_N as inverse of the signature function φ. The function δ_N requires a bit string $s \in \{0,1\}^{d'}$ and a public key $k_p \in \mathbb{K}_p$ as input. It is defined as follows with $b \in \{0,1\}^d$ being the resulting bit string. If a secret key $k_s \in \mathbb{K}_s$ is inverse to the public key k_p, it holds $\delta_N(k_p, \varphi(k_s, b)) = b$ for all $b \in \{0,1\}^d$.

$$\delta_N : \mathbb{K}_p \times \{0,1\}^{d'} \to \{0,1\}^d, \quad \delta_N(k_p, s) := b \tag{4.18}$$

The verification function γ_N checks whether or not a given signature is a valid signature of a set of Named Graphs. The function requires a public key k_p and a signature value s which can be taken from the signature graph S. Additionally, the function requires a set of signed Named Graphs $\{NG_1, \ldots, NG_m\}$ with $m \in \mathbb{N}$. The key k_p is the public counterpart of the secret key k_s, which was used for creating the signature value s. The verification function γ_N combines the signature value s with the public key k_p and computes the hash value h' of the Named Graphs NG_i with $i = 1, \ldots, m$. The signature is valid iff both computed values are equal. Using $h' = \lambda_N(\nu_N(\kappa_N(NG_1)) \cup \ldots \cup \nu_N(\kappa_N(NG_m)))$, the verification function γ_N is defined as follows:

$$\gamma_N : \mathbb{K}_p \times \mathcal{P}(\mathbb{G}_N) \times \{0,1\}^* \to \{TRUE, FALSE\} \tag{4.19}$$

$$\gamma_N(k_p, \{NG_1, \ldots, NG_m\}, s) := \begin{cases} TRUE & \text{if } \delta_N(k_p, s) = h' \\ FALSE & \text{otherwise} \end{cases} \tag{4.20}$$

4.3.10. Fulfillment of the Requirements

This section outlines how the functional and non-functional requirements defined in Section 4.2 are fulfilled by the graph signing framework Siggi. A further discussion and comparison with the related work is provided in Section 4.8. The requirement of signing different types of graph data (**RI.F.1**) is supported via the definition of Named Graphs given in Equation 4.3. This definition allows to map RDF(S) graphs (**RI.F.1.1**) and OWL graphs (**RI.F.1.2**) to Named Graphs by using a representation like $(\varepsilon, \emptyset, G)$. In this case, ε is used as the graph's name, $\emptyset$ is used as the annotation graph, and $G \in \mathbb{G}$ corresponds to the original graph. Signing Named Graphs (**RI.F.1.3**) is directly supported by the graph signing function σ_N as defined in Equation 4.4. Thus, this function can also be used to sign RDF(S) graphs and OWL graphs. The fulfillment of the requirement of signing graph data at different levels of granularity (**RI.F.2**) depends on the particular granularity level. Signing a complete graph (**RI.F.2.4**) is directly supported by the graph signing function σ_N. Signing individual triples (**RI.F.2.1**), arbitrary sets of triples (**RI.F.2.2**), or MSGs (**RI.F.2.3**) is achieved by first creating a new graph which contains all triples to be signed. The resulting graph can then be transformed into a Named Graph by applying Equation 4.3 and signed using the graph signing function φ. T-box knowledge and A-box knowledge (**RI.F.3**) share the same syntactical representation which consists of a set of triples. A Named Graph can contain any type of triples and does not distinguish between schema knowledge and factual knowledge. Thus, signing T-box knowledge (**RI.F.3.1**) and A-box knowledge (**RI.F.3.2**) is equally supported by the graph signing function σ_N. The requirement of iterative signing of graph data (**RI.F.4**) is fulfilled by the design of the assembly function α_N defined in Equation 4.16. The output of the function is a Named Graph which can be signed again using the graph signing function σ_N. Signing multiple graphs at once (**RI.F.5**) is supported by the combining function for graphs ϱ_N defined in Equation 4.14. The function combines the hash values of multiple graphs into a single hash value. Singing this hash value corresponds to signing these graphs at the same time. As the function does not distinguish between local graphs and remote graphs, it equally supports both types of graphs.

The requirement of an encoding-independent signature (**RI.N.1**) is collectively fulfilled by the canonicalization function for graphs κ_N, the hash function for graphs λ_N, and the design of the framework's formal specification. The canonicalization function for graphs κ_N ensures that renaming a graph's blank node identifiers does not invalidate the graph's signature. The hash function for graphs λ_N normalizes the order of the graph's triples in such a way that changing this order does not influence the signature. The framework's formal specification does not rely on a particular encoding of the graph and interprets the graph as an abstract data structure which is processed as such. As the encoding format used by the assembly function α_N is independent of the signing process, it can be changed without invalidating the graph's signature. The requirement of different configurations of the graph signing framework (**RI.N.2**) is supported by the generic design of the framework's formal specification. The specification only defines the basic characteristics of the signing process and focuses on the input and output data of

each signing step. As it does not rely on a particular algorithm, it can be configured with different algorithms from different authors. Distinguishing between the different steps of the signing process also fulfills the requirement of a modular design (**RI.N.3**). Finally, the formal specification is independent from any particular software implementation and thus fulfills requirement **RI.N.4**. Signing a graph and verifying a graph's signature can be done using different software implementations as long as both implementations comply with the framework's formal specification.

4.4. Four Configurations of the Signing Framework

This section discusses four possible configurations A, B, C, and D of the graph signing framework Siggi. The example configurations are extracted from the related work described in Section 4.1 and are referred to by the names of their respective authors. New configurations can also be created by combining different algorithms from different authors or by creating new algorithms from scratch. Each configuration of the graph signing framework must define the particular implementation of the assembly function α_N and of all sub-functions of the graph signing function σ_N. However, the formal specification provided in Section 4.3 already defines the concrete operation of the canonicalization function κ_N for Named Graphs, the serialization function ν_N for Named Graphs, and the graph signing function σ_N. Thus, a particular configuration of the framework must only define those functions of the signing process whose implementation details are not covered by the formal specification. These functions are the canonicalization function κ, the serialization function ν, the basic hash function λ, the hash function for graphs λ_N, the combining function for graphs ϱ_N, and the assembly function α_N.

Table 4.2.: Basic implementation of the four example configurations of the the graph signing framework. To ease comparability, the four configurations mostly use the same sub-functions and only differ in the canonicalization function κ, the hash function for graphs λ_N, and in the assembly function α_N.

Function	**Implementation**
Graph signing function σ_N	as defined in Equation 4.4
Canonicalization function κ	depending on the configuration
Canonicalization function κ_N for Named Graphs	as defined in Equation 4.7
Serialization function ν	N-Triples [25]
Serialization function ν_N for Named Graphs	as defined in Equation 4.10
Basic hash function λ	SHA-2 [218]
Hash function for graphs λ_N	depending on the configuration
Combining function for graphs ϱ_N	as defined in Equation 4.21
Signature function φ	RSA [251]
Assembly function α_N	depending on the configuration

In order to ease the comparability between the four example configurations, the configurations only differ in their used canonicalization function κ, hash function for graphs λ_N, and assembly function α_N. All configurations use N-Triples [25] for the serialization function ν, SHA-2 [218] as basic hash function λ, and RSA [251] as signature function φ. As for the combining function for graphs ϱ_N, all four configurations use a function based on the hash function for graphs of Carroll [72]. The used function combines the hash values of all graphs to be signed by sorting them, concatenating them, and hashing the resulting string using a basic hash function λ. SHA-2 is also used as the basic hash function λ in the combining function ϱ_N. Using sort as sorting function and concat as concatenation function, the definition of the combining function for graphs ϱ_N provided in Equation 4.14 is refined as follows:

$$\varrho_N : \mathcal{P}(\{0,1\}^d) \to \{0,1\}^d, \quad \varrho_N(\{h_1, h_2, \ldots\}) := h_M \tag{4.14}$$

$$\varrho_N(\{h_1, h_2, \ldots\}) := \lambda(\mathsf{concat}(\mathsf{sort}(\{h_1, h_2, \ldots\}))) \tag{4.21}$$

Table 4.2 summarizes the similarities and differences of the four example configurations. Each of the four configurations is further analyzed regarding its runtime complexity, space complexity, and signature overhead when signing a single graph. The runtime complexity and space complexity depend on the characteristics of the graph to be signed as well as on the graph signing function σ_N and its sub-functions. The signature overhead depends on the additional triples created by these sub-functions and on the size of the signature graph S created by the assembly function α_N. Table 4.3 summarizes the complexity and signature overhead of the example configurations for signing a single graph.

Table 4.3.: Configurations A–D of the graph signing framework with runtime complexity, space complexity, and signature overhead for singing a single graph. n is the number of triples in the graph, b is the number of blank nodes, b_h is the number of blank nodes which require special treatment, and r is the number of disjoint MSGs in the graph.

Configuration	**Complexity of σ_N**		**Signature overhead**
	runtime	**space**	**of σ_N and α_N**
A) Carroll [72]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$b_h + 25$ triples, $b_h \leq b$
B) Tummarello et al. [295]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$b_h + 6r$ triples, $b_h \leq b$, $r \leq n$
C) Fisteus et al. [110]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$0 + 25$ triples
D) Sayers & Karp [261]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$b + 25$ triples

4.4.1. Configuration A: Carroll

Configuration A is based on the canonicalization function κ and the hash function for graphs λ_N of Carroll [72]. Due to their use of a sorting operation, both functions have

a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$ with n being the number of triples in the processed graph. The resulting graph signing function σ_N built upon these functions shares the same complexity. The canonicalization function κ handles blank node identifiers by sorting all of a graph's triples and creating additional triples for blank nodes sharing the same identifier. With b as the total number of blank nodes in the graph and $b_h \leq b$ as the number of blank nodes which require an additional triple, the canonicalized graph contains b_h more triples than the original graph. Configuration A uses an assembly function α_N which creates a signature graph S based on the signature ontology provided in Appendix C. This signature graph contains 25 triples which results in a total signature overhead consisting of $b_h + 25$ triples.

4.4.2. Configuration B: Tummarello et al.

Configuration B corresponds to the approach by Tummarello et al. [295] and is based on the canonicalization function κ and hash function for graphs λ_N of Carroll [72]. Although both functions are also used in configuration A, configuration B only allows to sign individual MSGs. Signing a complete graph with n triples and r disjoint MSGs requires the graph to be split into its MSGs first. Splitting the graph can be done with a runtime complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(n)$ by using an implementation based on bucket sort [83] where each MSG corresponds to one bucket. Each MSG is then signed individually using Carroll's functions. Signing a complete graph results in a runtime complexity of $O\left(\sum_{i=1}^{r} n_i \log n_i\right)$ and a space complexity of $O\left(\sum_{i=1}^{r} n_i\right)$ with n_i being the number of triples in MSG i. Since all MSGs are disjoint, it is $\sum_{i=1}^{r} n_i = n$. Thus, the total runtime complexity of configuration B's graph signing function σ_N is $\mathcal{O}(n \log n)$ and its space complexity is $\mathcal{O}(n)$. The assembly function α_N of Tummarello et al. stores a signature using six additional triples. Since each MSG of the graph is signed individually, the assembly function creates a separate set of signature triples for each MSG. Thus, the overhead created by the assembly function α_N for a graph with r MSGs is $6r$ triples. Combining these triples with the b_h triples created by Carroll's canonicalization function κ results in a total overhead of $b_h + 6r$ triples.

4.4.3. Configuration C: Fisteus et al.

Configuration C uses the canonicalization function κ and hash function for graphs λ_N of Fisteus et al. [110]. Combining these two functions results in a configuration with minimum signature overhead. Due to its use of a sorting algorithm, the canonicalization function κ has a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$ when canonicalizing a graph with n triples. Since the hash function for graphs λ_N operates incrementally on the triples of the graph and does not require sorting, its has a runtime complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(1)$. This results in a runtime complexity of the graph signing function σ_N of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$. As the functions of Fisteus et al. do not create any additional triples, the signature overhead is independent of the signed graph and only depends on the signature graph S. Using the

same assembly function α_N as configuration A which creates a signature graph S with 25 triples results in a total signature overhead of 25 triples.

4.4.4. Configuration D: Sayers and Karp

Configuration D is based on the canonicalization function κ and hash function for graphs λ_N by Sayers and Karp [261]. Signing a graph with n triples and b blank nodes using this configuration results in a minimum runtime complexity of $\mathcal{O}(n)$ of the graph signing function σ_N. The canonicalization function κ creates an additional triple for each blank node in the graph. In order to detect already handled blank nodes, this function maintains a list of additional triples created so far. This list contains at most b entries with b being the total number of additional triples. Blank nodes can only occur on subject and/or object position in a triple. Thus, each triple of a graph contains no, one, or two blank nodes. Assuming that each blank node is part of at least one triple, a graph can contain at most twice as many blank nodes as triples, i. e., $b \leq 2n$. This results in a space complexity of $\mathcal{O}(n)$ of the graph signing function σ_N. Using the same assembly function as configuration A and C, the signature overhead of configuration D consists of b triples added by the canonicalization function κ and 25 triples created by the assembly function α_N.

4.5. Cryptanalysis of the Four Configurations

The cryptographic security of a particular configuration of the graph signing framework Siggi depends on the cryptographic security of the graph signing function σ_N and its used sub-functions. A comprehensive cryptanalysis of a graph signing function σ_N must therefore cover all algorithms used for implementing these sub-functions. This section first introduces an attack model which covers different attacks on signing graph data. This attack model is then used as a foundation for analyzing the cryptographic security of the four example configurations A, B, C, and D presented in Section 4.4 for signing a set of Named Graphs. The analysis first covers the common security aspects of all four example configurations and then discusses the particular security of each configuration in more detail. As the four example configurations mainly differ in their used canonicalization function κ and hash function for graphs λ_N, the specific cryptanalysis of each configuration focuses on these two functions.

4.5.1. Attack Model

The graph signing framework Siggi implements the two security requirements authenticity and integrity of a set of Named Graphs. Authenticity means that the party who claims to have signed the set of graphs is actually the signature's creator. Integrity means that the signed graph data was not modified by an unauthorized party after the signature had been created [28]. Modifications of the set of graphs are considered to be unauthorized if they modify the semantics of at least one graph in the set. As requirement **RI.N.1** implies, purely syntactical modifications of a graph such as consistent

renaming of blank nodes or re-ordering the triples in the graph do not modify the graph's semantics. Thus, such modifications are not considered to be unauthorized and are even necessary for canonicalizing and hashing a graph as defined in the formal specification in Section 4.3. All unauthorized modifications are summarized as the following attacks on a set of signed Named Graphs. These attacks also form the basis of the cryptanalysis provided in this section.

AI.1: Removing triples from a signed graph
An attacker removes at least one triple from a graph in the set. The resulting graph has fewer triples than the original graph. This violates the integrity of the modified graph and thus the integrity of the set of graphs as well.

AI.2: Inserting additional triples into a signed graph
An attacker inserts at least one new triple into a graph in the set. The resulting graph has more triples than the original graph. Again, this violates the integrity of the modified graph and thus the integrity of the set of graphs as well. Inserting an already existing triple of a graph into the same graph again is not considered as an attack. As RDF graphs do not contain any duplicate triples, inserting already existing triples does not modify the graph's semantics.

AI.3: Modifying existing triples in a signed graph
An attacker modifies at least one existing triple in a graph or replaces an existing triple in the graph with a new triple which is not part of the graph. Possible modifications include the substitution of a subject URI with a different URI. Modifying or replacing triples in a graph violate the graph's integrity and thus the integrity of the set as well.

AI.4: Removing a signed graph from the set
An attacker removes at least one graph from the set of signed graphs. The resulting set of graphs has fewer graphs than the original set. Thus, this attack violates the integrity of the set of graphs.

AI.5: Inserting a new graph into the set
An attacker inserts a new graph into the set of signed graphs. The resulting set of graphs has more graphs than the original set. Inserting an already existing graph into the set a second time is not considered as an attack as the set can contain a graph at most once. Again, this attack violates the integrity of the set of graphs.

AI.6: Forging signatures of a set of graphs
An attacker creates a signature for a set of graphs of own choice without having access to the secret signing key k_s. The forged signature is considered as valid by the verification function γ_N as the function uses the corresponding public key k_p of the secret key k_s for verification. In this attack, the real owner of the secret key k_s has not approved of the signed set of graphs. However, the verification function γ_N cannot distinguish between a legitimate signature and a forged signature. Thus, this attack violates the authenticity of the set of graphs.

Removing a triple from the graph (**AI.1**) and inserting the triple again (**AI.2**) into the same graph at another position is not considered as an attack. This corresponds to simply re-ordering the graph's triples which is not an unauthorized modification. Modifying a signed graph is not listed as a separate attack since it is already covered by the attacks **AI.1**, **AI.2** and **AI.3**. All attacks violate the authenticity of the set of signed graphs. Modifying a single graph in the set or changing the number of graphs results in a new set of graphs which was not approved by the signing party. This directly contradicts with the authenticity of the set. Forging signatures of a set of graphs (**AI.6**) does not violate the integrity of this set as the attacker does not perform any unauthorized modifications of already existing graph data. However, this attack still violates the authenticity of the set of graphs.

4.5.2. Cryptanalysis of the Canonicalization Function κ_N

The canonicalization function κ deterministically renames the blank nodes of a graph. The cryptographic security of the canonicalization function κ determines the difficulty for an attacker to forge a signature of a graph (**AI.6**). If the canonicalization function κ works correctly, semantically equivalent graphs are mapped to an identical canonical form and semantically different graphs are mapped to different canonical forms. However, if the canonicalization function κ does not work correctly, semantically identical graphs may be mapped to different canonical forms. Similarly, semantically different graphs may be mapped to the same canonical representation. As this would allow an attacker to replace a signed graph with another graph of identical canonical form (**AI.6**), the cryptographic security of the canonicalization function for graphs κ depends on the function's stability and correctness. The canonicalization function for Named Graphs κ_N is built upon the function κ. Thus, the cryptographic security of the function κ_N depends solely on the function κ. The particular security of the canonicalization function κ used in the four example configurations is further discussed for each of the four example configurations below.

4.5.3. Cryptanalysis of the Serialization Function ν_N

The serialization function ν transforms a canonical graph into a set of bit strings. The cryptographic security of the serialization function ν determines the difficulty for an attacker to modify a triple of a signed graph without being noticed by the verification function γ_N (**AI.3**). All four example configurations use N-Triples [25] as serialization format. In this format, each bit string in the set corresponds to a syntactical representation of a single triple in the graph. As the serialization function ν does not require any cryptographic operations or secret information such as secret keys, its cryptographic security is solely based on its stability. The output of the serialization function serves as the input for the hash function for graphs λ_N. In order to create the same hash value for semantically identical graphs, the output of the serialization function ν must be stable, i. e., the serialization function must output identical sets of bit strings for identical input graphs. Although N-Triples does not support any shortcuts such as compact URI repre-

sentations, it still allows different syntactical notations for a single triple. For example, the type and amount of whitespace characters between the subject, predicate, and object of a triple is not restricted in the N-Triples format. However, using different whitespace characters still affects the syntactical representation of a triple and therefore also the output of the hash function for graphs λ_N. In order to produce a stable serialization for a canonicalized graph, each triple must be transformed to a canonical serialization. This is achieved in all four example configurations by using a single space as delimiter between all three triple parts and removing all other whitespace characters. This results in a stable serialization function ν which outputs identical sets of bit strings for identical input graphs, prohibiting an attacker from modifying any triple (**AI.3**) without being noticed by the verification function γ_N. As the serialization function for Named Graphs ν_N is built upon the serialization function ν, its cryptographic security also depends on this function and the used serialization format.

4.5.4. Cryptanalysis of the Hash Function for Graphs λ_N

The basic hash function λ is used by the hash function for graphs λ_N for computing the hash value of a graph. The specific effects of the basic hash function on the cryptographic security of the graph signing function σ_N depend on its particular usage within the hash function for graphs λ_N. Basic hash functions λ used for signing graph data must be resistant against pre-image attacks, second pre-image attacks, and collision attacks [56]. A basic hash function λ is resistant against pre-image attacks if it is difficult to find any input value that maps to a given hash value. Hash functions with this property are called *one-way functions*. A basic hash function λ is resistant against second pre-image attacks if it is difficult to find another input value with the same hash value as a given input value. Finally, a basic hash function λ is resistant against collision attacks if it is difficult to find any two different input values with the same hash value. All collision resistant hash functions are also resistant against pre-image attacks and second pre-image attacks [56].

All four example configurations use SHA-2 [218] as basic hash function λ as recommended by the National Institute for Standards and Technology (NIST) [217]. Computing a hash value with SHA-2 is done incrementally in several rounds. Each round uses the output of the previous round as input. The total number of rounds depends on the cryptographic strength of the resulting hash value. All four example configurations use SHA-2 with an output length of 224 bits which corresponds to a cryptographic strength of 112 bit [217]. This particular variant of SHA-2 uses 64 rounds for computing the hash value. Although several attacks on SHA-2 have already been published [258, 158, 201], all of them only use a reduced version of the hash function in which not all 64 rounds are covered. These attacks focus on collision resistance which also includes resistance against pre-image attacks and second pre-image attacks [56]. As there is no attack on complete SHA-2 yet, this basic hash function λ can still be considered as collision resistant.

The hash function for graphs λ_N is built upon the basic hash function λ and outputs a hash value for each graph to be signed. The cryptographic security of this function determines the difficulty for an attacker to remove triples from a signed graph (**AI.1**),

insert new triples into a signed graph (**AI.2**), modify triples in a signed graph (**AI.3**), and forge a signature of a graph (**AI.6**). In order to prohibit such attacks, the hash function for graphs λ_N must create different hash values for semantically different graphs. Additionally, the function must create identical hash values for semantically identical graphs. The cryptographic security of the hash function for graphs λ_N depends on the security of the used basic hash function λ and on how the function λ is particularly used within the function λ_N. As the four example configurations use different hash function for graphs λ_N, the cryptographic security of these functions is further discussed for each of the configurations below.

4.5.5. Cryptanalysis of the Combining Function for Graphs ϱ_N

The combining function for graphs ϱ_N combines the hash values of several graphs created with the hash function for graphs λ_N into a single hash value. The cryptographic security of the combining function for graphs ϱ_N determines the difficulty for an attacker to remove a graph from the set of signed graphs (**AI.4**) or insert a new graph into this set (**AI.5**) without being noticed by the verification function γ_N. All four example configurations use a combining function for graphs ϱ_N which operates similar as the hash function for graphs λ_N of Carroll [72]. As defined in Equation 4.14, the combining function for graphs ϱ_N sorts the hash values of all graphs, concatenates them, and hashes the resulting string using a basic hash function λ. Using a stable sorting algorithm such as merge sort [241], the sorting function produces identical output values for identical input values. Similarly, the concatenation function also produces identical output values for identical input values. As both functions do not influence the stability and security of the combining function for graphs ϱ_N, the cryptographic security of this function solely depends on the used basic hash function λ. Using SHA-2 [218] as basic hash function, the combining function for graphs ϱ_N can be considered as secure. Modifying the processed set of hash values by removing a hash value of a graph (**AI.4**) or inserting a hash value of a new graph (**AI.5**) results in a different output of the function ϱ_N. Unauthorized modifications like these can therefore be identified by the verification function γ_N.

4.5.6. Cryptanalysis of the Signature Function φ

The signature function φ digitally signs the hash value created with the combining function for graphs ϱ_N using the signing party's secret key k_s. The cryptographic security of the signature function φ determines the difficulty for an attacker to forge a signature for a set of graphs (**AI.6**). Signature functions φ can generally be used for digitally signing arbitrary messages. In the graph signing framework Siggi, these messages correspond to graph data. Signature functions φ must be resistant against key-only attacks and message attacks [130]. In both types of attacks, an attacker forges a valid signature for a message of own choice so that the resulting signature can be verified with the real signing party's public key k_p. In a key-only attack, the attacker has only access to this public key k_p whereas a message attack provides the attacker with several messages and their respective signature values. Message attacks can be further distinguished by the

influence that the attacker has on the received messages [130]. The strongest message attack is the adaptive chosen-message attack in which the attacker can requests a set of messages to be directly signed by the real signing party with the secret key k_s. The resulting pairs of messages and corresponding signatures can then be used by the attacker to create valid signatures for new messages of own choice. In order to prohibit an attacker from forging signatures (**AI.6**), a digital signature scheme must be resistant against adaptive chosen-message attacks as these are the strongest attacks which include all other weaker attacks.

All four example configurations use RSA [251] as signature function φ as recommended by NIST [217]. An RSA key pair consists of a secret key $k_s = \langle d, N \rangle$ and a public key $k_p = \langle e, N \rangle$ with d being a secret value, e being its public counterpart, and N being the product of large primes p and q. The primes p and q are only known to the owner of the secret key k_s. The cryptographic security of an RSA signature is based on the factorization problem for large numbers which is considered to be hard to solve [174]. The larger the prime factors of a number are, the more difficult it is to find them. An attacker who can factorize N can easily compute the value of d. Thus, the size of N and its prime factors p and q defines the cryptographic security of an RSA key pair. The four example configurations use a value of N with a length of 2048 bit as recommended by NIST [217]. This corresponds to a cryptographic strength of 112 bit. Although factorizations of 512 bit and 768 bit numbers N have already been implemented in reasonable time [77, 180], a factorization of a 2048 bit number has not been found yet.

However, factorizing N is not necessary in order to forge signatures for a message when using plain RSA which does not use any additional security operations. Plain RSA is not secure against key-only attacks which allows an attacker to create a valid signature for a message of own choice using only the public key $\langle e, N \rangle$ [175]. Furthermore, plain RSA is also not secure against adaptive chosen-message attacks in which an attacker choses two different messages to be signed by the signing party in order to create a valid signature for any other message. However, such attacks can be made more difficult for an attacker by signing a hash value of the message instead of signing the message directly [175]. All four example configurations use the combining function for graphs ϱ_N in order to create a combined hash value of all graphs to be signed. This hash value is used as input for the signature function φ which makes the attacks described above much harder to implement. In summary, the signature function φ used in the four example configurations can be considered as secure and resistant against signature forging attacks (**AI.6**).

4.5.7. Cryptanalysis of Configuration A

Configuration A uses the canonicalization function κ and hash function for graphs λ_N of Carroll [72]. The canonicalization function κ also includes a serialization function ν and outputs a canonical serialization of a graph. The canonicalization function κ removes all blank node identifiers from the graph, sorts the resulting triples based on their canonical N-Triples representation, and renames the blank nodes based on the order of the triples. If the order of the triples is not unique due to the removal of the blank node identifiers, additional triples are added for these blank nodes in order to create a unique triple

order. This ensures that the canonicalization function κ of Carroll always produces a stable output, prohibiting an attacker from finding two semantically different graphs with the same canonical form (**AI.6**). As the canonicalization function κ does not perform any further cryptographic operations, it does not affect the cryptographic security of configuration A.

The hash function for graphs λ_N of Carroll sorts all bit strings of the canonicalized and serialized graph, concatenates them, and computes a hash value of the resulting string using a basic hash function λ. Hash values of several graphs are combined with the combining function ϱ_N into a single value which is then signed using the signature functions φ. Since the combining function for graphs ϱ_N also uses a basic hash function λ, the cryptographic security of configuration A solely depends on the basic hash function λ and on the signature functions φ. If an attacker removes a triple from a signed graph (**AI.1**), its hash value changes which results in a different hash value of the set of graphs and an invalid signature as well. Similarly, adding new triples to a single graph (**AI.2**) or modifying existing triples (**AI.3**) changes the graph's hash value and thus invalidates the signature of the set as well. In order to prohibit an attacker from forging graph signatures (**AI.6**), configuration A uses SHA-2 as collision resistant basic hash function λ and RSA as signature function φ which is resistant against adaptive chosen-message attacks.

4.5.8. Cryptanalysis of Configuration B

Configuration B only differs from configuration A by using an additional split function which partitions a graph into disjoint MSGs. As the MSGs of a graph are well-defined, the split function operates deterministically and produces identical outputs for identical inputs. Thus, the split function does not influence the cryptographic security of configuration B. The canonicalization function κ and hash function for graphs λ_N of configuration B are identical to configuration A. Thus, the cryptographic security of configuration B also depends on the basic hash function λ and on the signature functions φ.

4.5.9. Cryptanalysis of Configuration C

Configuration C uses the canonicalization function κ and hash function for graphs λ_N of Fisteus et al. [110]. The canonicalization function κ uses hash values as blank node identifiers. The hash values are based on the structure of the graph and are computed in several steps. First, the initial hash value of all blank nodes is set to a constant value. Second, the hash value of each triple is computed with a hash function λ_t. The function combines the hash values of a triple's subject, predicate, and object using XOR modulo a large prime p. Finally, the hash values of the blank nodes are computed again using a hash function λ_b. The function combines the hash value of all triples of a blank node using XOR modulo a large prime p. If this results in two or more blank node identifiers being identical, the hash values of the blank nodes and triples are computed again. This process is repeated until there are either no collisions left or if the

remaining collisions cannot be resolved. In the latter case, the graph contains disjoint subgraphs which are syntactically equivalent except for their blank node identifiers. As these subgraphs are also isomorphic, they cannot be semantically distinguished from each other. Thus, the semantics of the canonicalized graph is also not affected by these subgraphs. The stability of the canonicalization function therefore only depends on the two hash functions λ_t and λ_b. Both functions are based on XHash, which is proven to be vulnerable to pre-image attacks and collision attacks [29]. This allows an attacker to create colliding hash values with the function λ_t and replace an existing triple with another triple of identical hash value. In a similar attack on the hash function λ_b, an attacker can replace blank nodes in the graph with other blank nodes sharing the same hash value. Both attacks allow an attacker to modify a triple in a signed graph (**AI.3**) without being noticed by the verification function γ_N.

The hash function for graphs λ_N of Fisteus et al. also uses the hash function λ_t for computing the hash value of each triple in the graph. The function λ_N combines these hash values by multiplying them modulo a large prime p. This operation is based on MuHash [29] which is considered to be collision resistant as long as the hash function λ_t is a one-way-function and the prime p is large enough. If both requirements are met, the cryptographic security of MuHash can be reduced to the discrete logarithm problem [29] which is considered to be hard to solve [197]. The hash function for graphs λ_N uses a prime p with a length of 1024 bit as recommended in the literature [285]. However, the hash function λ_t is not a one-way function as it is not resistant against pre-image attacks. Since the hash function for graphs λ_N is therefore vulnerable as well, it is possible to find two different input graphs with the same hash value. This allows an attacker to replace a graph in the set of signed graphs with another graph of identical hash value (**AI.1**, **AI.2**, **AI.3**).

In order to improve the cryptographic security of configuration C, the two hash functions λ_t and λ_b are modified as follows: The XOR operation used in the hash function λ_b is replaced with multiplication, resulting in a variant of MuHash. As prime p, the hash function λ_b also uses a value with a length of 1024 bit. The hash function λ_t is modified so that it computes the hash value of a triple by concatenating all of its parts and hashing the resulting bit string using a basic hash function λ. As basic hash function λ, SHA-2 is used with an output length of 224 bit. These modifications improve the cryptographic security of the hash function for graphs λ_N and thus the overall security of configuration C as well.

4.5.10. Cryptanalysis of Configuration D

Configuration D uses the canonicalization function κ and hash function for graphs λ_N of Sayers and Karp [261]. Instead of altering the blank node identifiers of a graph, the canonicalization function κ stores the current identifier of each blank node in an additional triple which is added to the original graph. The subject of such a triple corresponds to the blank node and the object is a literal containing the blank node's current label. As the new triples are deterministically created, the canonicalization function κ outputs the same canonical graph for identical input graphs and different canonical graphs for

different input graphs. The additional triples can be used to re-create the original graph by undoing any later modifications of the graph's blank node identifiers. Thus, the canonicalization function κ of Sayers and Karp prohibits an attacker from finding two semantically different graphs with the same canonical form (**AI.6**).

The hash function for graphs λ_N of Sayers and Karp computes the hash value of each serialized triple in the graph using a basic hash function λ and multiplies the resulting hash values modulo a large prime p. Similar to the hash function for graphs λ_N of Fisteus et al. [110], the hash function for graphs λ_N of Sayers and Karp is also based on MuHash [29]. Thus, the cryptographic security of hash function for graphs λ_N of Sayers and Karp is based on the collision resistance of the used basic hash function λ and the size of the prime p. The hash function for graphs λ_N uses SHA-2 with an output length of 224 bit as basic hash function λ and a prime p with a length of 1024 bit as recommended in the literature [285]. As the used SHA-2 variant is collision resistant, an attacker cannot replace an existing triple in the graph with another triple of identical hash value (**AI.3**) without being noticed by the verification function γ_N. Since the resulting hash function for graphs λ_N is collision resistant as well, an attacker cannot insert new triples into the graph (**AI.2**) or remove existing triples from the graph (**AI.1**) without changing the hash value of the graph. Configuration D prohibits an attacker from forging graph signatures (**AI.6**) by using SHA-2 as collision resistant basic hash function λ and RSA as signature function φ which is resistant against adaptive chosen-message attacks.

4.6. Performance of the Four Configurations

This section assesses the performance of the four example configurations A, B, C, and D introduced in Section 4.4. Each configuration and its sub-functions are analyzed and the experimental findings are compared with the theoretical analysis provided in Section 4.4. In the experiments, the runtime and required memory for singing a single graph and the number of additional triples created by the graph signing function σ_N and the assembly function α_N were measured. As datasets, synthetic RDF graphs ranging from 10,000 to 250,000 triples were used. The datasets were created with the Berlin SPARQL benchmark (BSBM) [40] which provides a framework for comparing different RDF data stores. In order to measure the influence of blank nodes in the graph on the graph signing function σ_N and the assembly function α_N, different percentages of blank nodes were introduced into the BSBM graph with 250,000 triples. This was done by mapping 1%, 5%, 10%, 25%, and 50% of the unique subject URIs [131] in the graph to corresponding blank node identifiers. The graph signing framework is implemented in Java using OpenJDK version 1.7. A detailed description of the implementation's architecture is provided in [263]. The evaluation was conducted using a system with 100 GB memory and an Intel® Xeon® CPU with 2.00 GHz running Debian GNU/Linux version 7. In order to avoid interference with statistical outliers, each operation was performed ten times and the mean value was calculated. Additionally, three warm-up phases were performed to reduce the influence of any initializations on the measured values.

4.6.1. Runtime and Memory Usage of the Functions κ_N and λ_N

Figure 4.2 shows the effect on the canonicalization function κ_N and hash function for graphs λ_N when increasing the size of a signed graph without blank nodes. The functions are taken from the four example configurations A, B, C, and D. As depicted, the runtime and required memory of both functions increase as the size of the graph increases. Since configurations A and B use the same canonicalization function κ_N, the runtime and required memory is also the same. As expected, the runtime of this function has

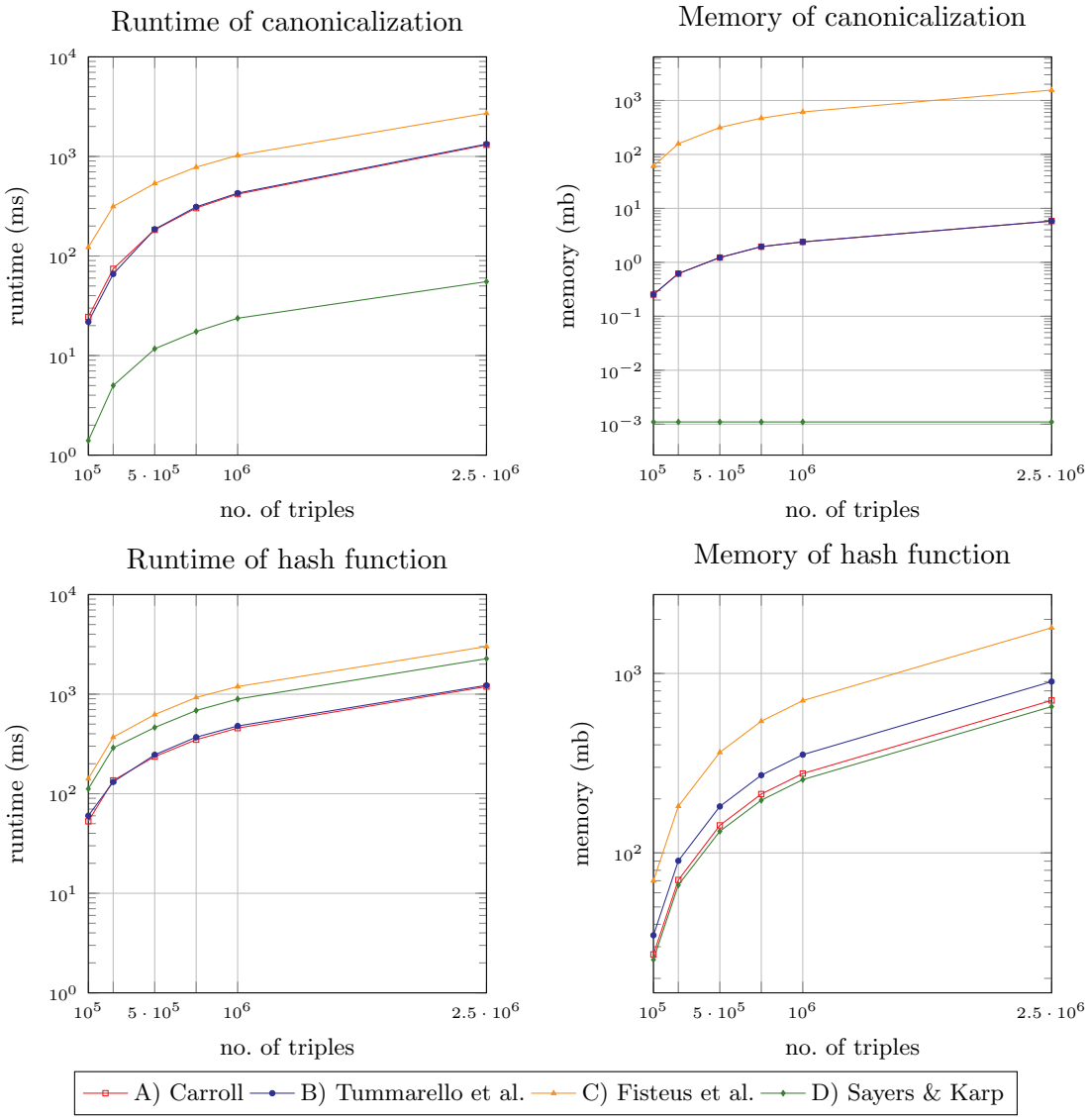


Figure 4.2.: Runtime and required memory of the canonicalization functions κ_N and hash functions for graphs λ_N from the four example configurations A–D.

complexity of $\mathcal{O}(n \log n)$ and the required memory corresponds to $\mathcal{O}(n)$. Although configurations A and B also use the same hash function for graphs λ_N, the hash function of configuration B requires more memory. This is due to the splitting function used in configuration B that splits the graph into disjoint MSGs. The splitting function is executed after the canonicalization function κ_N. As the hash function for graphs λ_N of configuration B is applied separately to all resulting MSGs, a slightly higher memory usage is observed. Although the expected runtime of the hash function for graphs λ_N used in configurations A and B has a complexity of $\mathcal{O}(n \log n)$, the observed complexity is only $\mathcal{O}(n)$. The theoretical complexity of the hash function for graphs λ_N is based on the complexity of the sorting algorithm used by this function. However, the canonicalization function κ_N used in configurations A and B already sorts the triples in the graph. As consequence, this reduces the runtime of the sorting operation used in the hash function for graphs κ_N since the triples are still sorted. Furthermore, this also reduces the runtime of the hash function for graphs κ_N used in configurations A and B.

Regarding configurations C and D, both the runtime and the required memory of the canonicalization function κ_N and the runtime of the hash function for graphs λ_N are as expected. Due to their incremental approach, the hash functions for graphs λ_N of these two configurations can be implemented with a memory complexity of $\mathcal{O}(1)$. After having processed one triple, the temporarily allocated memory can be freed and used again to process the next triple. However, Java does not support to explicitly free memory which results in a permanent increase of memory usage. Thus, the observed memory complexity of the hash functions for graphs λ_N of configurations C and D is $\mathcal{O}(n)$.

4.6.2. Accumulated Runtime of all Functions

Figure 4.3 shows the accumulated runtime of the four example configurations A–D for signing the BSBM graphs with no blank nodes. As depicted, the total runtime increases for all four configurations as the size of the signed graph increases. The signature functions φ of configurations A, C, and D are called only once for the whole graph, making their runtime independent from the graph size. On the other hand, the signature function φ of configuration B signs each MSG in the graph individually. In a graph with no blank nodes, each triple corresponds to one MSG. Thus, for such graphs, the number of MSGs increases as the size of the graph increases. This results in a larger runtime of configuration B's signature function φ. Although the runtime of the cryptographic operations conducted by the signature function φ are usually insignificant compared to other functions, executing a large amount of them highly influences the total runtime.

4.6.3. Influence of Blank Nodes

Figure 4.4 shows the influence of blank nodes on the four example configurations A–D for signing the BSBM graph with 250,000 triples. The overall runtime of configuration A increases and the required memory decreases as the signed graph contains more blank nodes. The larger runtime is primarily caused by the configuration's canonicalization

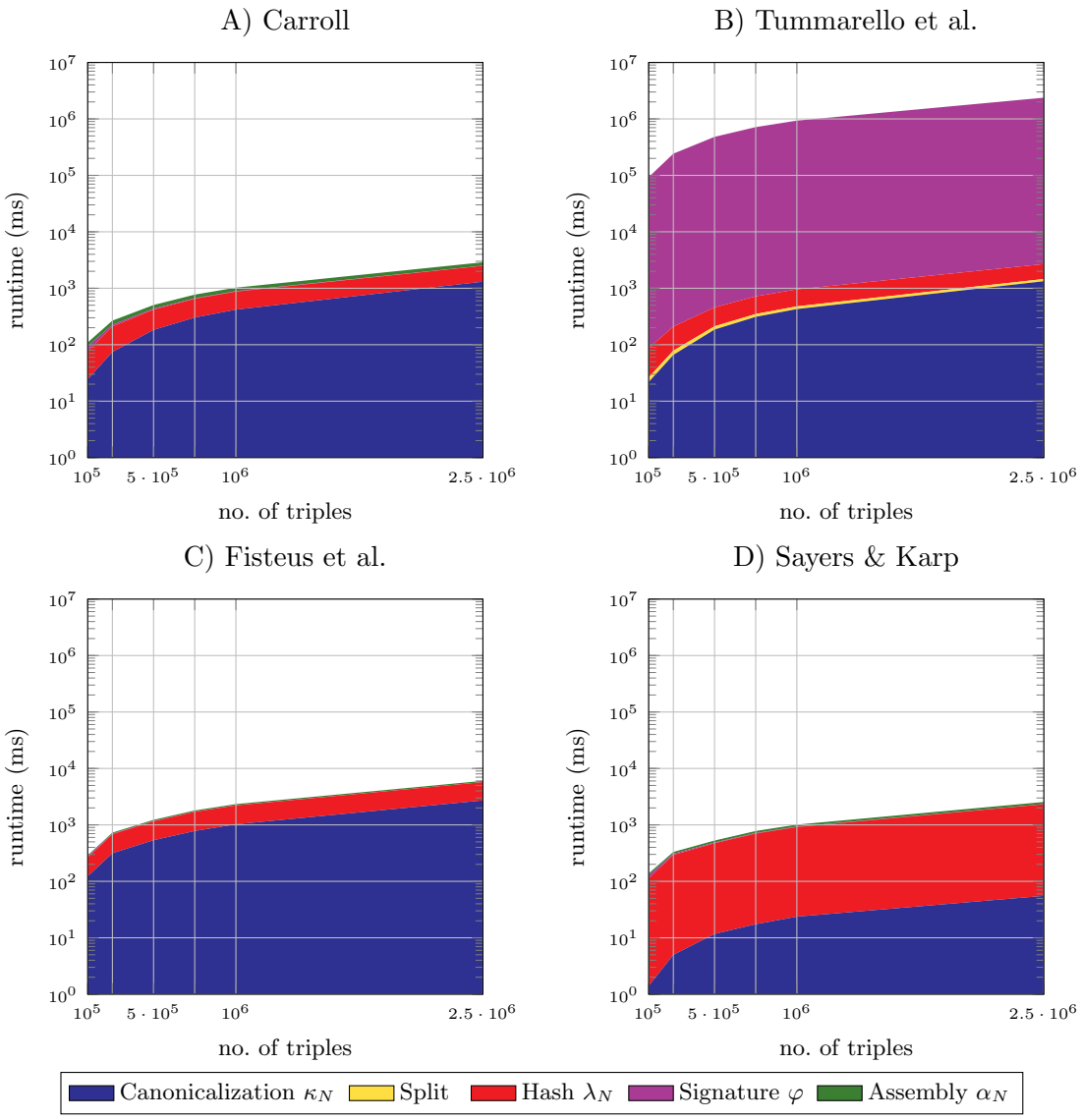


Figure 4.3.: Accumulated runtime of the four example configurations A–D.

function κ_N which renames all blank node identifiers. Renaming more blank node identifiers also increases the function's runtime. The memory required by configuration A's hash function for graphs λ_N and assembly function α_N declines, resulting in less required memory in total. Both functions operate on string representations of URIs and blank node identifiers. Since blank node identifiers usually have shorter string representations than URIs, processing them also requires less memory. Since all graphs used in the experiments have no blank nodes b_h, which require special treatment by configuration A's canonicalization function (see Section 4.4.1), the signature overhead is constant.

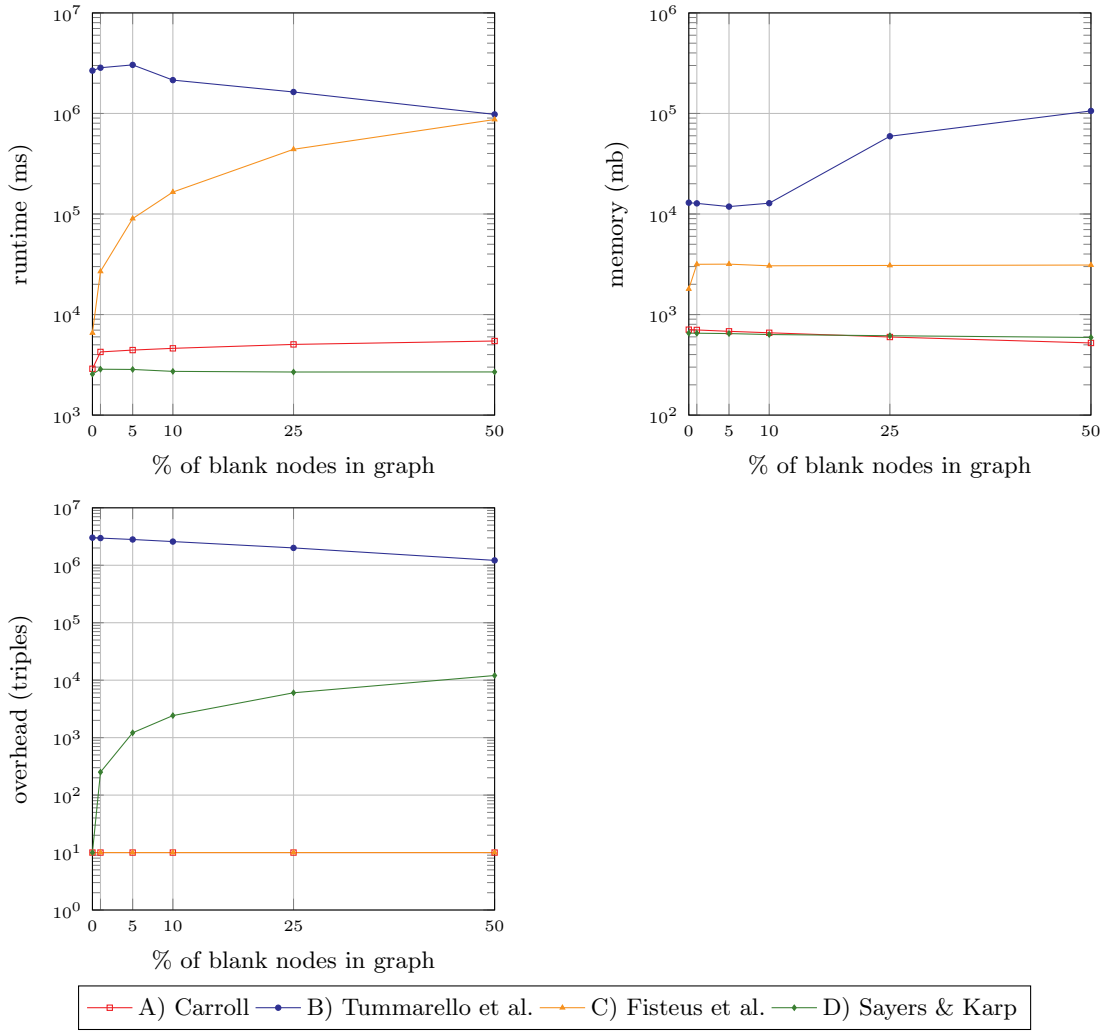


Figure 4.4.: Effect of blank nodes on the runtime, memory usage, and signature overhead for signing a graph with 250,000 triples.

The runtime and required memory of configuration B is mostly affected by the split function, the signature function φ, and the assembly function α_N. The runtime and required memory of the split function increase as the graph contains more blank nodes. At the same time, there are less MSGs to be signed which results in less signature data. Thus, the runtime and required memory of both the signature function φ and the assembly function α_N decrease. Similarly, the signature overhead also decreases as the assembly function α_N creates fewer signature graphs.

The total runtime and required memory of configuration C mainly depends on the canonicalization function κ_N and hash function for graphs λ_N. The canonicalization function κ_N renames blank nodes based on their hash values. Since the computation of a blank node's hash value requires more operations than the hash value of a URI or literal

(see Section 4.5.9), the runtime of the canonicalization function κ_N increases as more blank nodes are introduced into the graph. If the graph contains no blank nodes, only one iteration for renaming blank node identifiers is performed. However, all signed graphs with blank nodes result in two renaming iterations and thus double the memory required by the canonicalization function κ_N. As the hash function for graphs λ_N also requires the computation of blank nodes' hash values, its runtime increases with the number of blank nodes in the graph. Thus, the total runtime of configuration C increases as well. Configuration C does not create any additional triples which results in a constant signature overhead.

The overall runtime and required memory of configuration D is mainly affected by its hash function for graphs λ_N. The runtime of this function depends on the number of triples and the length of the strings to be hashed. The canonicalization function κ_N of configuration D creates an additional triple for each blank node in the graph. These additional triples are also processed by the hash function for graphs λ_N. The more blank nodes the graph contains, the more additional triples are hashed. However, blank node identifiers are also shorter than URIs and their hash values can be computed faster. Both aspects result in an almost constant runtime of the hash function for graphs λ_N. Although the memory required by the hash function for graphs λ_N is also affected by both aspects, its memory usage declines. Thus, the shorter string representations of the blank node identifiers have a larger impact on the hash function's memory usage than on its runtime. The signature overhead rises in configuration D since the signed graph contains additional triples for all blank nodes created by the canonicalization function κ_N.

4.6.4. Summary

All four example configurations A–D use RSA [251] with a key length of 2048 bit as signature function φ. This corresponds to a cryptographic security of 112 bit [217]. If a cryptographic security of 128 bit is desired, a key length of 3072 bit could be used instead. However, this would increase the runtime of the signature function φ by a factor of three. Although this hardly affects the overall runtime of configurations A, C, and D as they only compute a single signature for the whole graph, a larger key length highly increases the runtime of configuration B which signs all MSGs individually. As alternative, Elliptic Curve DSA [219] with a key length of 256 bits could be used as the signature function φ. It has the same security as RSA with a key length of 3072 bit but is about 76 times faster.

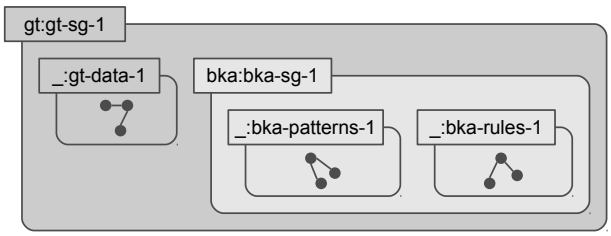
The experimental results show that the approach by Sayers and Karp (configuration D) is best suited for signing graphs with few blank nodes. For such graphs, the overhead created by the canonicalization function κ_N may be negligible. Signing RDF graphs with many blank nodes should be done with the approach by Fisteus et al. (configuration C). If indeed the approach by Tummarello et al. (configuration B) shall be used, e. g., for signing MSGs individually, the faster Elliptic Curve DSA should be used instead of RSA as signature function φ.

4.7. Applications and Use Cases

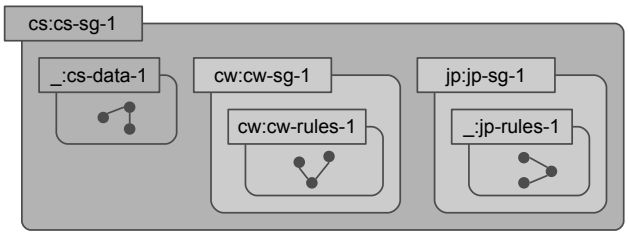
This section describes how the graph signing framework Siggi is used for implementing the scenarios introduced in Section 2. The first scenario described in Section 2.1 focuses on the regulation of Internet communication and is covered in Sections 4.7.1 to 4.7.5. The second scenario introduced in Section 2.2 covers the secure management of medical data and is addressed in Section 4.7.6.

4.7.1. Signing Policies for Regulating Internet Communication

The scenario for regulating Internet communication defines a workflow for creating and distributing regulation policies. The workflow is described in Section 2.1.2 and involves several authorities. Each authority creates a specific part of a regulation policy, signs it, and sends it to the next authority. This authority verifies the signature, adds its own part of the policy, signs the result again, and sends it to the next authority. The process is repeated until the policy is completed and contains all regulation details. The subsequent examples are structured along the workflow of the German Telecom depicted in Figure 2.2a and on the workflow of the German comprehensive school depicted in Figure 2.2b. The regulation policies are modeled with the InFO policy language and its domain-specific extensions as described in Chapter 3. The example graphs used in this section correspond to the example policies provided in Section 3.4.1. Each example is represented using an extension of the TriG syntax [38] that supports nested Named Graphs and blank nodes as graph identifiers. Alternative formats for representing signed graph data are also possible and are discussed in Section 4.9.6. Figure 4.5a and Fig-



(a) Resulting graph signed by the German Telecom



(b) Resulting graph from the German comprehensive school

Figure 4.5.: Different examples of iteratively signed graphs which containing policies for regulating Internet communication.

ure 4.5b show the resulting graphs after having completed the two workflows. Figure 4.5a depicts the resulting graph signed by the German Telecom. The graph contains signed graph data from the BKA which consists of a graph containing T-box knowledge and another graph with A-box knowledge. Figure 4.5b depicts the graph signed by the German comprehensive school. The graph contains a signed Named Graph from ContentWatch and another signed graph from JusProg. All signed graphs are created by applying the graph signing function σ_N and the assembly function α_N. In the following, the signing process for each party is demonstrated. The examples are based on configuration C of the graph signing framework as discussed in Section 4.4.3 and are presented along the functional requirements **RI.F.1** to **RI.F.5** as described in Section 4.2.

4.7.2. Signing an OWL Graph

The first workflow covers the regulation of the Stormfront network by the German Telecom. The BKA creates the regulation details for prohibiting access to the network based on the network's domain name. To this end, the BKA provides corresponding ontology design patterns (**RI.F.2.2**) which support the definition of name server-specific regulation policies. These patterns correspond to the patterns of the Name Server Ontology which is further described in Appendix A.3. The BKA uses these patterns in order to create two particular flow control rules. The rules contain the domain name of the Stormfront network and the BKA as their rule data provider. The complete rules are depicted in the InFO chapter in Figure 3.19a and Figure 3.19b. After having compiled all data, the BKA signs both the patterns of the Name Server Ontology and the two flow control rules. A fragment of the resulting graph is depicted in Listing 4.1. The graph contains the design patterns, the two flow control rules, and a signature graph. The patterns contain T-box knowledge of the BKA (**RI.F.3.1**) and are modeled as a separate graph shown in lines 30 to 37. The graph is identified by the blank node `_:bka-patterns-1`. The two flow control rules are encoded as an OWL graph (**RI.F.1.2**) and provide A-box knowledge (**RI.F.3.2**). They correspond to the graph `_:bka-rules-1` which is also identified by a blank node and shown in lines 40 to 51. Signing both `_:bka-patterns-1` and `_:bka-rules-1` results in the new Named Graph `bka:bka-sg-1` and a signature graph. The Named Graph `bka:bka-sg-1` contains the graphs `_:bka-patterns-1` and `_:bka-rules-1` as its content graphs and the signature graph as its annotation graph. The graph `bka:bka-sg-1` is shown in lines 11 to 52 and the triples of the signature graph are shown in lines 14 to 27. The graph `bka:bka-sg-1` and its two content graphs `_:bka-patterns-1` and `_:bka-rules-1` are also shown in Figure 4.5 as part of the graph `gt:gt-sg-1`.

```
1 @prefix bka:  <http://icp.it-risk.iwvi.uni-koblenz.de/policies/bka-graph.owl#> .
2 @prefix dns:
3    <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/name_server_flow_control.owl#> .
4 @prefix dul:  <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#> .
5 @prefix owl:  <http://www.w3.org/2002/07/owl#> .
6 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
7 @prefix sig:  <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/signature.owl#> .
8 @prefix tec:
```

```
9    <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/technical_regulation.owl#> .
10
11 bka:bka-sg-1 {
12
13   # Signature graph of the BKA
14   {
15     bka:bka-sig-1 a sig:Signature ;
16       sig:hasGraphSigningMethod bka:bka-gsm-1 ;
17       sig:hasSignatureValue "TmV2ZXIgR29ubmEgR2l2ZSBZb3UgVXA=" ;
18       sig:hasVerificationCertificate bka:bka-pck-1 .
19     bka:bka-gsm-1 a sig:GraphSigningMethod ;
20       sig:hasGraphCanonicalizationMethod sig:gcm-fisteus-2010 ;
21       sig:hasGraphSerializationMethod sig:gsm-n-triples ;
22       sig:hasDigestMethod sig:dm-sha224 ;
23       sig:hasGraphDigestMethod sig:gdm-fisteus-2010 ;
24       sig:hasGraphCombiningMethod sig:cm-sort-concat ;
25       sig:hasSignatureMethod sig:sm-rsa .
26     ...
27   }
28
29   # T-box knowledge containing design patterns
30   _:bka-patterns-1 {
31     dns:DomainNameBlockingRuleMethod a owl:Class ;
32       rdfs:subClassOf tec:DenyingFlowControlRuleMethod , [
33         a owl:Restriction ; owl:onProperty dul:isSatisfiedBy ;
34         owl:allValuesFrom dns:DomainNameBlockingRuleSituation
35       ] .
36     ...
37   }
38
39   # A-box knowledge containing flow control rules
40   _:bka-rules-1 {
41     bka:dnsr-1 a dns:DomainNameBlockingRuleMethod ; dul:defines bka:rdp-1 , bka:rt-1 ,
42       bka:sr-1 , bka:ss-1 , bka:rr-1 , bka:rs-1 .
43     bka:rr-1 a tec:ReceiverRole ; dul:classifies cn-1 .
44     bka:cn-1 a tec:ComputerNetwork ;
45       dul:hasQuality bka:dq-1 ; dul:hasSetting bka:dnsrs-1 .
46     bka:dq-1 a tec:DomainQuality ; dul:hasRegion bka:dnr-1 .
47     bka:dnr-1 a tec:DomainNameRegion ;
48       tec:hasDomainName "stormfront.org" ; dul:hasSetting bka:dnsrs-1 .
49     bka:dnsrs-1 a dns:DomainNameBlockingRuleSituation ; dul:satisfies bka:dnsr-1 .
50     ...
51   }
52 }
```

Listing 4.1: Example of a signed OWL graph.

The complete signature graph created by the assembly function α_N is depicted in Figure 4.6. The signature graph is modeled with the Signature Ontology which is presented in Appendix C and based on the XML signature standard [20]. The signature graph stores the computed signature `bka-sig-1`, its signature value, and all parameters of the graph signing function σ_N required for verifying this value. In the signature graph, the function σ_N is identified as `bka-gsm-1` and linked to all its subfunctions.

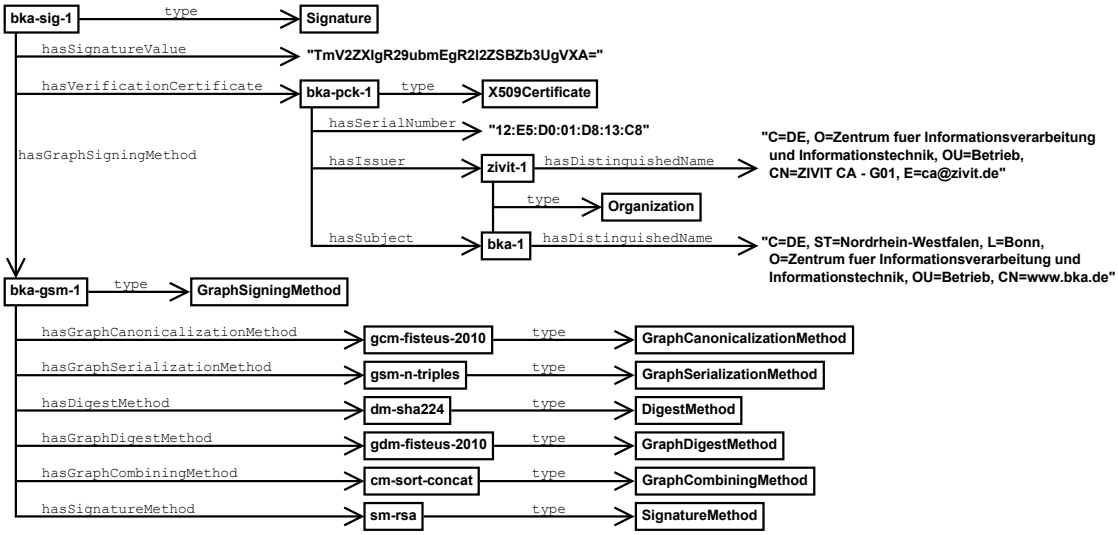


Figure 4.6.: Example signature graph of the BKA. The graph consists of 25 triples and is modeled using the Signature Ontology described in Appendix C. The Signature Ontology is based on the XML signature standard [20] and describes all details for verifying a graph signature.

This includes the canonicalization function for graphs `gcm-fisteus-2010`, the serialization function `gsm-n-triples`, the basic hash function (also called digest function) `dm-sha224`, the hash function for graphs `gdm-fisteus-2010`, the combining function for graphs `cm-sort-concat`, and the signature function `sm-rsa`. In order to verify the signature value, the signature graph also covers a reference to the BKA's public key certificate. The certificate contains the corresponding public key of the BKA's secret key, which was used as the signature key. The certificate is represented as `bka-pck-1` and corresponds to an X.509 certificate [82] issued by the German Center for Information Processing and Information Technology (Zentrum für Informationsverarbeitung und Informationstechnik; ZIVIT)[2]. The certificate's owner is identified as `bka-1` and its issuer is represented as `zivit-1`. X.509 certificates are uniquely identified by their serial number and the distinguished name [317] of their issuer. A distinguished name is an hierarchically structured identifier of an organization or natural person. In order to precisely identify the certificate used for verifying the graph's signature, the signature graph contains the certificate's serial number as well as the distinguished names of its owner and issuer.

4.7.3. Iteratively Signing of Graphs

The German Telecom receives the signed Named Graph `bka:bka-sg-1` from the BKA and verifies its signature. Although this graph contains two flow control rules which already provide some regulation details, it does not contain a complete flow control pol-

[2] `http://www.zivit.de`, last accessed: 01/21/16

icy with all necessary information for the regulation's enforcement. For example, the graph does not define the name server which shall be used as the regulation's enforcing system. Thus, the German Telecom completes the flow control policy by adding its own RDF graph `_:gt-data-1` with additional regulation details. These details include the IP address `2.160.15.78` of the enforcing name server `ns-2`, the regulation's legal authorization `la-1`, and the German Telecom's code of conduct `om-1`. The resulting flow control policy corresponds to the example policy of the InFO chapter depicted in Figure 3.16c. Subsequently, the German Telecom signs its own graph `_:gt-data-1` (**RI.F.2.4**) together with the received Named Graph `bka:bka-sg-1`. Thus, the access provider iteratively signs the graph `bka:bka-sg-1` (**RI.F.4**) which results in the new Named Graph `gt:gt-sg-1` depicted in Listing 4.2. This graph contains the created signature graph shown in lines 8 to 14, the graph `_:gt-data-1` created by the German Telecom shown in lines 17 to 27, and the BKA's Named Graph `bka:bka-sg-1` shown in lines 30 to 40. The signature graph covers the used graph signing function `gt:gt-gsm-1` (line 10), the resulting signature value (line 11), and the public key certificate `gt:gt-pck-1` of the German Telecom (line 12). The Named Graph `gt:gt-sg-1` contains the signature graph as its annotation graph and the two graphs `_:gt-data-1` and `bka:bka-sg-1` as its content graphs.

```
1  @prefix bka:  <http://icp.it-risk.iwvi.uni-koblenz.de/policies/bka-graph.owl#> .
2  @prefix gt:   <http://icp.it-risk.iwvi.uni-koblenz.de/policies/gt-graph.rdf#> .
3  ...
4
5  gt:gt-sg-1 {
6
7    # Signature Graph of the German Telecom
8    {
9      gt:gt-sig-1 a sig:Signature ;
10       sig:hasGraphSigningMethod gt:gt-gsm-1 ;
11       sig:hasSignatureValue "YXJlIGJlbG9uZyB0byB1cw==" ;
12       sig:hasVerificationCertificate gt:gt-pck-1 .
13     ...
14   }
15
16   # Flow regulation details of the German Telecom
17   _:gt-data-1 {
18     gt:dnsp-1 a tec:FlowControlPolicyMethod ; dul:hasMember bka:dnsr-1 , bka:dnsr-2 ;
19       dul:defines gt:ro-1 , gt:es-1 , gt:la-1 , gt:om-1 .
20     gt:es-1 a tec:EnforcingSystem ; dul:classifies ns-2 .
21     gt:ns-2 a tec:NameServer ; dul:hasSetting gt:dnsps-1 ; dul:hasQuality gt:ipaq-2 .
22     gt:ipaq-2 a tec:IPAddressQuality ; dul:hasRegion gt:ipar-6 .
23     gt:ipar-6 a tec:IPv4AddressRegion ; dul:hasSetting gt:dnsps-1 ;
24       tec:hasIPAddress "2.160.15.78" .
25     gt:dnsps-1 a tec:FlowControlPolicySituation ; dul:satisfies gt:dnsp-1 .
26     ...
27   }
28
29   # Graph data received from the BKA
30   bka:bka-sg-1 {
31     {
```

```
32      bka:bka-gsm-1 a sig:Signature ;
33        sig:hasGraphSigningMethod bka:bka-gsm-1 ;
34        sig:hasSignatureValue "TmV2ZXIgR29ubmEgR2l2ZSBZb3UgVXA=" ;
35        sig:hasVerificationCertificate bka:bka-pck-1 .
36      ...
37    }
38    _:bka-patterns-1 { ... }
39    _:bka-rules-1 { ... }
40  }
41 }
```

Listing 4.2: Example of iteratively signed graphs.

4.7.4. Signing a Named Graph

The second workflow describes how the German comprehensive school prohibits its students from accessing pornographic web content. To this end, the school receives regulation details from ContentWatch and JusProg. The regulation details contain the URLs of the websites to be blocked and are modeled with the Application-Level Proxy Ontology, which is further described in Appendix A.4. ContentWatch provides its regulation details as Named Graph (**RI.F.1.3**) while JusProg uses a regular RDF graph (**RI.F.1.1**). Due to the design of the graph signing framework described in Section 4.3, signing a Named Graph is similar to signing an RDF graph or OWL graph. Listing 4.3 depicts the Named Graph `cw:cw-sg-1` created by ContentWatch. The regulation details of ContentWatch are modeled as the graph `cw:cw-rules-1` which is shown in lines 18 to 24. They correspond to the example flow control rules depicted in Figure 3.19a and Figure 3.19b. Signing the Named Graph `cw:cw-rules-1` results in the signature graph shown in lines 9 to 15. The signature graph covers the used graph signing function `cw:cw-gsm-1` (line 11), the signature value (line 12), and ContentWatch's public key certificate `cw:cw-pck-1` (line 13). The signature graph and the Named Graph `cw:cw-rules-1` are part of the newly created Named Graph `cw:cw-sg-1` (lines 6 to 25), which contains the signature graph as its annotation graph and the graph `cw:cw-rules-1` as its content graph.

```
1 @prefix cw:  <http://icp.it-risk.iwvi.uni-koblenz.de/policies/cw-graph.owl#> .
2 @prefix prx:
3    <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/proxy_flow_control.owl#> .
4 ...
5
6 cw:cw-sg-1 {
7
8   # Signature graph of ContentWatch
9   {
10    cw:cw-sig-1 a sig:Signature ;
11      sig:hasGraphSigningMethod cw:cw-gsm-1 ;
12      sig:hasSignatureValue "SXQncyBibHVlIGxpZ2h0" ;
13      sig:hasVerificationCertificate cw:cw-pck-1 .
14    ...
15  }
16
```

```
17   # Flow control rules of ContentWatch
18   cw:cw-rules-1 {
19     cw:wst-1 a tec:WebSite ; dul:hasQuality cw:uq-1 ; dul:hasSetting cw:alprs-1  .
20     cw:uq-1 a tec:URLQuality ; dul:hasRegion cw:ur-1  .
21     cw:ur-1 a tec:URLRegion ;
22       tec:hasURL "http://www.porntube.com/" ; dul:hasSetting cw:alprs-1 .
23     ...
24   }
25 }
```

Listing 4.3: Example of a signed Named Graph.

The Named Graph created by JusProg is depicted in Listing 4.4 and identified as `jp:jp-sg-1`. Its structure is similar to that of the Named Graph `cw:cw-rules-1` created by ContentWatch. JusProg provides its regulation details as the RDF graph `_:jp-rules-1` shown in line 15. This graph contains the flow control rule depicted in Figure 3.19c. Signing the graph results in the signature graph shown in lines 7 to 12. The Named Graph `jp:jp-sg-1` contains the signature graph as its annotation graph and the RDF graph `_:jp-rules-1` as its content graph.

```
1  @prefix jp:  <http://icp.it-risk.iwvi.uni-koblenz.de/policies/jp-graph.rdf#> .
2  ...
3
4  jp:jp-sg-1 {
5
6    # Signature graph of JusProg
7    {
8      jp:jp-sig-1 a sig:Signature ;
9        sig:hasGraphSigningMethod jp:jp-gsm-1 ;
10       sig:hasSignatureValue "SSBsaWtlIHRyYWlucw==" ;
11     ...
12   }
13
14   # Flow control rule of JusProg
15   _:jp-rules-1 { ... }
16 }
```

Listing 4.4: Example of a signed RDF Graph.

4.7.5. Signing Multiple and Distributed Graphs

The German comprehensive school receives the graph `cw:cw-sg-1` from ContentWatch and the graph `jp:jp-sg-1` from JusProg. As both graphs only provide flow control rules and not a complete flow control policy, the school adds its own RDF graph `_:cs-data-1` with additional regulation details. These details cover the flow control policy which includes the flow control rules and associates them with their legal authorization and organizational motivation as well as their enforcing proxy server. The policy corresponds to the example flow control policy depicted in Figure 3.19d. The school signs the graph `_:cs-data-1` together with the two graphs `cw:cw-sg-1` and `jp:jp-sg-1`, thereby signing multiple and distributed graphs at once (**RI.F.5**). This results in the Named Graph

cs:cs-sg-1 shown in Listing 4.5. It contains the school's graph _:cs-data-1 (lines 16 to 24), ContentWatch' graph cw:cw-sg-1 (lines 27 to 30), and JusProg's graph jp:jp-sg-1 (lines 33 to 36). The school's signature graph contains the used graph signing function cs:cs-gsm-1 (line 9), the created signature value (line 10), and the school's public key certificate cs:cs-pck-1 (line 11).

```
1  @prefix cs:  <http://icp.it-risk.iwvi.uni-koblenz.de/policies/cs-graph.owl#> .
2  ...
3
4  cs:cs-sg-1 {
5
6    # Signatue graph of the comprehensive school
7    {
8      cs:cs-sig-1 a sig:Signature ;
9        sig:hasGraphSigningMethod cs:cs-gsm-1 ;
10       sig:hasSignatureValue "QWxsIHlvdXIgYmFzZSBhcmU=" ;
11       sig:hasVerificationCertificate cs:cs-pck-1 .
12     ...
13   }
14
15   # Flow regulation details of the comprehensive school
16   _:cs-data-1 {
17     cs:alpp-1 a tec:FlowControlPolicyMethod ;
18       dul:hasMember cw:alpr-1 , cw:alpr-2 , jp:alpr-3 ;
19       dul:defines cs:ro-3 , cs:es-3 , cs:la-3 , cs:om-2 .
20     cs:es-3 a tec:EnforcingSystem ; dul:classifies cs:pr-1 .
21     cs:pr-1 a tec:ProxyServer ; dul:hasSetting cs:alpps-1 .
22     cs:alpps-1 a tec:FlowControlPolicySituation ; dul:satisfies cs:alpp-1 .
23     ...
24   }
25
26   # Graph data received from ContentWatch
27   cw:cw-sg-1 {
28     { ... }
29     cw:cw-rules-1 { ... }
30   }
31
32   # Graph data received from JusProg
33   jp:jp-sg-1 {
34     { ... }
35     jp:jp-rules-1 { ... }
36   }
37 }
```

Listing 4.5: Example of multiple signed graphs.

4.7.6. Signing Medical Data

The scenario for securing medical data records introduced in Section 2.2 covers three different CDOs which exchange medical data records with each other. The overall process of these transmissions is depicted in Figure 2.5 and involves a GP, a radiographer, and a hospital. Each CDO creates several records, compiles them into an EHR, signs

the EHR, and sends it to the next CDO. Figure 4.7 depicts the final EHR which is created by the hospital and identified as `hp:hp-ehr-1`. This graph contains the EMR `hp:hp-emr-1`, the EHR `hp:hp-ehr-1`, and the graph `_:hp-rls-1` which are created by the hospital as well as the EHR `rg:rg-ehr-1` which is received from the radiographer. The EMR `hp:hp-emr-1` stores the results of the hospitals examination, the EHR `hp:hp-ehr-1` contains several graphs describing the different steps of the performed surgery, and the graph `_:hp-rls-1` corresponds to the hospital's discharge note. The EHR `rg:rg-ehr-1` created by the radiographer contains the EMR `rg:rg-emr-1` which covers the results of the radiographer's examination and the graph `_:rg-dia-1` which compiles the associated diagnosis. In addition, the graph `rg:rg-ehr-1` also contains the EHR `gp:gp-ehr-1` which is received from the GP. This graph contains the EMRs `gp:gp-emr-1` and `gp:gp-emr-2` which provide the results of two different examinations conducted by the GP. The EHR also contains the graph `_:gp-prs-1` which covers the GP's prescription for iodine tablets. All three graphs are signed individually before including them into the EHR `gp:gp-ehr-1`.

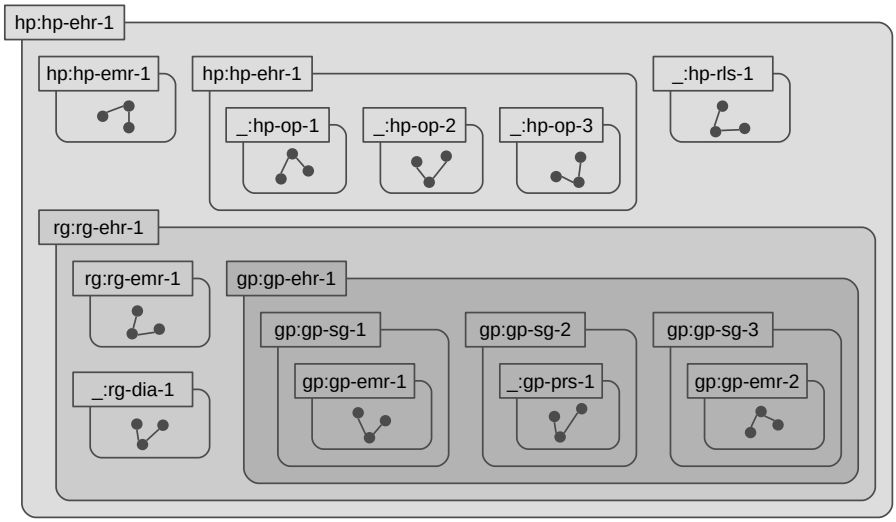


Figure 4.7.: Example graph signed by the hospital which contains medical data records from different CDOs.

All EMRs and EHRs of the scenario are provided as Named Graphs while all other graphs are associated with blank nodes as graph identifiers. EHRs are signed by their respective creators and are basically a collection of several other graphs without adding any additional triples. Thus, EHRs are directly used as output of the assembly function α_N and contain the other graphs as their content graphs. An additional signature graph stores the actual signature value. Listing 4.6 shows how the EHR `hp:hp-ehr-1` created by the hospital is encoded by using the extension of TriG for nested Named Graphs and blank nodes as graph identifiers. The EHR provides an example of signing multiple graphs at once (**RI.F.5**) as well as of iterative signing of graphs (**RI.F.4**). All

depicted graphs are either RDF graphs (**RI.F.1.1**) or Named Graphs (**RI.F.1.3**) and contain A-box knowledge (**RI.F.3.2**).

```
hp:hp-ehr-1 {

  # Signatue graph of the hospital
  { ... }

  # EMR of the hospital containing the results of the scintigraphy
  hp:hp-emr-1 { ... }

  # EHR of the hospital containing several graphs documenting the operation
  hp:hp-ehr-1 {
    { ... }
    _:hp-op-1 { ... }
    _:hp-op-2 { ... }
    _:hp-op-1 { ... }
  }

  # Graph containing the discharge note of the hospital
  _:hp-rls-1 { ... }

  # EHR of the radiographer including the EHR of the general practitioner
  rg:rg-ehr-1 {
    { ... }
    rg:rg-emr-1 { ... }
    _:rg-dia-1  { ... }

    # EHR of the general practitioner including the EMRs of two ultrasonographies
    gp:gp-ehr-1 {
      { ... }
      gp:gp-sg-1 { { ... } gp:gp-emr-1 { ... } }
      gp:gp-sg-2 { { ... }  _:gp-prs-1 { ... } }
      gp:gp-sg-3 { { ... } gp:gp-emr-2 { ... } }
    }
  }
}
```

Listing 4.6: Example of signed medical graphs.

4.8. Evaluation and Comparison with Existing Approaches

This section evaluates how the related work and state of the art discussed in Section 4.1 as well as the graph signing framework Siggi fulfill the requirements introduced in Section 4.2. The results of this assessment are depicted in Table 4.4. Although the related work presents different algorithms for signing graph data, only two approaches cover the whole signing process. These approaches are Tummarello et al. [295] and the XML signature standard [20]. Other approaches which focus on particular sub-functions of the signing process such as canonicalization functions and hash functions are not discussed as they cannot be directly used for achieving integrity and authenticity of graph data.

Similarly, alternative approaches for achieving integrity of graph data as presented in Section 4.1.7 are also not evaluated. These approaches include trusty URIs [184] and transmitting the graphs over SSL connections. As these approaches are designed for a different use case, they cannot be directly compared with the graph signing framework Siggi. In the following, Tummarello et al. and the XML signature standard are compared with Siggi along the functional requirements **RI.F.1.1** to **RI.F.5** and non-functional requirements **RI.N.1** to **RI.N.4**.

Table 4.4.: Comparison of the capabilities of different approaches for signing graph data with the requirements introduced in Section 4.2. Rows correspond to the different approaches and columns correspond to requirements. Requirements **RI.F.1.1** to **RI.F.5** are functional, while **RI.N.1** to **RI.N.4** are non-functional. The letter y corresponds to a complete fulfillment of the requirement, l corresponds to a partial fulfillment, and n corresponds to no fulfillment of the requirement.

	RI.F.1.1: Signing RDF(S) graphs	**RI.F.1.2**: Signing OWL graphs	**RI.F.1.3**: Signing Named Graphs	**RI.F.2.1**: Signing individual triples	**RI.F.2.2**: Signing arbitrary sets of triples	**RI.F.2.3**: Signing MSGs	**RI.F.2.4**: Signing entire graphs	**RI.F.3.1**: Signing T-box knowledge	**RI.F.3.2**: Signing A-box knowledge	**RI.F.4**: Iterative signing of graph data	**RI.F.5**: Signing multiple, distributed graphs	**RI.N.1**: Encoding-independent signature	**RI.N.2**: Configurability	**RI.N.3**: Modularity	**RI.N.4**: Implementation independence
Tummarello et al. [295]	y	y	n	l	n	y	y	y	y	l	n	y	n	n	y
XML Signature Standard [20]	y	y	y	y	y	y	y	y	y	y	y	n	l	y	y
Siggi	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

4.8.1. Evaluating the Functional Requirements

Signing RDF(S) graphs and OWL graphs is natively supported by Tummarello et al. The graph to be signed first must be split into disjoint MSGs which are then signed individually. The XML signature standard can be used for signing RDF(S) graphs and OWL graphs by first applying an XML-based serialization format to the graphs and

then signing the resulting document. RDF(S) graphs can be transformed into XML documents by using the serialization format RDF/XML [26] and OWL graphs can be serialized using OWL/XML [210]. Thus, both approaches fulfill requirements **RI.F.1.1** and **RI.F.1.2**. The graph signing framework Siggi supports these requirements by mapping RDF(S) graphs and OWL graphs to Named Graphs which can be directly signed with the framework. A Named Graph is uniquely identified by its URI and associates this URI with all its triples. Signing a Named Graph must therefore cover both the graph's triples as well as the link between these triples and the graph's URI. In order to sign a Named Graph with Tummarello et al., all MSGs of the graph must be associated with the graph's URI. As the approach does not natively support such a relation, it does not fulfill requirement **RI.F.1.3**. The XML signature standard can be directly used for signing Named Graphs by using TriX [75] as serialization format for the graph. TriX is an XML-based format which supports Named Graphs. The graph signing framework Siggi natively supports Named Graphs via the design of its formal specification.

The approach of Tummarello et al. is restricted to signing complete MSGs and can generally not be used for signing individual triples. An MSG is defined over blank nodes and contains one or more triples. Triples without blank nodes form an MSG on their own. Signing a single triple with Tummarello et al. is therefore only possible if the triple does not contain any blank nodes. This corresponds to a partial fulfillment of requirement **RI.F.2.1**. The structure of a graph and the number of blank nodes define how the graph is split into disjoint MSGs. As MSGs cannot flexibly be defined, arbitrary sets of triples cannot be signed with Tummarello et al. (**RI.F.2.2**). Signing an entire graph (**RI.F.2.4**) with multiple MSGs is possible by signing each of its MSGs individually. The XML signature standard does not have any restrictions on the triples to be signed. The standard can be used for signing single triples, arbitrary sets of triples, MSGs, and entire graphs. Similarly, the graph signing framework Siggi also supports signing different types of graph data. Triples which do not form a graph on their own, can be mapped to a new graph which can then be processed by the framework. T-box knowledge and A-box knowledge are syntactically represented as triples. These triples can be signed with any of the discussed approaches. Thus, Tummarello et al. and the XML signature standard as well as the graph signing framework Siggi fulfill requirements **RI.F.3.1** and **RI.F.3.2**.

Iterative signing allows to sign an already signed graph again which leads to a nested structure of graph signatures. In each iteration, the signing party may choose to countersign the already signed data or to add additional triples to the signed graph in order to sign them as well. Verifying a particular signature requires a clear distinction between signatures of different signing iterations. A signature created with Tummarello et al. is stored in six triples which are linked to the signed MSG by reifying one of the its triples. As these triples do not natively allow to distinguish between different signing iterations, Tummarello et al. do not fulfill requirement **RI.F.4**. In contrast, the XML signature standard can be directly used for iterative signing of graph data. A signature is stored in a new XML element `Signature` which is added to the XML serialization of the signed graph. This element precisely identifies the XML fragments which represent the signed graph data. The element can also identify other `Signature` elements as well.

This can be used for iteratively signing a graph together with additional triples. The graph signing framework Siggi supports iterative signing of graph data via the design of its assembly function α_N.

Tummarello et al. do not support signing multiple graphs at once (**RI.F.5**) as it focuses on signing individual MSGs only. Due to its design, the approach neither supports signing all MSGs of a single graphs at once nor signing all MSGs of multiple graphs at the same time. On the other hand, the XML signature standard can be used for signing multiple and distributed XML documents via the use of *detached signatures*. Such signatures allow to refer to the signed XML documents by their remote URI. Each XML document can contain one or more graphs, allowing the XML signature standard to sign multiple and distributed graphs as well. The graph signing framework Siggi fulfills requirement **RI.F.5** via the design of its combining function for graphs ϱ_N.

4.8.2. Evaluating the Non-Functional Requirements

Tummarello et al. and the graph signing framework Siggi do not rely on a particular serialization format of the graph data to be signed. Instead, they interpret the graph as an abstract data structure and thus fulfill requirement **RI.N.1**. In contrast, the XML signature standard requires an XML serialization of the graph. Although a graph can also be signed by signing a particular XML serialization of it, the resulting signature is inextricably linked to the signed XML document. It is attached to the signed document as a new `Signature` element which consists of plain XML data and does not contain any triples. If the serialization of the graph is lost, e. g., by loading the graph into a triple store, the signature can no longer be verified. Re-creating the particular serialization requires the same blank node identifiers and the same order of triples as used in the signed graph. Even if the serialization can be re-created, the `Signature` element may still be lost as it does not contain any triples which can be processed as graph data. Thus, the contents of this element cannot be stored with the graph when loading it into a triple store. In order to create a stable signature for a graph using the XML signature standard, a canonicalization function κ_N, a hash function for graphs λ_N, and an assembly function α_N must be used as defined in the graph signing framework Siggi. The canonicalization function and the hash function ensure that renaming blank node identifiers and re-ordering the triples does not invalidate the graph's signature. The assembly function ensures that the information for verifying the signature is provided as graph data which can also be processed together with the signed graph. None of these functions are natively supported by the XML signature standard and are only part of the graph signing framework Siggi. Thus, the XML signature standard does not fulfill requirement **RI.N.1**.

The graph signing framework Siggi provides a generic signature pipeline without relying on any particular algorithm. Instead, it can be configured with different algorithms and thus fulfills requirement **RI.N.2**. As the signature pipeline of the framework also has a modular design, it fulfills requirement **RI.N.3** as well. Similarly, the XML signature standard also provides a generic signature pipeline which supports different algorithms. However, in contrast to Siggi, the XML signature standard only supports one particular

assembly function α_N which stores all details for verifying the signature in an XML element. Thus, the XML signature standard only partially fulfills requirement **RI.N.2**. As the signature pipeline of the XML signature standard has a modular design, the standard fulfills requirement **RI.N.3** as well. In contrast, Tummarello et al. define specific algorithms for the signing process to be used. Thus, the approach neither fulfills requirement **RI.N.2** nor requirement **RI.N.3**. All three approaches focus on signing graph data on a conceptual level. As they do not rely on any particular software implementation, they all fulfill requirement **RI.N.4**.

4.8.3. Summary

Tummarello et al. do not fulfill all functional and non-functional requirements defined in Section 4.2. As the graph signing framework Siggi provides a generic framework of the signing process, the approach of Tummarello et al. can be integrated into Siggi and corresponds to configuration B of the framework as described in Section 4.4.2. In contrast, the XML signature standard fulfills almost all of these requirements except for requirement **RI.N.1** which corresponds to an encoding independent signature. The XML signature standard only supports signing XML documents and cannot be directly used for signing Semantic Web graphs. Even signing XML serializations of such graphs still requires the steps defined in the signature pipeline of Siggi. However, the graph signing framework Siggi reuses many of the concepts provided in the XML signature standard such as its modular signature pipeline which can be configured with various algorithms.

4.9. Limitations and Future Extensions

As demonstrated in the previous section, the graph signing framework Siggi fulfills all functional and non-functional requirements defined in Section 4.2. However, the framework only provides a formal specification of the signing process. In order to actually use the framework for signing graph data, it must be implemented in software or hardware. Furthermore, this implementation may be part of a particular environment or application context which requires additional security considerations. This section first describes the aspects that must be considered when using the graph signing framework Siggi in practice. Afterwards, possible extensions to the framework are discussed.

4.9.1. Reasoning on Signed Graph Data

Reasoning is the process of inferring additional data from existing data [11]. A Semantic Web reasoner uses T-box knowledge in order to interpret the provided A-box knowledge. The interpretation results in additional triples which may be included in the original A-box knowledge. Reasoning on signed data may invalidate the signature if it is no longer possible to distinguish between the original data and the data created through the reasoning process. In order to prohibit a reasoner from invalidating a graph's signature, it is necessary to store the inferred data separately from the original graph data. This

can be achieved by creating additional graphs for storing only the inferred triples. Triple stores such as Sesame [54] and Jena [74] provide mechanisms for implementing such a distinction. If it is also required to sign the inferred triples together with the original triples, iterative signing can be used. In this case, the graph which stores the inferred triples is signed together with the already signed graph that contains the original triples.

4.9.2. Security of the Graph Signing Framework

The main goal of the graph signing framework Siggi is to provide integrity and authenticity of graph data. In order to achieve these main objectives, the framework must be secure. The security of the graph signing framework Siggi depends on several aspects, including the cryptographic security of the algorithms used for a particular configuration, the security of the software implementation, and the security of the environment in which the software implementation is applied. The cryptographic security of the algorithms is discussed in Section 4.5. However, a cryptanalysis does not consider the context in which the algorithms are applied. Even though the used algorithms may be secure, it is still possible for a software implementation of the framework to contain implementation errors which may lead to security vulnerabilities. In order to assess the security of a particular software implementation, product-oriented methods such as [80, 66] can be used. These methods provide a framework for assessing the security of various IT products including software and hardware. A particular implementation of the graph signing framework may be part of a larger IT system. The security of such IT systems as well as their environment also influences the security of the actual signing process. Thus, it is necessary to evaluate the security of these systems and their respective environment to ensure that the framework's main objectives are fulfilled. The security of the environment can be evaluated by using holistic approaches such as [133, 59]. Important security aspects of the environment such as trust models and key management are discussed in Sections 4.9.4 and 4.9.3.

4.9.3. Key Management

The signature of a signed graph associates the graph with the owner of the secret signing key. Key management defines different organizational tasks for protecting the signing key as well as its corresponding public key from being compromised and misused by unauthorized parties. The tasks of key management include the generation of the key pair, its secure storage, its distribution, and its secure destruction [269, 217]. These tasks ensure that a secret signature key is only known to and used by its actual owner and that the corresponding public signature verification key can be related to this owner. Creating a key pair and storing the private key in a secure environment ensures that only authorized parties have access to the private key. Destroying old keys is necessary to prohibit a usage beyond their intended lifetime. Keys which are too old may not be secure anymore due to new attacks or greater computational power available to break the keys. Compromised keys must be revoked to prevent any further usage of them. The particular implementation of the individual key management tasks depends on the appli-

cation and environment in which the graph signing framework Siggi is used. Professional environments may have higher security requirements than private applications. Detailed guidelines for key management in professional environments are provided in [216, 217].

4.9.4. Public Key Infrastructure and Trust Model

Digitally signing graph data is a security mechanism for implementing authenticity of the signed data. The graph data is authentic if it is retrieved from an identified data source and if the identity of this data source is proven to be correct [28]. In order to prove the identity of the signing party, authenticity of graph data requires a connection between the secret signing key and its owner. Such a connection is created via a public key certificate which contains both the public key of the secret signing key and an identifier of its owner [4, 317]. Example identifiers are e-mail addresses or distinguished names [317]. Public key certificates are managed by public key infrastructures (PKIs) which ensure that the mapping provided by a public key certificate is actually correct. The management of public key certificates includes the creation and distribution of the certificates as well as their revocation [5]. A PKI consists of several services which include a certification authority (CA) and a registration authority (RA). The registration authority verifies the identity of a party and the party's ownership of a key pair [232, 4]. The verified mapping is then sent to the certification authority which issues the public key certificate [269].

A PKI only provides an infrastructure for managing public keys. The organizational processes for verifying the identities of certificate owners and issuing their certificates depend on the certificate authority. A trust model defines the conditions under which an entity trusts a public key certificate and its creation by a CA [6]. By signing a public key certificate, a CA states that the party identified in the certificate is actually the owner of the certificate's public key. Thus, the trustworthiness of a public key certificate depends on the trustworthiness of its issuing CA. Trust is basically a subjective assumption of a PKI user [6] that the CA behaves in such a way the user expects it to [290]. Different trust models provide different methods for managing trust between users and CAs of a PKI. Two widely used trust models for public key certificates are PGP [320] and X.509 [82]. X.509 organizes all CAs as a hierarchy with a few trusted root CAs which issue public key certificates for other CAs. These CAs may issue public key certificates to other CAs as well or provide public key certificates to end users. In the X.509 model, the trust of the hierarchy depends on the trust of the root CAs which are usually pre-configured as trust-worthy in most operating systems. In contrast, PGP has no hierarchy and allows its participants to be end users and CAs at the same time, resulting in a interconnected web of certificates. In the PGP model, trusting a CA is achieved by issuing a certificate directly or by manually selecting a trust-worthy certification path. Applying a particular trust model depends on the intended application. While X.509 may be used in professional environments, PGP is mostly sufficient for private use. An overview of different trust models and their characteristics is provided in [232, 6].

4.9.5. Secure Time Stamps

The examples provided in Section 4.7 demonstrate how the graph signing framework Siggi can be applied for signing graph data. The examples also include a possible signature graph providing the basic information about the signature verification process. However, the examples do not contain any additional data which may be relevant for the context in which the signature is used. Such data may include a time stamp that specifies when the signature was created. This can be used for defining the topicality of the signature and the data. A time stamp may be provided by a secure time stamping service which provides signed time stamp information [7]. If the time stamp is provided as graph data, it can be signed using the graph signing framework. The resulting graph can then be signed together with the actual graph data using the framework's iterative signing feature.

4.9.6. Alternative Assembly Functions

The assembly function α_N attaches a signature graph to a signed graph by embedding both graphs into a new Named Graph. In the examples of Section 4.7, the assembly function encodes this Named Graph by using an extended version of TriG [40] which supports nested Named Graphs and blank nodes as graph identifiers. However, the assembly function can also be built upon alternative encodings which are directly compatible with existing Semantic Web concepts such as RDF datasets [87]. An RDF dataset is similar to a Named Graph except that it does not support nested structures or blank nodes as graph identifiers. Thus, RDF datasets can be expressed with the regular TriG syntax. In order to encode a Named Graph as an RDF dataset, its nested structure must be mapped to a flat structure and blank nodes as graph identifiers must be eliminated. Removing the nested structure can be achieved by storing all subgraphs in different TriG documents so that the resulting documents contain only valid RDF datasets. The nested structure of the original graphs can then be expressed by creating additional triples which explicitly describe the graphs' nested structures. This can be done by using a property such as `hasSubGraph` to relate a graph to its corresponding subgraphs. Blank nodes used as graph identifiers can be eliminated by replacing them with URIs.

Another alternative encoding of nested Named Graphs is the use of N-Quads [71]. N-Quads is an extension of N-Triples which adds an optional additional context URI to each triple. The context URI corresponds to the URI of the graph containing the triple and allows N-Quads to be used for encoding Named Graphs. Similar to TriG, N-Quads does not support nested graphs or blank nodes as graph identifiers. In order to use N-Quads for encoding Named Graphs as defined in Equation 4.3, blank nodes first must be mapped to URIs and the nested structure must be transformed to a flat structure. This can be achieved by storing all subgraphs in separate N-Quads documents and introducing additional triples which describe the nested structure of the original graphs. As the graph signing framework Siggi does not rely on any particular assembly function α_N, both alternative encodings are already compatible with the framework.

4.10. Summary

This chapter has presented the graph signing framework Siggi which supports iterative signing of graph data. It allows to sign graphs at different levels of granularity, signing Named Graphs, and signing multiple and distributed graphs at once. It is based on a formal specification which divides the signing process into different steps. Each step can be implemented with various algorithms, allowing the framework to be configured for achieving different features such as minimum signature overhead or minimum runtime. The framework processes a graph as abstract data structure and does not rely on a particular encoding. Thus, a signature created with the graph signing framework is independent from the order of the triples in the graph, the local identifiers of the graph's blank nodes, and the used serialization format. As signing a graph achieves integrity and authenticity of the graph, the graph signing framework Siggi answers research questions **RQ.2** and **RQ.3**.

Chapter 5.

T-Store: Searching in Encrypted Graph Data

This chapter presents T-Store, an approach for searching in encrypted RDF graphs. T-Store restricts access to particular triples of a plaintext graph to authorized parties, i. e., only authorized parties are able to apply queries to the corresponding ciphertext graph. Unauthorized parties are not able to access any plaintext triples and even authorized parties can only retrieve triples which match a legitimate query. Thereby, T-Store achieves confidentiality of the plaintext graph and answers research question **RQ.1**. A fundamental design principle of T-Store is the distinction between a data owner and several users. The data owner possesses the plaintext graph and manages its access. To this end, the data owner encrypts the plaintext graph and sends the resulting ciphertext graph to the users. Users are authorized by the data owner to apply queries to the ciphertext graph. The design of T-Store requires only little communication between the data owner and the users which covers the distribution of the ciphertext graph and the exchange of query authorizations. Query processing is conducted offline by the users on their local systems and does not involve the data owner or any third party. T-Store supports a restricted set of SPARQL [292] queries of type `ASK`, `CONSTRUCT`, and `SELECT`. The basic concept of T-Store was first published in [171]. This chapter extends this basic concept with additional features, evaluates the performance of the extended approach, and conducts a detailed analysis of its cryptographic security.

The remainder of this chapter is organized as follows: The state of the art and related work for searching in encrypted data is summarized in Section 5.1. Based on this section and on the scenarios introduced in Chapter 2, the functional and non-functional requirements for T-Store are defined in Section 5.2. Section 5.3 provides a short overview of the concept for searching in encrypted graphs including a basic formalization. The basic formalization is then used to describe the details of T-Store's design in Section 5.5. The performance of T-Store is evaluated in Section 5.6 and Section 5.7 analyses its cryptographic security. Section 5.8 shows two different applications of the approach for searching in encrypted graphs which are based on the scenarios of Chapter 2. Section 5.9 discusses the state of the art and related work and compares it with T-Store. Limitations and possible improvements of T-Store are discussed in Section 5.10 before the chapter is concluded.

5.1. State of the Art and Related Work

The general process of searching in encrypted RDF graphs can be separated into five different steps as depicted in Figure 5.1. The first two steps prepare the plaintext graph for query processing. In the first step, the graph is encoded with a serialization format that can be used for querying. Possible formats are RDF/XML [26] and relational databases. RDF/XML transforms the graph into an XML document and a relational database stores the graph's triples in one or more database tables [99, 54]. In the second step, the encoded plaintext graph is encrypted to create a ciphertext graph. In order to apply a query to the ciphertext graph, each query is processed similarly to the plaintext graph. The third step transforms a SPARQL query [292] so that it can be applied to the encoded graph created in the first step. Possible encodings are XPath [252] for querying XML documents and SQL [160] for querying relational databases. The encoded query is encrypted in the fourth step so that it is compatible with the ciphertext graph created in the second step. The last step conducts the actual query processing by applying the ciphertext query to the ciphertext graph.

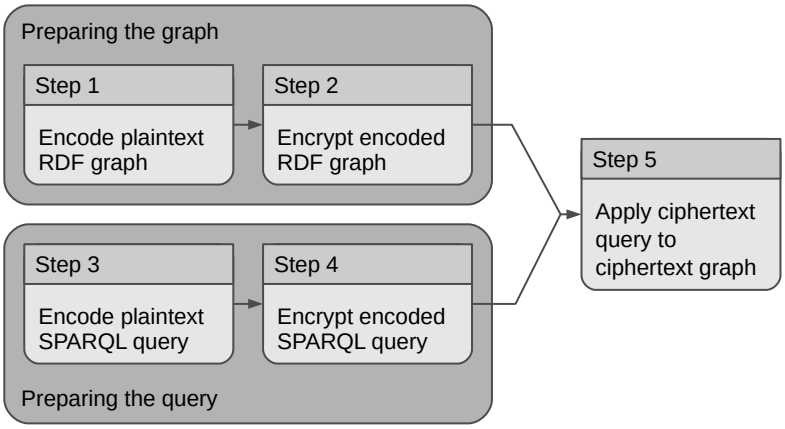


Figure 5.1.: The general process of searching in encrypted RDF graphs.

This section summarizes the state of the art and related work of searching in encrypted data. As far as the author knows, there is no approach yet which is specifically designed for searching in encrypted RDF graphs. Therefore, this section presents such approaches that operate on other data structures but can generally be used for RDF graphs as well by mapping them to the process depicted in Figure 5.1. The approaches are distinguished between the type of data on which they operate and support relational databases, XML documents, and graph structures. A detailed analysis of each approach, its mapping to the general process of Figure 5.1, and a comparison with T-Store is provided in Section 5.9. This section focuses on such approaches that can be used for applying generic SPARQL queries to encrypted RDF graphs. Approaches which only support very specific queries such as range queries are not discussed as they cannot be used for other types of queries as well. Examples of range queries are provided in [150, 188, 305].

Approaches which focus on document collections are also not discussed as they only retrieve complete documents and do not support searching within them. Examples of such approaches are presented in [107, 68, 234].

5.1.1. Searching in Encrypted Relational Databases

A relational database stores data records in tables which consist of columns, rows, and cells. A row contains a particular data record, a column represents an attribute, and a cell stores a record's value of a specific attribute. All approaches for searching in encrypted relational databases store the encrypted data records at an untrusted server. Depending on the approach, the records are encrypted row-wise or cell-wise. A row-wise encryption maps the complete record to a single ciphertext and a cell-wise encryption encrypts each attribute value separately. Many approaches focus on *data outsourcing* in which a user stores encrypted data at a potentially untrusted server managed by a third party [281]. The user is completely trusted and the server conducts most of the query processing. Data outsourcing is often implemented with *filtering and refining* [296] which processes queries in two steps. The user initiates a query by transforming it into a ciphertext query and sending it to the server. The server conducts a filtering step in which it applies the ciphertext query to the encrypted data. This leads to an encrypted superset of the actual query result which is sent back to the user. In a refining step, the user decrypts the preliminary query result and applies the plaintext query to the resulting plaintext data to retrieve the final query result.

Filtering and refining is used by Hacıgümüş et al. [137] as well as Wang et al. [309]. Both approaches allow to search for all ciphertext records which match a set of attribute values. Hacıgümüş et al. use row-wise encryption and index all ciphertext records based on their attribute values with *bucketization*. Bucketization splits the value space of each attribute into disjoint buckets. The server stores a mapping from a ciphertext record to the buckets of its attributes and the user associates each plaintext value with their corresponding buckets. In order to search for particular attribute values, the user maps the values to their respective bucket identifiers and sends them to the server. The server returns all ciphertext records for the specified buckets which are then further processed by the user in a refining step. Wang et al. use cell-wise encryption and associate each ciphertext with a hash value. The hash value is used as an index and stored together with the ciphertext at the server. In order to search for all data records with particular attribute values, the user computes their hash values and sends them to the server which returns all matching ciphertext records. As the used hash function produces collisions, the received records form a superset of the actual query result which is further refined by the user.

Exact query results without refining are returned by Z. Yang et al. [316], Elovici et al. [98], and Evdokimov and Günther [104]. All three approaches use probabilistic, cell-wise encryption which maps identical plaintexts to different ciphertexts. Z. Yang et al. and Elovici et al. support queries that retrieve all ciphertext records with a particular attribute value whereas Evdokimov and Günther support keyword queries with an arbitrary keyword. Z. Yang et al. use trapdoors [42] to check if a particular cipher-

text record matches a query. Each plaintext is mapped to a tuple of two ciphertexts which are stored at the server. The first ciphertext encrypts the actual plaintext and the second ciphertext encrypts the first ciphertext with the plaintext and the name of its corresponding attribute. A user initiates a query by creating a trapdoor from the queried value and the name of its attribute and sends it to the server. The server tries to decrypt the second ciphertext of each tuple with the trapdoor. If the result is identical to the tuple's first ciphertext, the corresponding ciphertext record is returned to the user. Evdokimov and Günther propose a similar approach which is also based on trapdoors but supports keyword queries instead of attribute queries. Elovici et al. create an index tree for each column and store it together with the encrypted cell values at the server. A user applies an attribute query by iteratively traversing the index tree and sending corresponding messages to the server. In each iteration, the user receives an encrypted tree node, decrypts it, and requests the next node. This process is repeated until the user has received all matching tree nodes which are then used to retrieve the desired ciphertext records.

Exact queries and multiple users are supported by CryptDB [235], Probabilistic Randomly Partitioned Encryption (Prob-RPE) [259], and Y. Yang et al. [315]. CryptDB supports different types of queries, Y. Yang et al. support keyword queries, and Prob-RPE supports range queries and attribute queries for a single attribute. CryptDB encrypts all cell values of a column individually with the same encryption key. Each column is encrypted as multiple layers which provide different security and functionality. Processing a query may require to permanently remove an encryption layer from a queried column. A trusted proxy server authorizes querying users, maps their queries to ciphertext queries, and sends them to the database server. The proxy server also stores all encryption keys and can remove encryption layers. The database server executes ciphertext queries and sends the encrypted results to the proxy server which are then decrypted and forwarded to the querying user. Prob-RPE divides all values of a column into random buckets and probabilistically encrypts all values of a bucket individually. Identical plaintext values are mapped to different tuples consisting of a bucket identifier and a ciphertext. This mapping is stored at a trusted proxy server which authorizes all users, intercepts and transforms their queries, and stores all encryption keys. If a plaintext value is mapped to multiple tuples, the proxy server requests all tuples from the database server. Y. Yang et al. use a row-wise, probabilistic encryption. Each user is authenticated with a public key pair which is registered at the database server. The server associates ciphertext records with encrypted keywords. In order to apply a query, a user digitally signs the requested keyword with her private key and sends it to the server. The server verifies the signature and thereby removes any user-specific parts from the keyword which is then used to retrieve all matching ciphertext records.

In summary, all presented approaches for searching in encrypted relational databases can generally be used for searching in encrypted RDF graphs as well. This can be accomplished by storing a graph in a single database table with three different columns. The columns represent the subject, predicate, and object of the graph's triples. The presented approaches can then be directly applied to the resulting database.

5.1.2. Searching in Encrypted XML Documents

XML documents [49] store data in a hierarchically organized tree of elements. The tree has a single root element and can be divided into subtrees with their own root. Each element has a name and can optionally have attributes and data values. The set of all possible element names and attribute names are restricted by the document's schema [124]. A path is a sequence of connected elements in the tree, starting at the root and ending at a particular element. Approaches for searching in encrypted XML documents generally support path queries, element queries, and subtree queries. Path queries match a path against the XML tree and return the elements at the end of the path. Each element in the path may be further specified by its attribute values. Element queries retrieve individual elements with particular attribute values and data values. Subtree queries return individual subtrees based on the query parameters.

Approaches based on filtering and refining are Lin and Candan [191, 190], Jammalamadaka and Mehrotra [165], and Order Preserving Encryption with Splitting and Scaling (OPESS) [304]. Lin and Candan support path queries by traversing the XML tree. Each element is stored in multiple buckets, encrypted individually, and stored at the server together with two indexes. The first index lists all ciphertexts for each bucket and the second index maps an element to its child elements and their buckets. A user initiates a query by requesting all child elements of the root element and their buckets. The user randomly selects one bucket for each child element, retrieves all its ciphertexts, and selects the next element in the queried path. The process is repeated until the user has retrieved all query results. Jammalamadaka and Mehrotra support element queries which retrieve elements based on their attribute values. The approach is similar to Hacıgümüş et al. and splits all attribute values into disjoint buckets. Each element is encrypted together with its attribute values and indexed with their bucket identifiers. Storage, index management, and query processing is identical to Hacıgümüş et al. OPESS supports path queries based on the names and data values of the path's elements. All subtrees of an XML tree are encrypted individually and indexed with structural and value-based information. The structural index maps all subtrees to their encrypted root element and the value index maps all encrypted data values to the subtrees in which they occur. The user prepares a query by encrypting its element names and data values and sends it to the server. The server uses its indexes to retrieve all matching subtrees and sends them to the user who conducts a refining step to extract the desired elements.

Exact query results without refining are provided by SemCrypt [274], Bouganim et al. [46], and two approaches of Brinkman et al., which are referred to as Brinkman 1 [52] and Brinkman 2 [53]. Bouganim et al. allow multiple users whereas the other approaches focus on data outsourcing. Brinkman 1 supports subtree queries which retrieve all subtrees with a particular element. The XML tree is encoded as hierarchically structured polynomial by mapping all element names to integers. The polynomial is randomly split into two parts which are stored at the user and the server, respectively. A user searches for an element by sending its integer representation to the server. Starting at the root element, the server iteratively computes the value of its polynomial and returns the result. The user computes its own polynomial, combines it with the server's value and

compares it with a predefined value. If both values are equal, the queried element is part of the subtree whose polynomial was just calculated. Brinkman 2 supports path queries which are based on trapdoors and consist of element names, attribute values, and data values. Encryption and query processing is similar to Z. Yang et al. SemCrypt supports path queries based on element names, attribute values, and data values. The user maps each path in the XML tree to a unique identifier which is used in a structural index and a value index stored at the server. The structural index maps a path identifier to an encrypted data value and the value index associates a data value with all elements that have the same data value. A user initiates a path query by transforming it into a sequence of ciphertext queries for each step in the path. Each ciphertext query is sent to the server and evaluated by applying the two indexes. Bouganim et al. support arbitrary queries and require a secure processing unit for every user. The XML document is split into subtrees which are encrypted with different encryption keys. An access policy is created for each user and associated with several encryption keys. An authorized user receives the encrypted document and her secure processing unit receives the access policy and the encryption keys. The unit processes the user's queries after having decided whether a query is allowed or prohibited by evaluating the access policy.

In summary, many approaches for searching in encrypted XML documents focus on queries which evaluate the document's structure such as path queries and subtree queries. These queries cannot be directly used for searching in encrypted RDF graphs. Although such a graph can easily be encoded as XML document, the actual contents of a triple are either stored as attribute values or data values. Storing the contents of a triple as element names is not possible as the set of possible element names is restricted by the document's schema. However, element queries can be used for searching in RDF graphs as they can contain data values like a URIs and literals as query parameters.

5.1.3. Searching in Encrypted Graph Structures

A graph is an abstract data structure which can be used for storing different types of data such as text documents and RDF triples. A graph generally consists of nodes which are connected via edges and can be split into subgraphs by removing some of the nodes and edges. Depending on the type of graph, the graph may also have additional characteristics. A labeled graph associates its nodes and/or edges with a name. In a bipartite graph, the nodes are divided into two disjoint sets and the edges connect two nodes from each set. This section summarizes three different approaches for searching in encrypted graph data. All approaches store the encrypted graph and an encrypted index at an untrusted server. The server processes queries from a single user by applying the index to the ciphertext graph. Privacy-Preserving Graph Query (PPGQ) [69] supports subgraph queries in encrypted graph collections which retrieve all graphs with a particular subgraph. Each graph is indexed with a feature vector which lists all its subgraphs. The feature vector is encrypted and stored together with the encrypted graph at the server. A user initiates a subgraph query by mapping it to a feature vector, encrypting it, and sending it to the server. The server combines the received feature vector with the feature vector of each encrypted graph. If the result matches a predefined comparison

value, the encrypted graph is sent to the user. Chase and Kamara [78] provide a generic indexing mechanism to support different queries on encrypted labeled graphs. The index essentially maps one or two query parameters to an encrypted set of query results. A user initiates a query by computing a query key from the query parameters and sending it to the server. The server processes the query by applying the query key to the index. Chase and Kamara apply their index to support neighbor queries, subgraph queries, and queries that retrieve all edges between two nodes. CryptGraph [314] supports subgraph queries on bipartite graphs. A graph is represented by its adjacency matrix which is encrypted using probabilistic homomorphic encryption. Homomorphic encryption [111] allows arithmetic operations on ciphertexts without decrypting them first. A subgraph query combines several atomic queries which ask whether or not two nodes are directly connected. An atomic query is a polynomial and is evaluated with the adjacency matrix. A user initiates a subgraph query by creating the polynomials for all atomic queries and sending them to the server. The server applies the polynomials to the adjacency matrix and sends the resulting value to the user who decrypts it to retrieve the actual query result.

In summary, some approaches for searching in encrypted graph structures focus on subgraph queries and can only partially be used for searching in RDF graphs. Subgraph queries basically correspond to SPARQL `ASK` queries and determine whether or not a graph contains a set of triples. However, SPARQL also supports `SELECT` queries which are used more often in practice than `ASK` queries [203, 14, 233].

5.1.4. SPARQL Query Language

SPARQL [292] is a W3C-standardized query language for RDF graphs and is supported by several RDF triple stores such as Sesame [54] and Jena [74]. The syntax and design of SPARQL is inspired by SQL [160]. A SPARQL query consists of a *query form* and a *query algebra* and is applied to an *RDF dataset*. The query form defines the type of the query and the format of the query result. Possible query types include `SELECT`, `CONSTRUCT`, and `ASK`. A `SELECT` query returns a list of individual query results whereas a `CONSTRUCT` query returns a new graph created from these results. An `ASK` query determines whether or not the query algebra matches against the queried dataset and returns a boolean value. The query algebra corresponds to the `WHERE` clause of a query. It defines the query's matching conditions and consists of several *triple patterns*. A triple pattern is similar to a triple but may contain an unbound *query variable* at subject, predicate, and/or object position. Considering that each position in a triple pattern can be either bound or unbound results in eight different *variants* of the same pattern. Triple patterns of a query algebra are combined into graph patterns. The simplest form of a graph pattern is a *basic graph pattern* which is a collection of an arbitrary number of triple patterns. More complex graph patterns such as unions of basic graph patterns or optional basic graph patterns are also possible. Finally, an RDF dataset [87] is a collection of several graphs. It is similar to a Named Graph as defined in Section 4.3.1 except that it does not allow nested graphs and blank nodes as graph identifiers.

A SPARQL query is evaluated by matching the query algebra against the queried RDF dataset. This process replaces the query variables in the algebra's triple patterns with corresponding URIs, blank nodes, or literals. A mapping from a single query variable to its corresponding value is referred to as a *variable binding* [292]. A *solution mapping* is a set of variable bindings and corresponds to a single solution of a triple pattern or a basic graph pattern. It is created by matching a pattern against a graph and replacing all query variables in the pattern with corresponding URIs, blank nodes, or literals. Several solution mappings are combined into a *solution sequence* which contains all solutions of a triple pattern or a basic graph pattern. Solution sequences may either be returned as the query result or only used during query processing. If the query algebra contains multiple triple patterns sharing the same query variables, their solution sequences are combined with a *join* operation. A join can only be conducted on two solution sequences if they are compatible with each other. This is the case if their solution mappings share identical variable bindings, i. e., if their query variables are mapped to identical values. The result of a join on two solution sequences is a new solution sequence which contains the combination of all compatible solution mappings.

An example SPARQL query of type `SELECT` is depicted in Listing 5.1. The query returns the name of all persons with the e-mail address `tdurden@example.com`. The namespace `foaf` corresponds to the FOAF vocabulary [50] for describing social networks. The query contains a single basic graph pattern with two triple patterns. The solution sequences of both triple patterns are joined using the variable bindings of the query variable `?person`. Line 2 states that the bindings of the variable `?name` are returned as the query result while the bindings of `?person` are only used during query processing.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name
3 WHERE {
4    ?person foaf:name ?name .
5    ?person foaf:mbox "tdurden@example.com" .
6 }
```

Listing 5.1: Example SPARQL query.

5.2. Requirements for Searching in Encrypted Graphs

T-Store allows to apply different types of SPARQL queries to encrypted RDF graphs. The queries are executed locally by authorized users without involving a server. Based on these general objectives, this section defines the specific requirements for T-Store. The requirements are distinguished between functional (**RC.F.***) requirements and non-functional (**RC.N.***) requirements. As defined in Section 3.2, functional requirements define the functions that a system must provide and non-functional requirements describe how functional requirements are implemented [282]. The following requirements are based on the scenario for regulating Internet communication presented in Section 2.1 and on the related work for searching in encrypted data summarized in Section 5.1. T-Store must fulfill the following functional requirements:

RC.F.1: Applying SPARQL triple patterns
T-Store must support queries which consist of a single SPARQL triple pattern with none, one, two, or three query variables. Such queries are the most basic types of queries and are the foundation for creating more complex queries.

RC.F.2: Applying SPARQL basic graph patterns
The approach must support queries which consist of a single SPARQL basic graph pattern. A basic graph pattern contains an arbitrary number of triple patterns. If the triple patterns share the same query variables, executing the basic graph pattern requires the computation of a join operation. Thus, this requirement also implies the support of join operations. In the scenario, the German comprehensive school applies queries to the encrypted log files of its proxy server.

RC.F.3: Supporting SPARQL query forms
T-Store must support the three different SPARQL query forms `SELECT`, `ASK`, and `CONSTRUCT`. The query forms define the format of the query result. Supporting the three different query forms is necessary as they are most frequently used in many SPARQL queries [14, 233].

RC.F.4: Supporting dynamic query authorizations
T-Store must support the authorization of particular queries after the plaintext graph has been encrypted. This allows a more flexible query authorization than predefining all supported queries when encrypting the plaintext graph. In the scenario, an investigator is authorized to apply queries to the log file after it has been encrypted by the proxy server.

RC.F.5: Separating the data owner from authorized users
The approach must distinguish between a data owner who encrypts the plaintext graph and users who can apply queries to the encrypted graph. The data owner can access the complete graph and possesses all encryption keys. In contrast, users are explicitly authorized by the data owner to apply particular queries to the graph. In the scenario, the school's administration (SA) and the parents' association (PA) collectively authorize an investigator to apply particular queries.

RC.F.6: Supporting query templates
T-Store must support the authorization of specific queries and query templates. A query template represents a group of similar queries. It corresponds to an incomplete query which must be further refined by the querying user in order to receive a complete query. Query templates reduce the communication overhead between the data owner who encrypts the plaintext graph and the users who query the ciphertext graph. Thus, this requirement implies requirement **RC.F.5**.

In addition to these functional requirements, T-Store must also support the following non-functional requirements:

RC.N.1: Processing queries offline

T-Store must not rely on a server which processes the queries, even if the server operates only on encrypted data. Instead, users must be able to process all authorized queries on their local systems after having received a corresponding query authorization and the ciphertext graph. In the following, this type of query processing is referred to as an *offline approach*. In the scenario, the authorized investigator applies the queries locally at her own computer system after having received a copy of the encrypted log files.

RC.N.2: Eliminating trusted systems

T-Store must not involve a trusted system which processes queries on behalf of authorized users and which can access the query results or parts of them. Instead, query results must only be accessible to the user who initiates a query. Examples of trusted systems are secure processing units, which are located at the user side, and proxy servers, which act as an intermediary between an authorized user and a server.

RC.N.3: Providing exact query results

Conducting queries with T-Store must not reveal any triples of the plaintext graph which are not part of the final query result. Thus, authorized users must receive exactly those triples of the plaintext graph which satisfy their queries. This requirement is necessary as otherwise an authorized user may receive more plaintext triples than the authorization intends. In the scenario, the investigator must only receive the graph data which is being investigated.

RC.N.4: Prohibiting data distinguishability

T-Store must encrypt the plaintext graph in such a way that the resulting ciphertext graph does not reveal the amount of identical plaintext URIs, blank nodes, or literals and their individual positions in the graph. More specifically, it must be computationally hard to identify identical plaintext URIs, blank nodes, or literals as such. This requirement is called *data indistinguishability* [167]. It is fulfilled if there is no algorithm which decides in polynomial time whether or not two ciphertexts represent the same plaintext value. Fulfilling this requirement is necessary in order to prevent an attacker from analyzing the ciphertext graph and use it to infer any information about the plaintext graph.

RC.N.5: Concealing graph characteristics

The ciphertext graph created by T-Store must not reveal any characteristics of the corresponding plaintext graph. The characteristics of a graph can generally be distinguished between local characteristics and global characteristics. Local characteristics cover individual plaintext triples and their respective parts. Such characteristics include the string length of all URIs, blank nodes, and literals in the graph (**RC.N.5.1**). Global characteristics depend on the whole graph and not on individual triples. Such characteristics include the density and the connectivity of the plaintext graph (**RC.N.5.2**) as well as the number of different plaintext

triples (**RC.N.5.3**). The density indicates how the individual triples of a graph are connected with each other, i. e., how many independent subgraphs a graph contains. Similar to requirement **RC.N.4**, fulfilling this requirement prevents an attacker from analyzing the ciphertext graph and inferring any information about the corresponding plaintext graph.

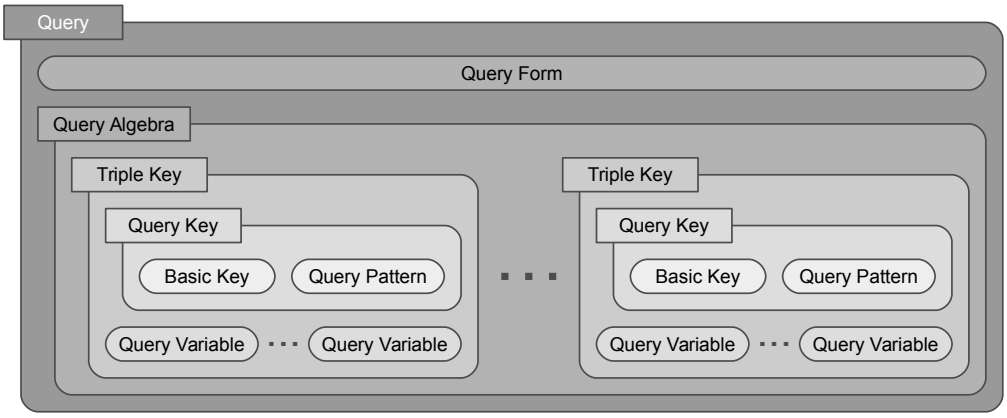
The non-functional requirements **RC.N.3** to **RC.N.5.3** cover the security of T-Store and affect the confidentiality of the plaintext graph. If these requirements are not fulfilled, information about the plaintext triples may be revealed to unauthorized parties. Section 5.9 describes how T-Store fulfills the functional and non-functional requirements and provides a comparison with the state of the art and related work.

5.3. Basic Terminology and Solution Overview

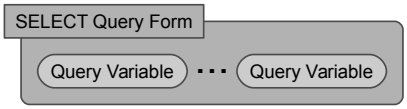
This section provides a general overview of the design and functions of T-Store. First, a representation of SPARQL queries in T-Store as well as the used data structures and cryptographic keys is given. A formalization of the terminology is provided in Section 5.4. Subsequently, the different cryptographic operations of preparing and processing queries in T-Store are introduced. T-Store allows to search in encrypted RDF *graphs*. An RDF graph consists of several triples and is owned by a *data owner*. The data owner encrypts a plaintext graph by encrypting all its triples individually and creates an additional *index* for the resulting ciphertext graph. The ciphertext graph and its index are then published on the web. *Users* are authorized by the data owner to apply *queries* to the ciphertext graph. A query allows a user to specifically access those triples of the plaintext graph which satisfy the query. Users are authorized by receiving *authorization keys* from the data owner. They combine the authorization keys with self-defined *user patterns* to create *query keys*. *Query functions* apply query keys to a ciphertext graph and its index and search for all matching triples. The index allows it to quickly identify all triples in the ciphertext graph that match a particular query key.

5.3.1. Representing SPARQL Queries in T-Store

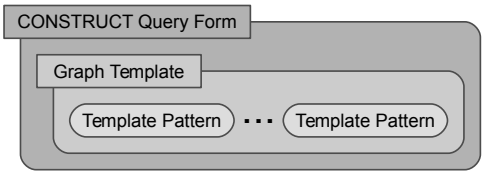
T-Store supports the execution of SPARQL queries of type `SELECT`, `CONSTRUCT`, and `ASK` on a ciphertext graph. The supported query algebra is restricted to a single basic graph pattern which consists of an arbitrary number of triple patterns. Figure 5.2 depicts how the supported SPARQL queries are represented in T-Store. The representation distinguishes between a *query form* and a *query algebra*. The query form defines the type and format of the query result and the query algebra defines the matching conditions. When processing a particular query, the query algebra is first applied to the queried ciphertext graph. This results in a single solution sequence which is further refined by the query form in order to create the final query result. As depicted in Figure 5.2a, the query algebra consists of several *triple keys* each of which represents a single SPARQL triple pattern. A triple key contains all information for creating the solution sequence of its corresponding triple pattern. In particular, a triple key associates a query key with



(a) Representing the query algebra in T-Store.



(b) Query form of `SELECT` queries.



(c) Query form of `CONSTRUCT` queries.

Figure 5.2.: Representing a SPARQL query in T-Store. A query has a query form and a query algebra. The query form distinguishes between `SELECT` queries, `CONSTRUCT` queries, and `ASK` queries. The query form of `SELECT` queries defines a list of query variables, `CONSTRUCT` queries require a graph template with template patterns, and `ASK` queries do not need any additional parameters. The query algebra is identical for all types of queries and consists of multiple triple keys. A triple key associates a query key with several query variables. A query key combines a basic key with a query pattern.

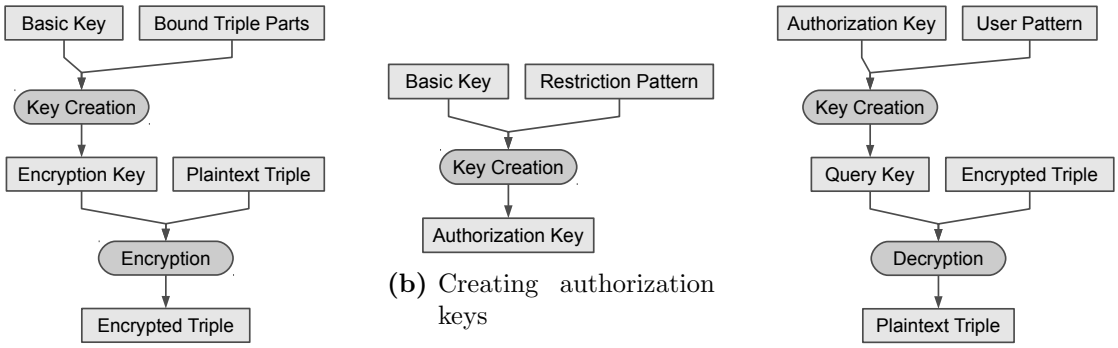
a respective number of *query variables*. The query key defines the matching condition of the triple key and the query variables are used to create the variable bindings of all successful matches. A query key is created from a *query pattern* and a *basic key*. The query pattern consists of bound and unbound parts which determine the input and output of a SPARQL triple pattern, respectively. I. e., the bound parts define the values that are being matched when applying a query key to a ciphertext graph and the unbound parts are returned for each successful match. Similar to a SPARQL triple pattern, there are eight different *variants* of a query pattern in T-Store. A basic key defines the position of the bound and unbound parts in the query pattern and determines its variant. T-Store uses eight different basic keys to distinguish between the different variants of a query pattern. The query variables of a triple key associate the unbound parts of the query pattern with a name in order to support join operations. Join operations combine compatible solution sequences of different triple keys of the same query algebra into a single solution sequence.

The refinement that is conducted by the query form depends on the type of the query. The query form of `SELECT` queries is depicted in Figure 5.2b. It defines a list of query variables whose variable bindings shall be returned as the final query result. The variable bindings are taken from the solution sequence created by the query algebra. Figure 5.2c shows the query form of `CONSTRUCT` queries, which return a new graph as the query result. The query form defines a *graph template* which contains an arbitrary number of *template patterns*. A template pattern is similar to a triple except that it may contain a query variable at subject position, predicate position, or object position. When processing a `CONSTRUCT` query, the query variables in the graph template are replaced by corresponding variable bindings in order to create the new graph to be returned. Again, the variable bindings are taken from the solution sequence of the query algebra. Finally, `ASK` queries only return a boolean value. Thus, their query form does not define any additional parameters for refining the solution sequence created by the query algebra.

5.3.2. Preparing and Applying Queries in T-Store

Preparing and processing a SPARQL query in T-Store requires different cryptographic operations. These operations are essentially implemented by query keys whereas triple keys are only used to support join operations. When applying the query algebra of a SPARQL query to a ciphertext graph, all its triple keys are processed individually and their individual solution sequences are combined with a join operation. A single triple key is applied to the graph by processing its query key and associating the result with its query variables. Query keys can directly be applied to ciphertext graphs. This section provides an overview of the different cryptographic operations used by T-Store for preparing and processing a query on a ciphertext graph. Figure 5.3 shows the different operations for encrypting a plaintext graph and querying the resulting ciphertext graph with a single query key. The operations must be conducted for all query keys in the query algebra. As depicted, the operations are divided into three different phases.

The first phase is shown in Figure 5.3a. In this phase, the data owner chooses eight different basic keys for each of the eight variants of a query pattern. The basic key as well as the bound parts of a query pattern are combined into an *encryption key*. Encryption keys are used for encrypting all triples in the plaintext graph individually for each of the eight variants of a query pattern. The second phase is shown in Figure 5.3b and covers the creation of an authorization key which combines a basic key and a *restriction pattern*. The basic key defines the query pattern variant, i. e., the number and position of the bound and unbound parts of a query pattern. The restriction pattern is created by the data owner and further restricts the number of possible user patterns that a user can define for querying the ciphertext graph. To this end, the data owner predefines some of the bound parts of a query pattern in the restriction pattern. Thus, the resulting authorization key already contains some of the required query parameters. The authorization key is sent to the user via a secure communication channel. The third phase covers the creation and application of a query key and is depicted in Figure 5.3c. The user creates a query key based on the received authorization key by adding a user pattern which contains additional query parameters. The resulting query key encodes all bound and



(a) Encrypting triples

(b) Creating authorization keys

(c) Applying query keys

Figure 5.3.: Overview of the computations for searching in encrypted graphs. The overview shows the different keys and cryptographic operations involved in the process.

unbound parts of a particular SPARQL triple pattern. The bound parts are either taken from the user pattern or from the data owner's restriction pattern which is embedded into the authorization key. The combination of the user pattern and the restriction pattern corresponds to a query pattern. Thus, a query key combines a query pattern with a corresponding basic key. If an encrypted triple can successfully be decrypted using a query key, the triple matches the encoded query pattern. If the decryption fails, the triple does not match the pattern. The decryption of an encrypted triple is successful iff the query key is identical to the encryption key. This is the case if the combination of the data owner's restriction pattern and the user pattern correspond to the bound triple parts of the encryption key. Section 5.4 formally defines the basic data structures and cryptographic keys used by T-Store. A detailed description of the approach is given in Section 5.5.

5.4. Basic Formalization

This section provides a basic mathematical formalization of the queries, data structures, and cryptographic keys of T-Store as introduced in the previous section. This formalization is used in Section 5.5 to describe the detailed process of querying encrypted graphs.

5.4.1. Plaintext Graphs and Plaintext Triples

A plaintext RDF graph G is a finite set of triples t. As described in Section 4.3.1, the set of all plaintext triples t is defined as follows:

$$\mathbb{T} := (\mathbb{R} \cup \mathbb{B}) \times \mathbb{R} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}) \tag{4.1}$$

$\mathbb{R}$ corresponds to the set of all resource URIs, $\mathbb{B}$ is the set of blank nodes, and $\mathbb{L}$ is the set of literals. It is $t = (s, p, o)$ with $s \in \mathbb{R} \cup \mathbb{B}$ being the subject of the triple, $p \in \mathbb{R}$

being the predicate, and $o \in \mathbb{R} \cup \mathbb{B} \cup \mathbb{L}$ being the object. A plaintext graph consisting of m triples is defined as $G = \{t_1, t_2, \ldots, t_m\}$ with $m \in \mathbb{N}$. The set of all plaintext graphs is defined as follows:

$$\mathbb{G} := \mathcal{P}(\mathbb{T}) = \mathcal{P}((\mathbb{R} \cup \mathbb{B}) \times \mathbb{R} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L})) \tag{4.2}$$

5.4.2. Encrypted Graphs and Encrypted Triples

Encrypting a plaintext triple t results in an encrypted triple c. An encrypted triple is a tuple of eight bit strings. Each bit string is used for one of the eight variants of a query pattern. Thus, an encrypted triple c contains eight different ciphertexts of the same plaintext triple t for each of the eight query pattern variants. The set of all possible encrypted triples is defined as follows:

$$\mathbb{T}_C := \underbrace{\{0,1\}^* \times \{0,1\}^* \times \ldots \times \{0,1\}^*}_{8 \text{ times}} \tag{5.1}$$

It is $c = (c_{\texttt{---}}, c_{\texttt{+--}}, c_{\texttt{-+-}}, c_{\texttt{--+}}, c_{\texttt{++-}}, c_{\texttt{+-+}}, c_{\texttt{-++}}, c_{\texttt{+++}})$. The indices `+` and `-` state if the corresponding part of the query pattern is bound or unbound. For example, the bit string $c_{\texttt{-+-}}$ supports query patterns with a bound predicate that retrieve tuples of subjects and objects for each matching triple. Encrypting a plaintext graph G results in an encrypted graph $G_C = \{c_1, c_2, \ldots, c_m\}$. The set of all encrypted graphs is defined as follows:

$$\mathbb{G}_C := \mathcal{P}(\mathbb{T}_C) = \mathcal{P}(\underbrace{\{0,1\}^* \times \{0,1\}^* \times \ldots \times \{0,1\}^*}_{8 \text{ times}}) \tag{5.2}$$

5.4.3. Basic Keys

A basic key $bk \in \mathbb{K}_b$ is a bit string of length $d \in \mathbb{N}$ and is used for encrypting the triples t of a plaintext graph G for a particular query pattern variant. The data owner choses eight different basic keys for each pattern variant which are identified as $bk_{\texttt{---}}$, $bk_{\texttt{+--}}$, $bk_{\texttt{-+-}}$, $bk_{\texttt{--+}}$, $bk_{\texttt{++-}}$, $bk_{\texttt{+-+}}$, $bk_{\texttt{-++}}$, and $bk_{\texttt{+++}}$. Each of these keys is used for creating a particular bit string of the encrypted triples c. For example, the basic key $bk_{\texttt{-+-}}$ is used for creating the bit strings $c_{\texttt{-+-}}$. The set of all basic keys is defined as $\mathbb{K}_b \subset \{0,1\}^d$.

5.4.4. Query Keys, Query Patterns, and Authorization Keys

A query key $qk \in \mathbb{K}_q$ is a bit string of length $d \in \mathbb{N}$ which encodes the bound and unbound parts of a query pattern. It is applied to an encrypted graph and may contain an unbound subject, predicate, and/or object. The type of a query key is defined by using the symbols `+` and `-` which mark the bound and unbound parts, respectively. For example, a query key of type `+++` encodes a bound subject, a bound predicate, and a bound object. It can be used for SPARQL `ASK` queries and determines whether or not the specified triple is part of the queried graph. Applying a query key returns a set of all matching values. For example, applying a query key of type `-++` returns a set of subject

URIs and requires a bound predicate and a bound object. The set of all query keys is defined as $\mathbb{K}_q \subset \{0,1\}^d$.

As described in Section 5.3.1, a query key is created from a basic key and a query pattern. The basic key defines the type of the query key and the query pattern specifies its bound and unbound parts. A query pattern consists of two different parts which are a restriction pattern r and a user pattern u. A restriction pattern $r \in \mathbb{P}_q$ is defined a-priori by the data owner and narrows down the possible queries that a user can apply. A user pattern $u \in \mathbb{P}_q$ represents the query parameters specified by the user. The set of all query patterns $\mathbb{P}_q$ is defined as follows:

$$\mathbb{P}_q := (\mathbb{R} \cup \mathbb{B} \cup \{?\}) \times (\mathbb{R} \cup \{?\}) \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L} \cup \{?\}) \tag{5.3}$$

The symbol `?` identifies the unbound parts of a query pattern and corresponds to a variable like `?x`. A query key encodes a query pattern $(s_?, p_?, o_?)$ with $s_? \in (\mathbb{R} \cup \mathbb{B} \cup \{?\})$ as the queried subject, $p_? \in (\mathbb{R} \cup \{?\})$ being the queried predicate, and $o_? \in (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L} \cup \{?\})$ as the queried object. An authorization key $ak \in \mathbb{K}_a$ is a bit string which corresponds to a partially specified query key. It already encodes a basic key bk and a data owner's restriction pattern r. However, an authorization key does not encode a user pattern u. Thus, a complete query key is created from an authorization key by combining it with a user pattern u. The set of all authorization keys is defined as $\mathbb{K}_a \subset \{0,1\}^*$.

5.4.5. Index

An index $I \in \mathbb{I}$ is a mapping from a query key $qk \in \mathbb{K}_q$ to a set of encrypted triples $c \in \mathbb{T}_C$. The index associates a query key with a set of all encrypted triples of a ciphertext graph $G_C \in \mathbb{G}_C$ that match the query pattern which is encoded in the query key. An index is created by the data owner and applied by a user to speed up the querying process. The set $\mathbb{I}$ of all indexes is defined as follows:

$$\mathbb{I} := \mathbb{K}_q \longrightarrow \mathcal{P}(\mathbb{T}_C) \tag{5.4}$$

5.4.6. Query Functions

A query function f applies a single query key $qk \in \mathbb{K}_q$ to an encrypted graph $G_C \in \mathbb{G}_C$ and its corresponding index $I \in \mathbb{I}$ and returns a result set based on all matching triples t of the plaintext graph $G \in \mathbb{G}$. A query function requires a query key qk, the encrypted graph G_C, and its index I as input. Each query function supports one particular type of query keys. Thus, there are eight different query functions which are identified as f_{---}, f_{+--}, f_{-+-}, f_{--+}, f_{++-}, f_{+-+}, f_{-++}, and f_{+++}. Again, the symbols `+` and `-` mark the bound and unbound parts of the supported query keys, respectively. A `+` at the first position requires a subject to be specified in the query key. At the second or the third position, the symbol `+` requires a predicate or an object to be specified, respectively. The result of a query function f also depends on the symbols `+` and `-`. The result can be a set of triples, a set of tuples, a set of resources, a set of blank nodes, a set of literals, or a

boolean value. For example, the query function f_{+--} returns a set of tuples (p, o) with $p \in \mathbb{R} \cup \mathbb{B}$ and $o \in \mathbb{R} \cup \mathbb{B} \cup \mathbb{L}$.

The bound and unbound parts of a query key qk are defined by its encoded query pattern. The query pattern consists of a user pattern $u = (s_u, p_u, o_u)$ and a data owner's restriction pattern $r = (s_r, p_r, o_r)$ as depicted in Figure 5.3 and must comply with the type of the query function. For example, the query function f_{-++} requires a bound predicate and a bound object, i. e., the function requires either $p_r \neq$ `?` or $p_u \neq$ `?` and either $o_r \neq$ `?` or $o_u \neq$ `?`. The data owner may define a restriction pattern $r = (?, \texttt{rdf:type}, ?)$ which specifies `rdf:type` as the predicate of the query pattern. This restriction pattern allows a user to search for instances of a particular ontological class. However, the user must still define a specific class in the user pattern $u = (?, ?, o_u)$ and cannot leave the object position unbound. In this case, the object o_u contains the URI of the class to be queried. Furthermore, any particular value which is already specified in the restriction pattern r cannot be specified in the user pattern u as well. For example, if the data owner defines a restriction pattern $r = (?, \texttt{rdf:type}, ?)$ for the query function f_{-++}, the user cannot define a user pattern $u = (?, \texttt{foaf:mbox}, ?)$ and overwrite the data owner's restriction on the predicate. Thus, a restriction pattern allows the data owner to restrict the possible queries that a user can apply by specifying the corresponding parts in the restriction pattern r. The result of the query function f_{-++} is a list of all subjects s for which the plaintext graph G contains a matching triple (s, p, o). Given a plaintext graph G, its ciphertext graph G_C, an index I, and a query key qk which encodes the query pattern (s, p, o), the eight different query functions f are defined as follows:

$$f_{---} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}(\mathbb{T}), \quad f_{---}(qk_{---}, G_C, I) := G \tag{5.5}$$

$$f_{+--} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}(\mathbb{R} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L})) \tag{5.6}$$
$$f_{+--}(qk_{+--}, G_C, I) := \{(y, z) \mid y \in \mathbb{R}, z \in (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}), \exists (s, y, z) \in G\}$$

$$f_{-+-} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}((\mathbb{R} \cup \mathbb{B}) \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L})) \tag{5.7}$$
$$f_{-+-}(qk_{-+-}, G_C, I) := \{(x, z) \mid x \in (\mathbb{R} \cup \mathbb{B}), z \in (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}), \exists (x, p, z) \in G\}$$

$$f_{--+} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}((\mathbb{R} \cup \mathbb{B}) \times \mathbb{R}) \tag{5.8}$$
$$f_{--+}(qk_{--+}, G_C, I) := \{(x, y) \mid x \in (\mathbb{R} \cup \mathbb{B}), y \in \mathbb{R}, \exists (x, y, o) \in G\}$$

$$f_{++-} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}(\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}) \tag{5.9}$$
$$f_{++-}(qk_{++-}, G_C, I) := \{z \mid z \in (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L}), \exists (s, p, z) \in G\}$$

$$f_{+-+} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}(\mathbb{R}) \tag{5.10}$$
$$f_{+-+}(qk_{+-+}, G_C, I) := \{y \mid y \in \mathbb{R}, \exists (s, y, o) \in G\}$$

$$f_{\texttt{-++}} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \mathcal{P}(\mathbb{R} \cup \mathbb{B}) \tag{5.11}$$
$$f_{\texttt{-++}}(qk_{\texttt{-++}}, G_C, I) := \{x \mid x \in (\mathbb{R} \cup \mathbb{B}), \exists (x, p, o) \in G\}$$

$$f_{\texttt{+++}} : \mathbb{K}_q \times \mathbb{G}_C \times \mathbb{I} \longrightarrow \{TRUE, FALSE\} \tag{5.12}$$
$$f_{\texttt{+++}}(qk_{\texttt{+++}}, G_C, I) := \begin{cases} TRUE & \text{if } (s,p,o) \in G \\ FALSE & \text{otherwise} \end{cases}$$

The query function $f_{\texttt{---}}$ returns the complete plaintext graph G if neither the restriction pattern $r = (s_r, p_r, o_r)$ nor the user pattern $u = (s_u, p_u, o_u)$ specify any particular URIs, blank nodes, or literals. Thus, the function requires $r = u = (\texttt{?}, \texttt{?}, \texttt{?})$. On the other hand, the query function $f_{\texttt{+--}}$ requires either $s_r \neq \texttt{?}$ or $s_u \neq \texttt{?}$. The function searches for all triples $(s, p, o) \in G$ with $s = s_r$ or $s = s_u$ and returns a set of tuples (p, o). Similarly, the functions $f_{\texttt{-+-}}$ and $f_{\texttt{--+}}$ return sets of tuples (s, o) or (s, p), respectively. The function $f_{\texttt{++-}}$ requires both a subject and a predicate to be encoded in the query key. For every matching triple $(s, p, o) \in G$, the object o is returned. The query functions $f_{\texttt{+-+}}$ and $f_{\texttt{-++}}$ are similar and return a set of predicates or subjects, respectively. Finally, the function $f_{\texttt{+++}}$ returns $TRUE$ if a query key encoding the query pattern (s, p, o) matches a triple in G, i. e., if $(s, p, o) \in G$. Otherwise, the function returns $FALSE$.

5.4.7. Triple Keys

A triple key $tk \in \mathbb{K}_t$ represents a SPARQL triple pattern and is used for supporting join operations. It contains a query key $qk \in \mathbb{K}_q$ and associates it with zero, one, two, or three query variables. The specific number of query variables depends on the number of unbound parts which are encoded in the query key. The type of a triple key corresponds to the type of its query key as defined in Section 5.4.4. It is denoted using the symbols `+` and `-` which mark the bound and unbound parts of the query key, respectively. For example, triple keys of type `+--` contain query keys with two unbound parts. Such triple keys require a query variable for the predicate and a query variable for the object. In contrast, triple keys of type `+++` do not require any query variables as their query keys only contain bound triple parts. A triple key is applied to an encrypted graph $G_C \in \mathbb{G}_C$ by processing its query key qk with a corresponding query function f and binding the individual results to their respective query variables. The resulting variable bindings form a solution sequence. Using $\mathbb{V}$ as the set of all possible query variables and $\mathbb{K}_q$ as the set of all query keys as defined in Section 5.4.4, the set $\mathbb{K}_t$ of all triple keys is defined as follows:

$$\mathbb{K}_t := \mathbb{K}_q \times (\emptyset \cup \mathbb{V} \cup (\mathbb{V} \times \mathbb{V}) \cup (\mathbb{V} \times \mathbb{V} \times \mathbb{V})) \tag{5.13}$$

The number of query variables in a triple key must be equal to the number of unbound parts that are encoded in its query key qk.

5.4.8. Query Algebras, Query Forms, and Queries

A query $q \in \mathbb{Q}$ consists of a query form and a query algebra $QA \in \mathbb{A}$. The structure of the query form depends on the type of the query whereas the query algebra has the

same structure for all query types. The query algebra QA contains an arbitrary number of triple keys $tk \in \mathbb{K}_t$ and corresponds to the `WHERE` clause of a SPARQL query. The set of all query algebras is defined as $\mathbb{A} := \mathcal{P}(\mathbb{K}_t)$. A query algebra is applied to an encrypted graph G_C by applying all its triple keys to the graph and combining their individual solution sequences with a join operation into a single solution sequence. Join operations are conducted by evaluating the variable bindings of the solution sequences of all triple keys. The query form of a query defines how the solution sequence of the query algebra shall be further refined. The query form of a `SELECT` query defines a list of query variables whose variable bindings shall be returned as the final query result. The set of all `SELECT` queries is defined as follows:

$$\mathbb{Q}_{\texttt{SELECT}} := \{(v_1, \ldots, v_n) \mid v_i \in \mathbb{V}, i \in \{1, \ldots, n\}, n \in \mathbb{N}\} \times \mathbb{A} \tag{5.14}$$

The query variables $v_i \in \mathbb{V}$ of a query form must also be defined within the triple keys of the query algebra $QA \in \mathbb{A}$. The query form of a `CONSTRUCT` query defines a graph template which is used for creating a new RDF graph from the solution sequence of the query algebra. This RDF graph is then returned as the final query result. A graph template contains an arbitrary number of template patterns $tp \in \mathbb{P}_t$. A template pattern is similar to a query pattern and also contains both bound and unbound triple parts. In a template pattern, bound parts are identified by resources $\mathbb{R}$, blank nodes $\mathbb{B}$, or literals $\mathbb{L}$ whereas unbound parts are represented as query variables $\mathbb{V}$. The set of all template patterns is defined as follows:

$$\mathbb{P}_t := (\mathbb{R} \cup \mathbb{B} \cup \mathbb{V}) \times (\mathbb{R} \cup \mathbb{V}) \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L} \cup \mathbb{V}) \tag{5.15}$$

A graph template is instantiated by replacing all query variables in the template patterns with corresponding values. The values are taken from the solution mappings of the query algebra's solution sequence. The triples that result from all instantiations are combined into a single RDF graph which is returned as the final query result. Using the set $\mathbb{A}$ of all query algebras and the set $\mathbb{P}_t$ of all template patterns, the set of all `CONSTRUCT` queries is defined as follows:

$$\mathbb{Q}_{\texttt{CONSTRUCT}} := \mathcal{P}(\mathbb{P}_t) \times \mathbb{A} \tag{5.16}$$

All query variables of the graph template must also be defined in the triple keys tk of the query algebra $QA \in \mathbb{A}$. Finally, `ASK` queries return a boolean value and only require a query algebra with an arbitrary number of triple keys. The set of all `ASK` queries is defined as follows:

$$\mathbb{Q}_{\texttt{ASK}} := \mathbb{A} \tag{5.17}$$

Based on these three different sets of queries, the set of all SPARQL queries is defined as follows:

$$\mathbb{Q} := \mathbb{Q}_{\texttt{SELECT}} \cup \mathbb{Q}_{\texttt{CONSTRUCT}} \cup \mathbb{Q}_{\texttt{ASK}} \tag{5.18}$$

5.5. Design of T-Store

The overall process for searching in encrypted RDF graphs is divided into four phases which are depicted in Figure 5.4. The encryption phase encrypts a plaintext graph $G \in \mathbb{G}$ which results in a ciphertext graph $G_C \in \mathbb{G}_C$. The indexing phase creates an index $I \in \mathbb{I}$ on top of the ciphertext graph in order to speed up the querying process. In the authorization phase, the data owner authorizes users to apply queries to the encrypted graph. The last phase is the actual querying phase which is carried out by an authorized user. The following subsections describe each phase in more detail.

Encryption phase	Step 1	Choosing basic keys
	Step 2	Creating encryption keys
	Step 3	Encrypting the triples
Indexing phase	Step 4	Choosing ciphertext identifiers
	Step 5	Grouping ciphertext identifiers
	Step 6	Creating ciphertext arrays
	Step 7	Encrypting ciphertext arrays
	Step 8	Computing index keys
	Step 9	Creating the index tree
Authorization phase	Step 10	Publishing ciphertext graph and index
	Step 11	Creating authorization keys
	Step 12	Distributing authorization keys to users
Query phase	Step 13	Creating user patterns
	Step 14	Creating query keys
	Step 15	Identifying the query results
	Step 16	Decrypting the triples

Figure 5.4.: Different phases and steps for searching in encrypted graphs.

5.5.1. Encryption Phase

The encryption phase consists of three steps in which the data owner encrypts all triples of the plaintext graph in such a way that the resulting ciphertext graph can be used for querying. Users are not involved in this phase.

Step 1: Choosing Basic Keys

The data owner chooses eight different basic keys $bk \in \mathbb{K}_b$. Each basic key bk defines a specific variant of query patterns. Basic keys are used for creating encryption keys ek and authorization keys ak. A particular set of basic keys is only used for one plaintext graph. This ensures that users who are able to apply queries to one graph cannot apply queries to other graphs as well by using the same authorization keys.

Step 2: Creating Encryption Keys

The data owner creates a symmetric encryption key ek for each plaintext triple $t \in G$ and for each of the eight basic keys bk created in the first step. An encryption key $ek \in \mathbb{K}_e$ is a bit string of length $d \in \mathbb{N}$ with $\mathbb{K}_e \subset \{0,1\}^d$ being the set of all encryption keys. It encodes a basic key $bk \in \mathbb{K}_b$ and the bound parts of a query pattern which correspond to the query parameters. The number and position of the bound parts are defined by the query pattern variant encoded in the basic key. For example, the basic key $bk_{\texttt{+--}}$ requires a bound subject. Thus, it is encoded in an encryption key ek together with a triple's subject. An encryption key ek is similar to a query key qk which is created and applied by an authorized user in the query phase.

An encryption key ek is created with a basic hash function λ and a combining function ϱ. A basic hash function transforms a bit string of arbitrary length to a hash value of fixed length [255]. Examples of basic hash functions are MD5 [250] and SHA-2 [218]. A formal definition of the basic hash function is provided in Equation 4.12 in Section 4.3.5. The basic hash function is used for reducing the possibility of creating identical encryption keys for different input data. The combining function ϱ combines a bit string $b \in \{0,1\}^*$ and the bound parts of a triple t into a single bit string. Each of the bound triple parts is represented as a bit string of arbitrary length. The combining function ϱ is based on the basic hash function λ. Given a product $N \in \mathbb{N}$ of two large prime numbers and $n \in \{0,1,2,3\}$ as the number of all bound triple parts, the function is defined as follows:

$$\varrho : \{0,1\}^* \times \mathcal{P}(\{0,1\}^*) \rightarrow \{0,1\}^*, \tag{5.19}$$

$$\varrho(b, \{b_1, \ldots, b_n\}) := \mathsf{bit}\left(\mathsf{int}(b)^{\prod_{i=1}^{n} \mathsf{int}(\lambda(b_i))} \bmod N\right)$$

Table 5.1.: Step 2. Creating encryption keys ek for a single triple $t = (s,p,o)$ using eight different basic keys bk. The symbol $||$ represents the operator for concatenating bit strings.

Input		**Output**
Basic key bk	**Plaintext triple** t	**Resulting encryption key** ek
$bk_{\texttt{---}}$	(s,p,o)	$ek_{\texttt{---}} = \lambda(\varrho(bk_{\texttt{---}}, \{\}))$
$bk_{\texttt{+--}}$	(s,p,o)	$ek_{\texttt{s--}} = \lambda(\varrho(bk_{\texttt{+--}}, \{w_{\texttt{s}}\|\|s\}))$
$bk_{\texttt{-+-}}$	(s,p,o)	$ek_{\texttt{-p-}} = \lambda(\varrho(bk_{\texttt{-+-}}, \{w_{\texttt{p}}\|\|p\}))$
$bk_{\texttt{--+}}$	(s,p,o)	$ek_{\texttt{--o}} = \lambda(\varrho(bk_{\texttt{--+}}, \{w_{\texttt{o}}\|\|o\}))$
$bk_{\texttt{++-}}$	(s,p,o)	$ek_{\texttt{sp-}} = \lambda(\varrho(bk_{\texttt{++-}}, \{w_{\texttt{s}}\|\|s, w_{\texttt{p}}\|\|p\}))$
$bk_{\texttt{+-+}}$	(s,p,o)	$ek_{\texttt{s-o}} = \lambda(\varrho(bk_{\texttt{+-+}}, \{w_{\texttt{s}}\|\|s, w_{\texttt{o}}\|\|o\}))$
$bk_{\texttt{-++}}$	(s,p,o)	$ek_{\texttt{-po}} = \lambda(\varrho(bk_{\texttt{-++}}, \{w_{\texttt{p}}\|\|p, w_{\texttt{o}}\|\|o\}))$
$bk_{\texttt{+++}}$	(s,p,o)	$ek_{\texttt{spo}} = \lambda(\varrho(bk_{\texttt{+++}}, \{w_{\texttt{s}}\|\|s, w_{\texttt{p}}\|\|p, w_{\texttt{o}}\|\|o\}))$

The function int maps a bit string to an integer and bit maps an integer to a bit string. When creating an encryption key ek with the combining function ϱ, the bit string b corresponds to a basic key $bk \in \mathbb{K}_b$. However, the function is also used for creating authorization keys ak and query keys qk. Before applying the combining function ϱ to a particular triple, each of the bound parts of the triple is prefixed with a bit string of fixed length. The prefix w_{s} is used for the triple's subject, w_{p} identifies the triple's predicate, and w_{o} marks the triple's object. In Equation 5.19, the bit strings b_i with $i \in \{1, \ldots, n\}$ correspond to the prefixed bit string representations of the bound triple parts. Table 5.1 depicts the details of creating different encryption keys ek for the same triple $t = (s, p, o)$ and different basic keys bk.

Step 3: Encrypting the Triples

The data owner uses the encryption key ek for encrypting the unbound parts of a triple which are not already encoded in the encryption key. For example, an encryption key ek based on a basic key $bk_{\texttt{+--}}$ encodes the subject of a triple t. This encryption key is used for encrypting the predicate and object of the same triple t. The ciphertext resulting from this operation is denoted as $c_{\texttt{+--}}$. Each ciphertext depends on all parts of the plaintext triple which are either encoded in the encryption key ek or encrypted with this key. The encryption is conducted using a symmetric encryption function ξ. The function requires a bit string representation of the triple and the encryption key as input and returns an encrypted bit string. Examples of symmetric encryption functions are AES [213] and Twofish [273]. The encryption function ξ is defined as follows:

$$\xi : \mathbb{K}_e \times \{0,1\}^* \rightarrow \{0,1\}^* \tag{5.20}$$

Table 5.2.: Step 3. Encrypting a single triple $t = (s, p, o)$ with the encryption keys ek created in step 2. The symbol $||$ corresponds to the concatenation operator and ε corresponds to the empty bit string. The result is an encrypted triple $c = (c_{\texttt{---}}, \ldots, c_{\texttt{+++}})$.

Input		Output
Encryption key ek	**Plaintext triple** t	**Resulting ciphertext**
$ek_{\texttt{---}}$	(s, p, o)	$c_{\texttt{---}} = \xi(ek_{\texttt{---}}, s\|\|p\|\|o)$
$ek_{\texttt{s--}}$	(s, p, o)	$c_{\texttt{+--}} = \xi(ek_{\texttt{s--}}, p\|\|o)$
$ek_{\texttt{-p-}}$	(s, p, o)	$c_{\texttt{-+-}} = \xi(ek_{\texttt{-p-}}, s\|\|o)$
$ek_{\texttt{--o}}$	(s, p, o)	$c_{\texttt{--+}} = \xi(ek_{\texttt{--o}}, s\|\|p)$
$ek_{\texttt{sp-}}$	(s, p, o)	$c_{\texttt{++-}} = \xi(ek_{\texttt{sp-}}, o)$
$ek_{\texttt{s-o}}$	(s, p, o)	$c_{\texttt{+-+}} = \xi(ek_{\texttt{s-o}}, p)$
$ek_{\texttt{-po}}$	(s, p, o)	$c_{\texttt{-++}} = \xi(ek_{\texttt{-po}}, s)$
$ek_{\texttt{spo}}$	(s, p, o)	$c_{\texttt{+++}} = \xi(ek_{\texttt{spo}}, \varepsilon)$

Table 5.2 depicts the details of encrypting a single triple t. Each triple $t \in G$ is encrypted for each of the eight basic keys bk. This results in an encrypted triple $c = (c_{---}, c_{+--}, c_{-+-}, c_{--+}, c_{++-}, c_{+-+}, c_{-++}, c_{+++})$ as defined Section 5.4.2. The ciphertext c_{+++} is created by encrypting the empty bit string $\varepsilon \in \{0,1\}^*$. The result of this step is an encrypted graph $G_C \in \mathbb{G}_C$.

5.5.2. Indexing Phase

The indexing phase consists of six steps in which the data owner creates an index $I \in \mathbb{I}$ for all encrypted triples c. This index is used for speeding up the query processing conducted by the user. It is based on a self-balancing binary search tree (BST) [181] and stores sets of all those ciphertexts which match against the same query pattern. As a query pattern is encoded in a query key $qk \in \mathbb{K}_q$, the index allows a user to efficiently retrieve all ciphertexts which can be decrypted with a given query key qk. The creation and usage of the index is explained along the example graph shown in Listing 5.2, which consists of eight simplified triples and is encoded using N-Triples [25].

```
<a> <b> <c> .      <a> <b> <d> .      <a> <b> <e> .      <a> <f> <c> .
<a> <f> <e> .      <c> <b> <a> .      <c> <b> <e> .      <e> <f> <d> .
```

Listing 5.2: Example graph which consists of eight triples. For simplicity reasons, the triples only contain URIs which are represented as single letters.

Step 4: Choosing Ciphertext Identifiers

The data owner associates each of the eight ciphertexts of an encrypted triple c with a unique and random identifier id_c. Encrypting a plaintext graph G with $n \in \mathbb{N}$ triples results in an encrypted graph G_C with $8n$ different ciphertexts. Thus, the data owner creates $8n$ different ciphertext identifiers $id_c \in \{1, \ldots, 8n\}$. Table 5.3 shows the result of this step for the example plaintext graph depicted in Listing 5.2. Since the example graph consists of eight triples, the data owner creates 64 different ciphertext identifiers.

Step 5: Grouping Ciphertext Identifiers

The data owner divides the ciphertext identifiers id_c created in the previous step into disjoint sets S in such a way that the corresponding ciphertexts of each set can be decrypted with the same encryption key ek. Thus, each set S is associated with a particular encryption key ek. The results of this step for the example graph is depicted in Table 5.4. In the example, 35 different sets S are created.

Step 6: Creating Ciphertext Arrays

The data owner maps all sets S of ciphertext identifiers to one-dimensional arrays A of equal size $z \in \mathbb{N}$. The array size z is a parameter of the index and chosen by the data owner. Its particular value may influence the efficiency and size of the index. Suitable

Table 5.3.: Step 4. Associating each ciphertext in an encrypted graph G_C with a unique and random ciphertext identifier id_c. The identifiers are created for all eight ciphertexts of an encrypted triple c.

		Encrypted triple c							
		c_{---}	c_{+--}	c_{-+-}	c_{--+}	c_{++-}	c_{+-+}	c_{-++}	c_{+++}
Plaintext triple t	<a> <b> <c>	34	52	48	3	40	12	46	50
	<a> <b> <d>	10	44	25	53	41	35	14	11
	<a> <b> <e>	30	18	59	4	21	63	61	20
	<a> <f> <c>	39	7	45	5	38	47	28	49
	<a> <f> <e>	55	56	16	6	43	62	60	31
	<c> <b> <a>	17	27	29	51	13	58	57	24
	<c> <b> <e>	1	32	23	33	15	8	54	19
	<e> <f> <d>	22	26	36	2	9	42	64	37

Table 5.4.: Step 5. Sets S of ciphertext identifiers id_c whose ciphertexts are created with the same encryption key ek. For readability reasons, the sets are grouped according to the basic keys bk which are used for creating the corresponding encryption keys ek.

Basic key bk	**Encryption key** ek	**Set of ciphertext identifiers** id_c
bk_{---}	ek_{---}	$S_1 = \{1, 10, 17, 22, 30, 34, 39, 55\}$
bk_{+--}	$ek_{\texttt{a}--}$	$S_2 = \{7, 18, 44, 52, 56\}$
bk_{+--}	$ek_{\texttt{c}--}$	$S_3 = \{27, 32\}$
bk_{+--}	$ek_{\texttt{e}--}$	$S_4 = \{26\}$
bk_{-+-}	$ek_{-\texttt{b}-}$	$S_5 = \{23, 25, 29, 48, 59\}$
bk_{-+-}	$ek_{-\texttt{f}-}$	$S_6 = \{16, 36, 45\}$
...	...	...
bk_{+++}	$ek_{\texttt{afe}}$	$S_{32} = \{31\}$
bk_{+++}	$ek_{\texttt{cba}}$	$S_{33} = \{24\}$
bk_{+++}	$ek_{\texttt{cbe}}$	$S_{34} = \{19\}$
bk_{+++}	$ek_{\texttt{efd}}$	$S_{35} = \{37\}$

values of the array size z are discussed in Section 5.6.4. Let $|S|$ be the number of ciphertext identifiers id_c in a set S. If it is $|S| > z$, the data owner creates multiple arrays for the set S. The total number of arrays which are created for a set S is $\lceil \frac{|S|}{z} \rceil$. Arrays which contain less than z ciphertext identifiers id_c are padded with the value 0 so that all arrays are of identical size z. The result of this step for the example graph is shown in Table 5.5. In the example, 38 arrays are created with an array size of $z = 4$.

The sets S_1, S_2, and S_5 are mapped to two arrays and all other sets are mapped to a single array.

Table 5.5.: Step 6. Mapping the sets S of ciphertext identifiers to arrays A of equal size z. In the example it is $z = 4$. Sets containing more than four identifiers are split into multiple arrays. Arrays of smaller sets are padded with 0s.

Set S	**Corresponding arrays** A
S_1	$A_{1.1} = \langle 1, 10, 17, 20 \rangle$, $A_{1.2} = \langle 30, 34, 39, 55 \rangle$
S_2	$A_{2.1} = \langle 7, 18, 44, 52 \rangle$, $A_{2.2} = \langle 56, 0, 0, 0 \rangle$
S_3	$A_3 = \langle 27, 32, 0, 0 \rangle$
...	...
S_{35}	$A_{35} = \langle 37, 0, 0, 0 \rangle$

Step 7: Encrypting Ciphertext Arrays

The data owner encrypts each array A with the encryption key ek of its corresponding set S by using the encryption function ξ as defined in Equation 5.20. Different arrays of the same set are encrypted with the same encryption key. The resulting encrypted arrays are stored in the index created in step 9. In order to speed up the process of retrieving all arrays of the same set, each array is encrypted together with an additional bit marking whether or not there are more arrays of the same set. For example, the set S_1 is mapped to the two arrays $A_{1.1}$ and $A_{1.2}$. The first array $A_{1.1}$ is encrypted together with the bit 1. This bit indicates the existence of an array $A_{1.2}$. In contrast, the array $A_{1.2}$ is encrypted together with the bit 0 since an array $A_{1.3}$ does not exist. The result of this step for the example graph is depicted in Table 5.6.

Step 8: Computing Index Keys

The data owner associates each encrypted array with an index key ik. An index key $ik \in \mathbb{K}_i$ is a bit string of length d with $\mathbb{K}_i \subset \{0,1\}^d$ being the set of all index keys. An index key is used for locating an encrypted array in the index I. It is computed with a basic hash function λ by using the encryption key ek of the array's set S and an array identifier $id_A \in \{1, \ldots, \lceil \frac{|S|}{z} \rceil\}$ as input. Array identifiers id_A distinguish between different arrays of the same set. For example, the index keys $ik_{1.1}$ and $ik_{1.2}$ refer to the arrays $A_{1.1}$ and $A_{1.2}$, respectively. The index key $ik_{1.1}$ is computed from the encryption key ek_{---} and from the array identifier $id_{A_{1.1}} = 1$ as $\lambda(ek_{---}||1)$ using $||$ as the concatenation operator. The index key $ik_{1.2}$ is computed from ek_{---} and $id_{A_{1.2}} = 2$ as $\lambda(ek_{---}||2)$. The result of this step for the example graph is shown in Table 5.7.

Table 5.6.: Step 7. Encrypting ciphertext arrays A with encryption keys ek. Each array is encrypted with an additional bit denoting whether or not there are more arrays of the same set. The value 1 marks the existence of such an array and 0 denotes its absence. Arrays of the same set are encrypted with the same encryption key. The symbol || corresponds to the concatenation operator.

Input		**Output**
Array A	**Encryption key** ek	**Resulting encrypted array**
$A_{1.1}$	$ek_{\texttt{---}}$	$\xi(ek_{\texttt{---}}, 1 \| \| A_{1.1})$
$A_{1.2}$	$ek_{\texttt{---}}$	$\xi(ek_{\texttt{---}}, 0 \| \| A_{1.2})$
$A_{2.1}$	$ek_{\texttt{a--}}$	$\xi(ek_{\texttt{a--}}, 1 \| \| A_{2.1})$
$A_{2.2}$	$ek_{\texttt{a--}}$	$\xi(ek_{\texttt{a--}}, 0 \| \| A_{2.2})$
A_3	$ek_{\texttt{c--}}$	$\xi(ek_{\texttt{c--}}, 0 \| \| A_3)$
...	...	...
A_{35}	$ek_{\texttt{efd}}$	$\xi(ek_{\texttt{efd}}, 0 \| \| A_{35})$

Table 5.7.: Step 8. Creating index keys ik for the encrypted arrays A. An index key is created from the encryption key ek which is associated with the array's set S and an array identifier id_A. The identifiers distinguish between different arrays of the same set S. The symbol || marks the concatenation operator.

	Input		**Output**
Array A	**Encryption key** ek	**Array identifier** id_A	**Resulting index key** ik
$A_{1.1}$	$ek_{\texttt{---}}$	$id_{A_{1.1}} = 1$	$ik_{1.1} = \lambda(ek_{\texttt{---}} \| \| 1)$
$A_{1.2}$	$ek_{\texttt{---}}$	$id_{A_{1.2}} = 2$	$ik_{1.2} = \lambda(ek_{\texttt{---}} \| \| 2)$
$A_{2.1}$	$ek_{\texttt{a--}}$	$id_{A_{2.1}} = 1$	$ik_{2.1} = \lambda(ek_{\texttt{a--}} \| \| 1)$
$A_{2.2}$	$ek_{\texttt{a--}}$	$id_{A_{2.1}} = 2$	$ik_{2.2} = \lambda(ek_{\texttt{a--}} \| \| 2)$
A_3	$ek_{\texttt{c--}}$	$id_{A_3} = 1$	$ik_3 = \lambda(ek_{\texttt{c--}} \| \| 1)$
...	...	...	...
A_{35}	$ek_{\texttt{efd}}$	$id_{A_{35}} = 1$	$ik_{35} = \lambda(ek_{\texttt{efd}} \| \| 1)$

Step 9: Creating the Index Tree

The data owner creates an index which uses the index keys ik from the previous step for indexing the encrypted ciphertext arrays created in step 7. Thus, the index provides a mapping from index keys to encrypted arrays. The index is a self-balancing binary search tree (BST) [181]. All nodes in such a tree consist of a key and a value and are ordered according to their key. In T-Store, the keys of the nodes correspond to index keys ik and the values are encrypted ciphertext arrays. The root node and all internal

nodes of a BST have one or two child nodes which form separate subtrees. All keys in the left subtree of a particular node are smaller than the key of this node and all keys in the right subtree are larger. As the index keys ik of T-Store are bit strings, the encrypted ciphertext arrays are ordered according to the natural order of these bit strings. For example, if the bit string representation of the index key $ik_{2.1}$ is larger than the bit string representation of $ik_{1.2}$, the ciphertext array $A_{1.2}$ is positioned to the left of the array $A_{2.1}$. In addition to these basic characteristics, all paths in a BST which start at the root node and end at a leaf node have almost the same length. This results in a runtime complexity of searching a node in the tree of $\mathcal{O}(\log m)$ with m being the number of nodes in the tree. Example implementations of self-balancing binary search trees are red-black trees [23] and AVL trees [9]. The result of this step for the example graph shown in Listing 5.2 is depicted in Figure 5.5.

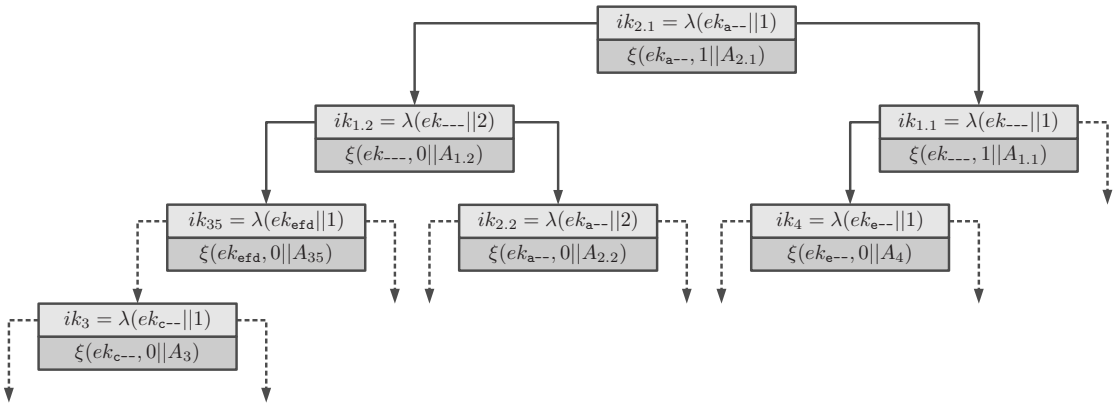


Figure 5.5.: Step 9. A fragment of the index for storing the encrypted ciphertext arrays created in step 7. The index is a self-balancing binary search tree. Encrypted ciphertext arrays are indexed using the index keys ik created in step 8. Index keys ik are basically hashed encryption keys ek and are ordered according to their bit string representation.

5.5.3. Authorization Phase

The authorization phase consists of three steps and involves both the data owner and the authorized users. In this phase, the data owner authorizes users to apply queries to an encrypted graph.

Step 10: Publishing ciphertext graph and index

The data owner publishes the ciphertext graph G_C created in step 3 and the index created in step 9 on the web. As the data stored in the graph and in the index is encrypted, a user cannot access any data without being authorized by the data owner first. Alternatively, the data owner can send the ciphertext graph and the index specifically to authorized users. However, this requires a direct interaction between the data owner and a user.

Step 11: Creating Authorization Keys

The data owner creates an authorization key ak for each query pattern of an allowed query. As defined in Section 5.4.4, an authorization key $ak \in \mathbb{K}_a$ is created from a basic key $bk \in \mathbb{K}_b$ and a data owner's restriction pattern $r = (s_r, p_r, o_r) \in \mathbb{P}_q$. The bound parts of the restriction pattern r are first prefixed with identifiers that define their position within the query pattern. Again, the prefix $w_{\mathtt{s}}$ marks the triple's subject, $w_{\mathtt{p}}$ denotes the triple's predicate, and $w_{\mathtt{o}}$ identifies the triple's object. The prefixed strings are then combined with the basic key bk to create an authorization key ak by using the combining function ϱ defined in Equation 5.19. The detailed process for creating authorization keys is depicted in Table 5.8. A basic key also determines the number and type of different authorization keys which can be created for a particular query pattern variant. For example, the basic key $bk_{\mathtt{+--}}$ can be used for creating the two authorization keys $ak_{\mathtt{r??}}$ and $ak_{\mathtt{u??}}$. The index `r` states that the corresponding part of the authorization key is predefined by the data owner's restriction pattern r and `u` indicates that the user must specify the part within the user pattern u. The index `?` marks the unbound parts in the authorization key which correspond to the query result.

Table 5.8.: Step 11: Creating authorization keys $ak \in \mathbb{K}_a$ by applying the combining function ϱ to basic keys $bk \in \mathbb{K}_b$ and restriction patterns $r \in \mathbb{P}_q$.

Input		**Output**
Basic key bk	**Restriction pattern** r	**Resulting authorization key** ak
$bk_{\mathtt{---}}$	$(\mathtt{?}, \mathtt{?}, \mathtt{?})$	$ak_{\mathtt{???}} = \varrho(bk_{\mathtt{---}}, \{\})$
$bk_{\mathtt{+--}}$	$(s_r, \mathtt{?}, \mathtt{?}), s_r \neq \mathtt{?}$	$ak_{\mathtt{r??}} = \varrho(bk_{\mathtt{+--}}, \{w_{\mathtt{s}} \| s_r\})$
$bk_{\mathtt{+--}}$	$(\mathtt{?}, \mathtt{?}, \mathtt{?})$	$ak_{\mathtt{u??}} = \varrho(bk_{\mathtt{+--}}, \{\})$
$bk_{\mathtt{++-}}$	$(s_r, p_r, \mathtt{?}), s_r, p_r \neq \mathtt{?}$	$ak_{\mathtt{rr?}} = \varrho(bk_{\mathtt{++-}}, \{w_{\mathtt{s}} \| s_r, w_{\mathtt{p}} \| p_r\})$
$bk_{\mathtt{++-}}$	$(s_r, \mathtt{?}, \mathtt{?}), s_r \neq \mathtt{?}$	$ak_{\mathtt{ru?}} = \varrho(bk_{\mathtt{++-}}, \{w_{\mathtt{s}} \| s_r\})$
$bk_{\mathtt{++-}}$	$(\mathtt{?}, p_r, \mathtt{?}), p_r \neq \mathtt{?}$	$ak_{\mathtt{ur?}} = \varrho(bk_{\mathtt{++-}}, \{w_{\mathtt{p}} \| p_r\})$
$bk_{\mathtt{++-}}$	$(\mathtt{?}, \mathtt{?}, \mathtt{?})$	$ak_{\mathtt{uu?}} = \varrho(bk_{\mathtt{++-}}, \{\})$
...	...	...

Step 12: Distributing Authorization Keys to Users

The data owner sends a set of authorization keys ak to a user. In doing so, the data owner authorizes the user to apply all queries which can be applied to an encrypted graph by using the received authorization keys ak. The authorization keys ak are transmitted via a secure communication channel in order to ensure that only authorized users can apply the queries. This step is the only step that requires a direct interaction between the data owner and the user. All previous steps are solely conducted by the data owner and all following steps are conducted by the user alone.

5.5.4. Query Phase

The query phase consists of four steps which are conducted by an authorized user. In this phase, an authorized user queries an encrypted graph G_C by using the received authorization keys ak. This section describes how a user can process a query consisting of a single triple key tk. The application of more complex queries consisting of multiple triple keys is discussed in Section 5.5.5. The user applies each triple key to the ciphertext graph G_C by processing its query key qk with a corresponding query function f.

Step 13: Choosing User Patterns

The user choses a user pattern $u \in \mathbb{P}_q$ which is compatible with the received authorization key $ak \in \mathbb{K}_a$. As described in Section 5.4.4, an authorization key ak already encodes a data owner's restriction pattern $r \in \mathbb{P}_q$. The more parts the data owner specifies in the restriction pattern r, the fewer choices does the user have to formulate a user pattern u. For example, the query function $f_{\text{-++}}$ requires a predicate and an object to be either specified in r or in u. If the data owner defines $r = (?, \texttt{rdf:type}, \texttt{foaf:Person})$, a user can only search for resources of `rdf:type foaf:Person` from the FOAF vocabulary [50]. In this case, the user can only define a user pattern $u = (?, ?, ?)$ which does not add any further query parameters to the restriction pattern r. However, if the data owner defines $r = (?, \texttt{rdf:type}, ?)$, the user can specify different values for the object of the user pattern. For example, the user may define $u = (?, ?, \texttt{foaf:Organization})$ to search for all resources of `rdf:type foaf:Organization`.

Step 14: Creating Query Keys

The user creates a query key $qk \in \mathbb{K}_q$ by combining an authorization key $ak \in \mathbb{K}_a$ with a compatible user pattern $u \in \mathbb{P}_q$ created in the previous step. A query key qk is similar to an encryption key $ek \in \mathbb{K}_e$ and thus created similarly as well. A query key encodes a query pattern and is used as input for a corresponding query function f. Table 5.9 depicts the details of creating a query key qk from a received authorization key ak and a user pattern u. A query key is created by using the combining function ϱ and the basic hash function λ. The bound parts of the user pattern are first prefixed with the identifiers $w_{\mathtt{s}}$, $w_{\mathtt{p}}$, and $w_{\mathtt{o}}$. These identifiers are used to determine the position of the bound parts of the user pattern. If a query key qk was created correctly, it encodes all bound parts of a query pattern and is identical to the encryption key ek which was used for encrypting the unbound parts. In this case, the query key can be directly used for decrypting the unbound triple parts.

Step 15: Identifying the Query Results

The user applies the index I in order to identify all ciphertexts which match against a particular query pattern. The index is accessed by index keys $ik \in \mathbb{K}_i$ which are computed from the query keys created in the previous step. For each query key qk, the user computes $ik_1 = \lambda(qk||1)$ and locates ik_1 in the index. If the index does not

Table 5.9.: Step 14: Creating a query key qk from an authorization key ak and a user pattern $u = (s_u, p_u, o_u)$. The authorization key encodes a basic key bk and the data owner's restriction pattern $r = (s_r, p_r, o_r)$.

Input		**Output**
Authorization key ak	**User pattern** u	**Resulting query key** qk
$ak_{\texttt{???}}$	$(\texttt{?}, \texttt{?}, \texttt{?})$	$qk_{\texttt{???}} = \lambda(\varrho(ak_{\texttt{???}}, \{\}))$
$ak_{\texttt{r??}}$	$(\texttt{?}, \texttt{?}, \texttt{?})$	$qk_{\texttt{r??}} = \lambda(\varrho(ak_{\texttt{r??}}, \{\})$
$ak_{\texttt{u??}}$	$(s_\texttt{u}, \texttt{?}, \texttt{?}), s_\texttt{u} \neq \texttt{?}$	$qk_{\texttt{u??}} = \lambda(\varrho(ak_{\texttt{u??}}, \{w_\texttt{s} \| \| s_\texttt{u}\}))$
$ak_{\texttt{rr?}}$	$(\texttt{?}, \texttt{?}, \texttt{?})$	$qk_{\texttt{rr?}} = \lambda(\varrho(ak_{\texttt{rr?}}, \{\})$
$ak_{\texttt{ru?}}$	$(\texttt{?}, p_\texttt{u}, \texttt{?}), p_\texttt{u} \neq \texttt{?}$	$qk_{\texttt{ru?}} = \lambda(\varrho(ak_{\texttt{ru?}}, \{w_\texttt{p} \| \| p_\texttt{u}\}))$
$ak_{\texttt{ur?}}$	$(s_\texttt{u}, \texttt{?}, \texttt{?}), s_\texttt{u} \neq \texttt{?}$	$qk_{\texttt{ur?}} = \lambda(\varrho(ak_{\texttt{ur?}}, \{w_\texttt{s} \| \| s_\texttt{u}\}))$
$ak_{\texttt{uu?}}$	$(s_\texttt{u}, p_\texttt{u}, \texttt{?}), s_\texttt{u}, p_\texttt{u} \neq \texttt{?}$	$qk_{\texttt{uu?}} = \lambda(\varrho(ak_{\texttt{uu?}}, \{w_\texttt{s} \| \| s_\texttt{u}, w_\texttt{p} \| \| p_\texttt{u}\}))$
...	...	...

contain an entry for the index key ik_1, the query pattern encoded in the query key qk has an empty result. In this case, the user can immediately stop any further processing. Otherwise, the user can retrieve the encrypted array associated with the index key ik_1 and decrypt it using the query key qk and a decryption function μ. The decryption function μ is inverse to the encryption function ξ defined in Equation 5.20. It requires a query key qk and a bit string as input and returns a decrypted bit string as output. The decryption function is defined as follows:

$$\mu : \mathbb{K}_q \times \{0,1\}^* \rightarrow \{0,1\}^* \tag{5.21}$$

Applying the decryption function to an encrypted array results in a plaintext array A and a preceding bit value. The array A contains up to z ciphertext identifiers id_c which refer to encrypted triples satisfying the query pattern. If the plaintext array A is preceded by the bit value 0, the user has already retrieved all ciphertext triples for the query pattern. In contrast, a leading bit value of 1 states that there are more arrays in the index which also contain relevant ciphertext identifiers. In this case, the user computes a new index key $ik_2 = \lambda(qk||2)$ and looks it up in the tree. This process is repeated until the preceding bit value of a plaintext array is 0. Finally, the user has obtained all ciphertext identifiers which refer to exactly those ciphertexts that match against the query pattern encoded in the query key qk.

Step 16: Decrypting the Triples

The user applies a query key qk to the decryption function μ in order to decrypt each ciphertext identified in the previous step. If the query key qk was created correctly in step 14, it is $ek = qk$. In this case, the decryption process is inverse to the encryption operation, i. e., $b = \mu(qk, \xi(ek, b))$ with $b \in \{0,1\}^*$ being the bit string representation of

the unbound parts of a plaintext triple t. Steps 15 and 16 are combined into a query function f. Thus, the user processes these two steps by applying a query function to a query key qk, an index I, and a ciphertext graph G_C.

5.5.5. Applying Queries with Multiple Triple Keys

A query $q \in \mathbb{Q}$ is applied to a ciphertext graph $G_C \in \mathbb{G}_C$ and its index $I \in \mathbb{I}$ by applying the query algebra and further processing the resulting solution sequence with the query form. T-Store supports SPARQL queries of type `SELECT`, `CONSTRUCT`, and `ASK`. Processing the query algebra is the same for all three types of queries whereas processing the query form depends on the query type. This section first defines a basic algorithm for applying a query algebra to a ciphertext graph. Subsequently, three different algorithms are presented for each of the supported query types. These algorithms process the query algebra using the first algorithm and refine the resulting solution sequence with a specific query form.

Algorithm 5.1 applies a query algebra $QA \in \mathbb{A}$ of a query q to a ciphertext graph G_C and its index I. A query algebra is applied to a ciphertext graph by matching all its triple keys $tk \in QA$ with the graph's triples. The query algebra can successfully be applied to the graph if all triple keys match at least one triple in the graph. In this case, the solution sequence of the query algebra is created from all successful matches. However, if the query algebra does not match at all, an empty solution sequence is returned. This is the case if at least one triple key of the query algebra does not match any of the graph's triples. Algorithm 5.1 first applies all triple keys of type `+++` to the ciphertext graph (line 2). These triple keys determine whether or not their encoded query pattern corresponds to a triple in the plaintext graph G. They are applied by using their query keys qk as input for the query function $f_{\texttt{+++}}$ (line 3). If at least one triple key of type `+++` does not match a triple, the query algebra QA has an empty solution sequence and the query q cannot be satisfied. In this case, query processing is stopped immediately and an empty result is returned (line 4). After having applied a triple key of type `+++`, it is removed from the query algebra QA (line 6). Thus, all triple keys tk remaining after the first step contain at least one unbound variable. Second, the algorithm applies all remaining triple keys to the ciphertext graph (line 9). Again, each triple key is applied by using its query key qk as input for a corresponding query function f. A query function returns a set of plaintext tuples, each of which represents a matching triple of the plaintext graph (line 11). These tuples are then bound to the query variables of the triple key in order to create the solution sequence of the key (line 12). A solutions sequence contains all possible solution mappings of a triple key's query variables. It can be interpreted as a table with its columns being the query variables and its rows being the different solution mappings [86]. The solution sequences of all triple keys are incrementally combined using the join operator $\bowtie$ (line 13). If two triple keys share the same query variables, a natural join is conducted on their solution sequences using these variables. Otherwise, the result of the join operation is the Cartesian product of the solution sequences. In its current design, T-Store conducts a join operation on plaintext solution sequences. The operation can be implemented using an arbitrary join algorithm

such as nested loop joins, sort-merge joins, or hash joins [242]. A detailed description of the use of join operators on SPARQL query results is provided in [86]. Implementing a join operation in T-Store which can be applied to encrypted solution sequences directly is left for future work. Finally, the algorithm returns the combined solution sequence (line 40). The columns of this solution sequence correspond to the query variables of all triple keys.

Algorithm 5.1 Applying a query algebra QA with multiple triple keys to a ciphertext graph G_C and its index I. Each triple key $tk \in \mathbb{K}_t$ is processed individually by applying its corresponding query key $qk \in \mathbb{P}_q$ to a query function f. The solution sequences of each triple keys are joined to create a single solution sequence.

```
1  function applyQueryAlgebra(QA, G_C, I)
2      for all tk ∈ QA with tk.type = +++ do          Apply all triple keys of type +++.
3          if f+++(tk.qk, G_C, I) = false then
4              return {}                              Stop if a triple key does not match.
5          end if
6          QA := QA \ {tk}                            Remove the current triple key.
7      end for
8      result := {}                                   Prepare the query result.
9      for all tk ∈ QA do                             Apply all remaining triple keys.
10         if tk.type = --- then
11             tuples := f---(tk.qk, G_C, I)          Apply the query key.
12             sequence := bindVariables(tk.vars, tuples)   Bind the query variables.
13             result := result ⋈ sequence            Join the solution sequences.
           ...
34         else if tk.type = -++ then
35             tuples := f-++(tk.qk, G_C, I)
36             sequence := bindVariables(tk.vars, tuples)
37             result := result ⋈ sequence
38         end if
39     end for
40     return result                                  Return the joined solution sequence.
41 end function
```

Algorithm 5.2 applies a `SELECT` query $q \in \mathbb{Q}_{\texttt{SELECT}}$ to a ciphertext graph G_C and its index I. As defined in Section 5.4.8, the query form of a `SELECT` query contains a list of query variables whose variable bindings shall be returned as the final query result. First, the algorithm retrieves the solution sequence of the query algebra by applying the function applyQueryAlgebra of Algorithm 5.1 to the query (line 2). Second, the query form of the query is evaluated by conducting a projection Π on the solution sequence (line 3). The projection removes the variable bindings of all query variables which are not defined in the query form. Thus, after having conducted the projection, the modified solution sequence only contains the variable bindings of query variables

which are also defined in the query form. Finally, the algorithm returns the modified solution sequence as the final query result (line 4).

Algorithm 5.2 Applying a `SELECT` query q to a ciphertext graph G_C and its index I. The result of the query is a list of variable bindings.

```
1 function applySelectQuery(q, G_C, I)
2     solutionSequence := applyQueryAlgebra(q.QA, G_C, I)
3     projectedSequence := Π_{v1,...,vn ∈ q.queryVariables}(solutionSequence)
4     return projectedSequence
5 end function
```

Algorithm 5.3 applies a `CONSTRUCT` query $q \in \mathbb{Q}_{\texttt{CONSTRUCT}}$ to a ciphertext graph G_C and its index I. The query form of a `CONSTRUCT` query contains a graph template which is used for creating a new graph $resultGraph \in \mathbb{G}$ based on the solution sequence of the query algebra. First, the algorithm processes the query algebra with the function applyQueryAlgebra of Algorithm 5.1 in order to retrieve an initial solution sequence (line 2). Second, an empty graph is created which serves as a basis for the new graph to be created (line 3). Third, the graph template is instantiated with each solution mapping of the initial solution sequence (line 5). This is done by substituting all query variables in the graph template with the variable bindings of a particular solution mapping. Each instantiation of the graph template corresponds to a new set of triples. Fourth, the set of triples is combined with the current result graph (line 6). Finally, the combined graph created from all instantiations of the graph template is returned as the final query result (line 8).

Algorithm 5.3 Applying a `CONSTRUCT` query q to a ciphertext graph G_C and its index I. The result of the query is a newly created graph.

```
1 function applyConstructQuery(q, G_C, I)
2     solutionSequence := applyQueryAlgebra(q.QA, G_C, I)
3     resultGraph := {}
4     for all mapping ∈ solutionSequence do
5         tripleSet := substitute(q.graphTemplate, mapping)
6         resultGraph := resultGraph ∪ tripleSet
7     end for
8     return resultGraph
9 end function
```

Algorithm 5.4 applies an `ASK` query $q \in \mathbb{Q}_{\texttt{ASK}}$ to an ciphertext graph G_C and its index I. An `ASK` query determines whether or not the query algebra of a query matches against a graph and returns a corresponding boolean value. The algorithm first applies the function applyQueryAlgebra of Algorithm 5.1 in order to retrieve an initial solution sequence (line 2). If the solution sequence contains at least one solution mapping, the algorithm returns $TRUE$ and $FALSE$ otherwise (line 3).

Algorithm 5.4 Applying an `ASK` query q to a ciphertext graph G_C and its index I. The result of the query is a boolean value.

```
1 function applyAskQuery(q, G_C, I)
2     solutionSequence := applyQueryAlgebra(q.QA, G_C, I)
3     return size(solutionSequence) > 0
4 end function
```

The four algorithms demonstrate how T-Store can be used for applying SPARQL queries of type `SELECT`, `CONSTRUCT`, and `ASK` to an encrypted graph and its corresponding index. Please note that these algorithms primarily serve as a basic proof-of-concept and can be further optimized towards their efficiency.

5.6. Performance of T-Store

This section assesses the performance of T-Store for processing queries on encrypted graphs. The performance is evaluated in different experiments, each of which covers a particular aspect of the query process. In order to assess the influence of the index on the query process, two different variants of T-Store are implemented and compared with each other. These variants are referred to as T-Store BSC and T-Store NDX, which correspond to the basic version and indexed version of T-Store, respectively. T-Store BSC directly operates on a ciphertext graph without using an index and implements the encryption phase, the authorization phase, and the query phase as depicted in Figure 5.4. In the query phase, T-Store BSC successively tries to decrypt all encrypted triples in the ciphertext graph with all query keys in a query algebra. If the decryption is successful, the decrypted triple satisfies the query pattern encoded in the query key. The triple is then included in the solution sequence of the query algebra. If the decryption fails otherwise, the triple does not satisfy the query pattern. This process is conducted for all encrypted triples in the ciphertext graph and all query keys in the query algebra. As T-Store BSC does not use any optimization, query processing solely depends on the size of the ciphertext graph and on the number of triple keys in the query algebra. T-Store NDX implements all four phases depicted in Figure 5.4 including the indexing phase. It applies an index to the ciphertext graph to identify all ciphertext triples which satisfy a particular query pattern. This speeds up query processing since not all ciphertext triples in the encrypted graph must be processed anymore. In addition, both variants of T-Store are compared with the plaintext triple store Sesame version 2.6.10 [54]. This triple store is chosen as it is implemented in the same programming language as the two variants of T-Store. This section first presents the setup of the experimental analysis and the implementation details of T-Store before the experimental results are discussed.

5.6.1. Experimental Setup and Implementation Details

The experimental analysis is divided into two phases which are the preparation phase and the query processing phase. The preparation phase combines the encryption phase

and the indexing phase as depicted in Figure 5.4 and the query processing phase corresponds to the query phase. In the preparation phase, different plaintext graphs are generated, encrypted, indexed, and loaded into memory. The runtime of encrypting a graph, creating its index, and loading the created files is measured. In addition, the size of the created files is measured as well. In the query processing phase, multiple queries with different numbers of triple keys are applied to the ciphertext graphs. Again, the runtime of applying each query is measured. In order to avoid interference with statistical outliers, each operation is performed ten times and the mean value is calculated.

As experimental data, the Berlin SPARQL benchmark (BSBM) [40] is used which is designed to assess the performance of different SPARQL implementations. BSBM provides tools for generating synthetic RDF graphs and SPARQL queries which simulate a user browsing through the graphs. The tools are used to create five different RDF graphs ranging from 10,000 triples to 200,000 triples. Each graph is serialized using N-Triples [25] and stored in a separate file. Queries are taken from the explore scenario of the BSBM framework which provides twelve different queries. The queries differ in their number of SPARQL triple patterns and in the size of their solution sequence. Queries that use SPARQL features which are not yet supported by T-Store such as `OPTIONAL` and `FILTER` are modified by removing the unsupported parts. Queries using `UNION` are split into separate queries and the runtime of these queries is summed up. The modified queries are available online at the website of this thesis which is referenced in the conclusion.

As described in Section 5.5.2, the array size z is a parameter of the index of T-Store NDX. In order to measure the influence of the array size on the index, twelve indexes with different values of z are created for the BSBM graph with 200,000 triples. For a fair comparison between T-Store and Sesame, all files required for processing a query are loaded into main memory before the actual query processing is conducted. This minimizes potential side effects from accessing the hard drive when applying a query. T-Store BSC only loads the ciphertext graph while T-Store NDX loads the ciphertext graph and its index. Sesame loads the plaintext graph using its `MemoryStore` which stores all triples in main memory. When loading a graph, the `MemoryStore` automatically creates an index for the graph from scratch. The index provides a mapping from every subject, predicate, and object in the graph to a one-dimensional array of all triples in which they occur. All experiments were conducted on a system with 50 GB memory and an Intel® Xeon® CPU with 3.00 GHz running Ubuntu GNU/Linux version 14.04.2.

Both variants of T-Store are implemented in Java and use the Java Cryptography Extension (JCE) for all cryptographic operations. JCE is an official Java API which provides different encryption algorithms and hash algorithms. As the encryption function ξ, the Advanced Encryption Standard (AES) [213] with a key length of 256 bits is used. The algorithm and the used key length are proven of high security and are recommended by the National Institute of Standards and Technology (NIST) [215]. AES is a block-based symmetric encryption scheme which operates on blocks of 16 bytes. In order to encrypt arbitrary plaintext data, the data is first split into blocks which are then encrypted. If the data is not a multiple of the block size, it is padded before the encryption, which may slightly increase the size of the resulting ciphertext. As the basic

hash function λ, the Secure Hash Algorithm 2 (SHA-2) [218] is used with an output length of 256 bits as recommended by NIST [215]. Thus, the set $\{0,1\}^d$ of all bit strings with fixed length d corresponds to the set $\{0,1\}^{256}$, i. e., $d = 256$. As feature by design, the length of the hash value and the key length of the encryption function ξ are the same. This allows it to directly use the computed hash values as encryption keys ek. The combining function ϱ uses a value of N with a length of 2048 bit. The index applied by T-Store NDX is based on the Java class `TreeMap` which implements a red-black tree [23]. A red-black tree is a self-balancing BST which stores a color for each node in order to balance the tree.

5.6.2. Preparation Phase

Figure 5.6 shows the file size of the created plaintext graph, the ciphertext graph, the indexed ciphertext graph, and the file size of the created index. As depicted, increasing the size of a plaintext graph by a certain factor roughly increases the size of the other files by the same factor as well. The ciphertext graphs created by T-Store BSC are about 4.5 times larger than their corresponding plaintext graphs. For the indexed ciphertext graphs of T-Store NDX, this ratio is about 4.6 due to the additional ciphertext identifiers id_c stored in each file. The files storing the index are about 1.02 times larger than the respective plaintext graphs. Thus, the total amount of ciphertext data created by T-Store NDX for a particular plaintext graph is about 5.62 times larger than the file size of this graph.

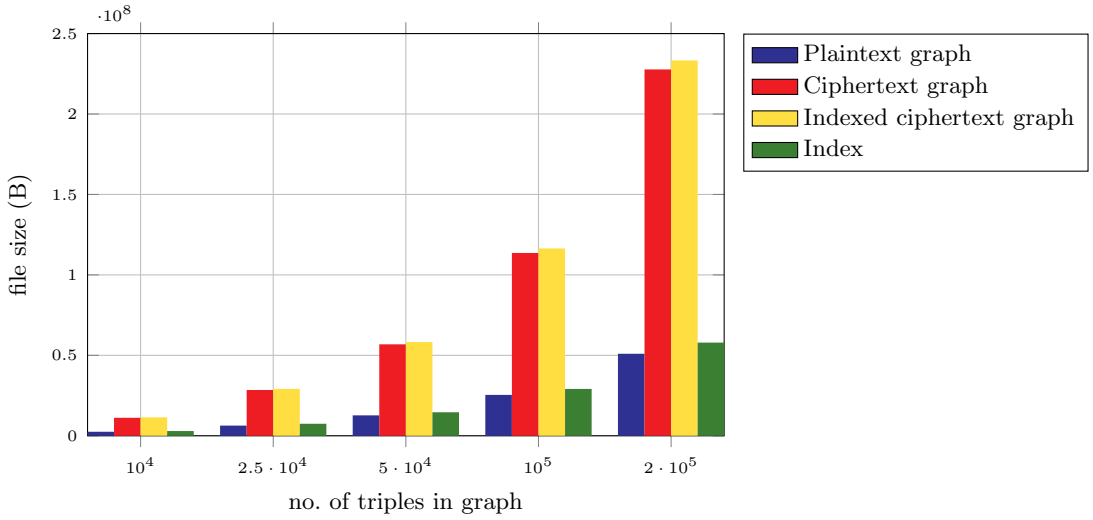


Figure 5.6.: File size of the plaintext graph, ciphertext graph without an index, indexed ciphertext graph, and index. The file sizes are given in bytes (B).

Figure 5.7 depicts the runtime of the different steps in the preparation phase. Figure 5.7a shows the runtime of encrypting the different plaintext graphs with both T-Store BSC and T-Store NDX. T-Store BSC only creates the ciphertext graph and T-Store NDX

creates both the indexed ciphertext graph and its corresponding index. The index of a ciphertext graph is created simultaneously while encrypting its corresponding plaintext graph. This allows it to process each triple only once which reduces the overall runtime of the preparation phase of T-Store NDX. As depicted in Figure 5.7a, the overhead of creating the index hardly influences the overall runtime of T-Store NDX. Thus, the encryption time of T-Store NDX and T-Store BSC is roughly the same. The runtime of encrypting a plaintext graph is linear in both variants of T-Store with respect to the number of triples in the graph. Increasing the size of the plaintext graph by a certain factor also increases the runtime required for its encryption by the same factor. Both variants of T-Store encrypt the BSBM graph with $2 \cdot 10^5$ triples in about 104 minutes. The large encryption time is mainly caused by the combining function ϱ which is used for creating eight different encryption keys for each triple in the graph.

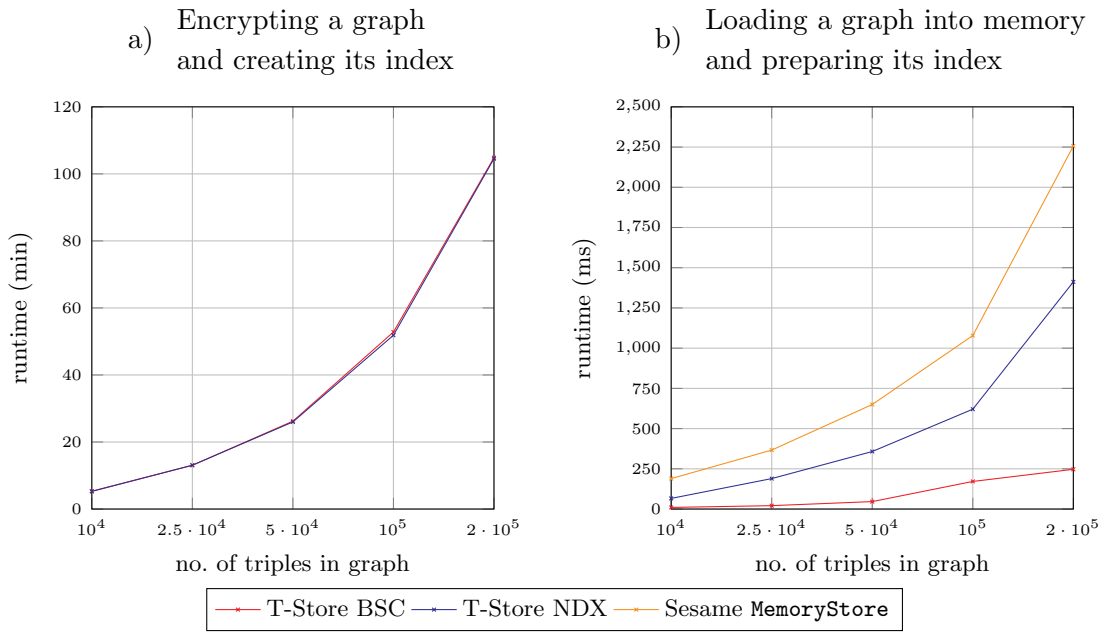


Figure 5.7.: Runtime of preparing plaintext graphs of different sizes. The preparation includes the encryption of the plaintext graph and the creation of the index as well as and loading all created files into main memory.

Figure 5.7b shows the runtime of loading all files into memory which are required for processing a query. T-Store BSC loads the ciphertext graph and maps it to several Java byte arrays which do not require any further processing. In contrast, T-Store NDX loads both the indexed ciphertext graph and its index and maps the loaded data to instances of Java classes. This is necessary to re-create the index stored in the file in such a way that it can be used for query processing. However, this process requires more time than simply loading byte arrays into memory. Thus, T-Store NDX has a higher runtime than T-Store BSC for loading all necessary files. The Sesame `MemoryStore` loads the plaintext graph into memory and simultaneously creates a new index from scratch. The creation of

the index is conducted while the graph is being loaded and cannot be separated from the loading process. This makes the index creation an integral part of the loading process. As creating an index requires more time than parsing an already existing index stored in a file, the Sesame `MemoryStore` has a higher loading time than T-Store NDX.

5.6.3. Query Processing Phase

Figure 5.8 depicts the runtime of applying the BSBM queries to graphs of different size for the two variants of T-Store and the Sesame `MemoryStore`. For a better comparison between the different results, the figure also shows the number of triples in a query's solution sequence which is depicted in Figure 5.8d. The query processing time generally depends on the number of a query's triple keys, the size of its solution sequence, and on the size of the queried graph. Figure 5.8a shows the runtime of processing a query with T-Store BSC. As depicted, the runtime is mainly affected by the number of triple keys and the size of the queried graph and hardly influenced by the size of a query's solution sequence. Increasing the size of a graph by a certain factor roughly increases the runtime of querying the graph by the same factor. Similarly, increasing the number of triple keys in a query by a given factor increases the query's runtime by the same factor as well. This linear dependency is due to the fact that T-Store BSC does not use any query optimization. Instead, it always processes all ciphertext triples with all triple keys of the query. The runtime of processing the BSBM query #5 slightly differs from this observation. As depicted in Figure 5.8d, the solution sequence of this query is larger than the solution sequences of the other queries. A large solution sequence increases the overall processing time of a query due to the larger runtime that is required for joining the individual solution sequences of the query's triple keys. Furthermore, one triple key of query #5 is of type `-+-` and specifies a common RDFS predicate as a query parameter. This results in a large solution sequence of the key which has to be further processed during query execution.

Figure 5.8b shows the runtime of processing the BSBM queries with T-Store NDX. As depicted, the runtime is mainly affected by the size of a query's solution sequence and hardly influenced by the number of its triple keys or the size of the queried graph. Due to the use of an index, query processing in T-Store NDX is significantly faster than in T-Store BSC. The only exception is query #5 whose processing time is almost the same. Thus, the index is not yet sufficient for speeding up queries which produce large solution sequences. Processing the two queries #5 and #6 is slower than processing other queries which have a similar number of triple keys. The higher runtime of these two queries is based on their large solution sequences as depicted in Figure 5.8d. The queries #7 and #9 are processed even faster than in the Sesame `MemoryStore`. This is due to the fact that both queries consist of one very specific triple key which produces a solution sequence with a single triple. The index used in T-Store NDX basically stores the solution sequences of all possible triple keys. Thus, assembling the solution sequence of a very specific triple key essentially corresponds to traversing the index tree once and decrypting the identified ciphertext triples.

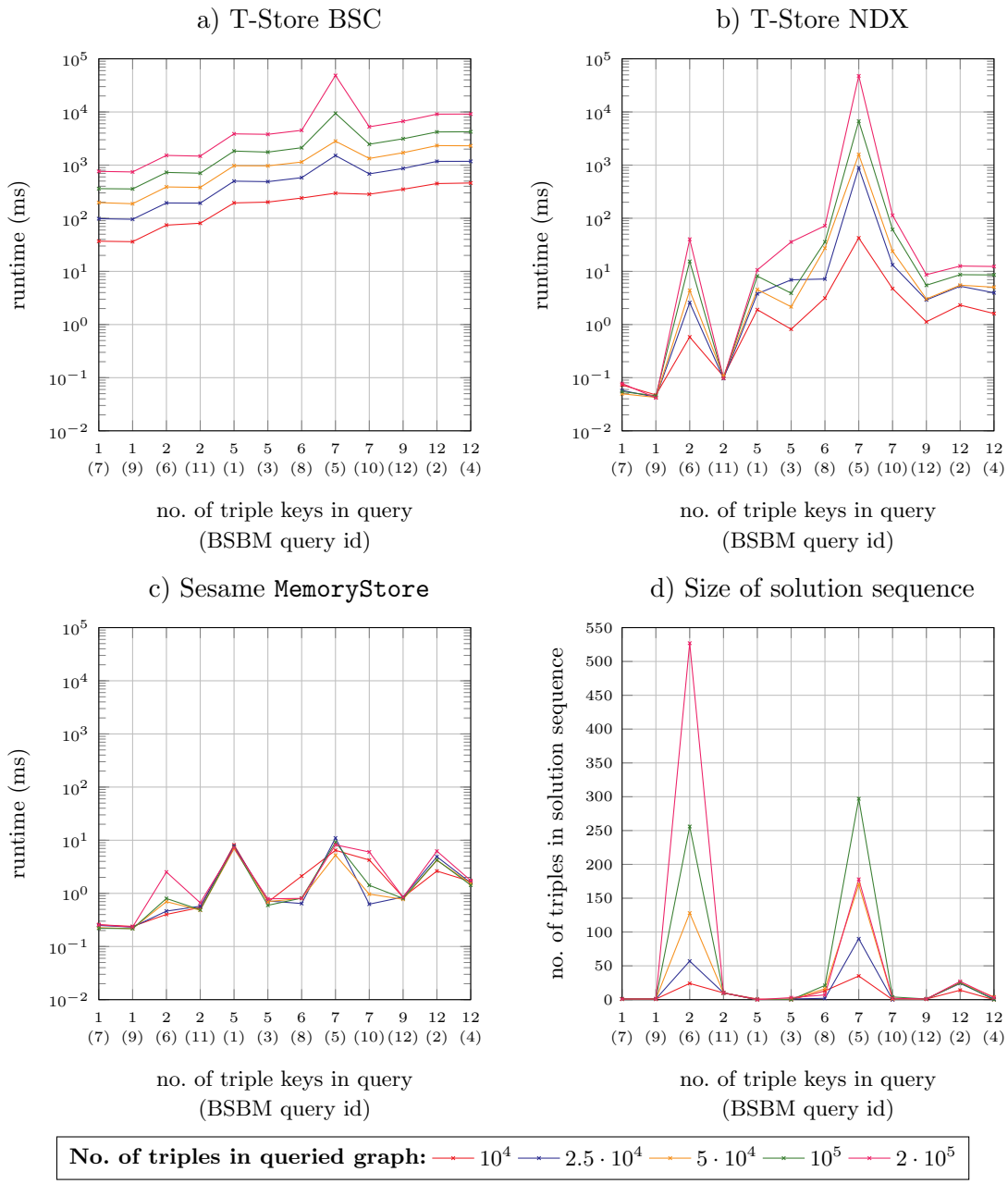


Figure 5.8.: Runtime of applying the BSBM queries to graphs of different sizes and their respective solution sequences. The queries differ in their number of triple keys and in the size of their solution sequences.

Figure 5.8c depicts the runtime of processing the BSBM queries with the Sesame `MemoryStore`. As depicted, the runtime mainly depends on the number of triple keys in the query and on the size of the query's solution sequence. The size of the queried

plaintext graph affects the runtime of the query only indirectly through a potentially larger solution sequence. As depicted, the queries #1, #7, and #9 are not significantly affected by the size of the queried graph since they only produce small result sets. Although query #1 produces a small query result, its runtime is almost as large as query #5 which produces a large result. This is due to the fact that both queries contain the same triple key of type `-+-` which specifies a common RDFS predicate as a query parameter. Applying this triple key results in a large solution sequence which also affects the overall runtime of processing the whole query.

5.6.4. Influence of the Array Size z

The index of T-Store NDX basically maps a query key to one-dimensional arrays of size z. These arrays contain the identifiers of all ciphertexts which can be decrypted with the same query key. As described in Section 5.5.2, the array size z is a parameter of the index and influences its size and efficiency. Storing all ciphertext identifiers that match a particular query key requires at least 1 array and at most $\frac{n}{z}$ arrays with $n \in \mathbb{N}$ being the number of triples in the plaintext graph. The specific number of arrays required for storing all ciphertext identifiers depends on the structure of the plaintext graph and on the number of possible triple keys which contain the query keys. In general, smaller values of z require more traversals of the index as the ciphertext identifiers are stored in more arrays. This increases the runtime of processing a query. In contrast, larger values of z result in less arrays which are stored in the index. However, some of these arrays may require additional padding if they contain less than z ciphertext identifiers. Padding a high number of arrays increases the size of the index. Thus, the value of z must be optimized towards the graph structure and the average size of all non-empty solution sequences produced by the triple keys.

Figure 5.9 depicts the runtime of encrypting and indexing the BSBM graph with 200.000 triples for different values of z ranging from 1 to 12. The figure also shows the size of the resulting index in bytes. Since the index is created while encrypting the graph, the depicted runtime covers the time for creating both the ciphertext graph and the index. As shown in Figure 5.9a, the runtime of creating the indexed ciphertext graph and its index is almost constant and hardly affected by the value of z. In contrast, the file size of the index is significantly influenced by the array size z as depicted in Figure 5.9b. As described in Section 5.5.2, each array contains z ciphertext identifiers and an additional bit which marks whether or not there are more arrays of the same set id_S. Each ciphertext identifier is stored as a Java integer variable which requires 4 bytes per value. Since Java does not allow to store a single bit separately, the bit is stored as an additional byte. Thus, storing an array with z ciphertext identifiers requires $4 \cdot z + 1$ bytes in total. In order to encrypt a plaintext array with AES, the number of bytes that the array occupies must be a multiple of the AES block size of 16 bytes. If this is not the case, the array must be padded with $v \in \mathbb{N}$ additional bytes so that $4 \cdot z + 1 + v \bmod 16 = 0$. For example, an array size of $z = 1$ requires a padding of 11 bytes since $4 \cdot 1 + 1 + 11 = 16$ and $16 \bmod 16 = 0$. The same padding of 11 bytes is also used for $z = 1 + k$ with $k \in \{x \mid x \bmod 4 = 0\}$ being a multiple of 4. Arrays storing

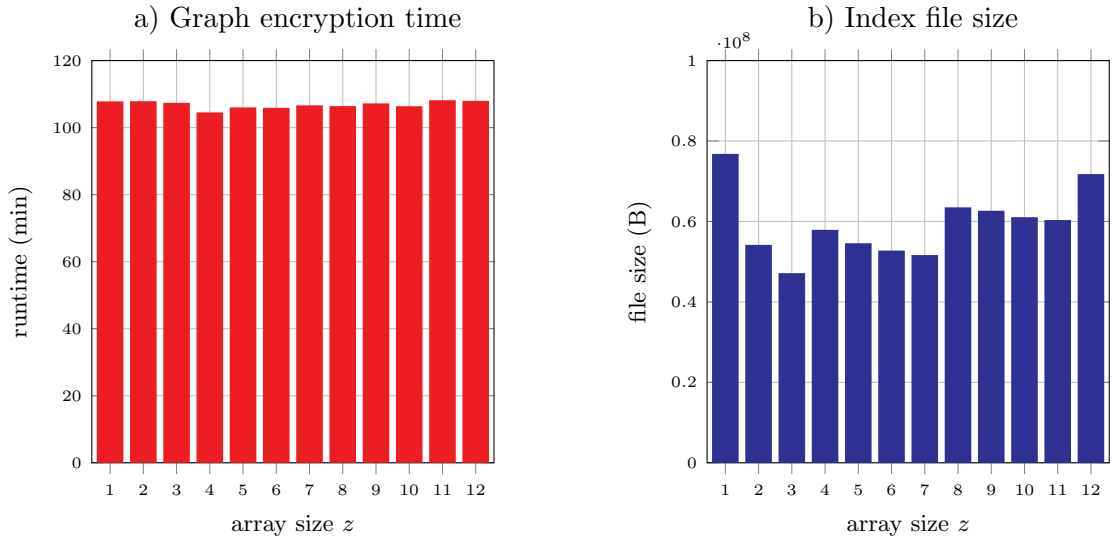


Figure 5.9.: Effects of the array size z on the index for the BSBM graph with $2 \cdot 10^5$ triples. The file sizes are given in bytes (B) and the runtime of encrypting the graph and creating the indexes are given in minutes (min).

$z = 2$ ciphertext identifiers require a padding of 7 bytes, arrays with a size of $z = 3$ are padded with 3 additional bytes, and arrays containing $z = 4$ ciphertext identifiers contain 15 bytes of additional padding. The same padding size is also used for $z = 2+k$, $z = 3+k$, and $z = 4+k$ with $k \in \{x \mid x \bmod 4 = 0\}$. Thus, there are only four different paddings which are used for every fourth value of z. This is also reflected in the overall size of the index as depicted in Figure 5.9b. As the diagram shows, an array size of $z = 1$ introduces a large overhead due to the high amount of padding bytes. In general, the storage efficiency of an array size can be represented by the ratio between the number of bytes storing ciphertext identifiers and the number of padding bytes. As each ciphertext identifier occupies four bytes, this ratio is $\frac{4 \cdot z}{v}$ with $v \in \mathbb{N}$ being the number of padding bytes. A higher value corresponds to a better efficiency and a smaller value corresponds to a worse efficiency. For a array size $z = 1$, the efficiency ratio is $\frac{4 \cdot 1}{11}$ which is smaller than the efficiency ratio of other array sizes sharing the same amount of padding bytes. For example, an array size of $z = 5$ has a higher efficiency ratio of $\frac{4 \cdot 5}{11}$. As shown in Figure 5.9b, this results in less bytes for storing the index. The diagram also shows that an array size of $z = 3$ results in an index which requires the least amount of bytes to be stored. In the example BSBM graph, the average number of triples matching a triple key, which does not produce an empty solution sequence, is 2.32. Thus, an array size of $z = 3$ achieves the best efficiency for storing ciphertext identifiers as it results in the least amount of padding bytes on average.

Figure 5.10 shows how different values of the array size z affect the runtime of applying the individual BSBM queries to the graph with 200.000 triples. As depicted, the array size mostly affects those queries which return a small solution sequence such as the

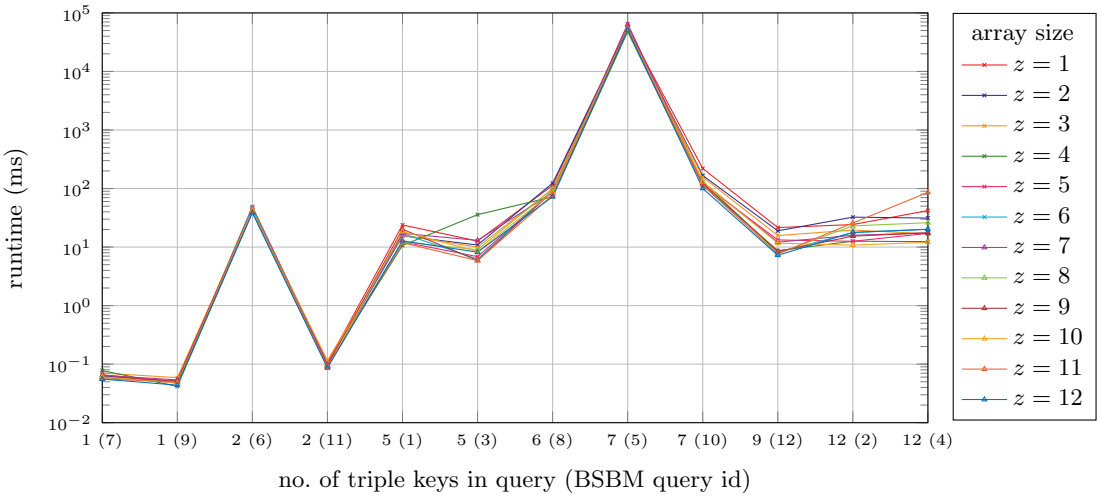


Figure 5.10.: Runtime of applying the BSBM queries to the indexed ciphertext graph with 200.000 triples using different values of the array size z. The diagram shows how different values of z influence the processing time of the individual queries. The queries differ in the number of their triple keys and the size of their solution sequence.

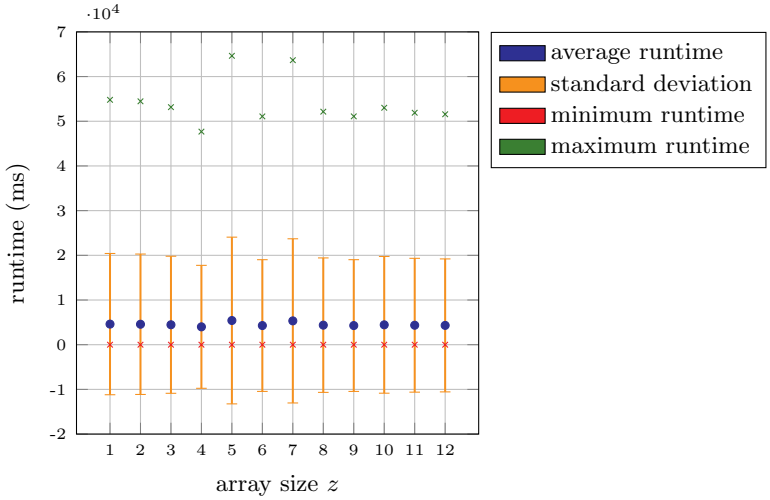


Figure 5.11.: Average runtime and standard deviation of applying all BSBM queries to the indexed ciphertext graph with 200.000 triples using different values of the array size z. The diagram also shows the minimum and maximum runtime of the queries.

queries #1 to #4. The diagram shows that some queries may benefit from a particular array size while the processing time of other queries increases for the same array size. Thus, there is no array size which simultaneously reduces the runtime of all queries.

Figure 5.11 shows how different values of the array size z influence the average runtime of all BSBM queries. The diagram depicts the minimum runtime, maximum runtime, average runtime, and standard deviation of processing all twelve queries for each of the different values of z. As depicted, an array size $z = 4$ results in the smallest average runtime of processing the queries as well as the smallest standard deviation. The maximum runtime and the minimum runtime of all queries are also smaller when using a value of $z = 4$ than for other values. Thus, it can be stated that this array size is best suited for minimizing the average runtime when querying the BSBM graphs. However, the optimal value of z may be different for other graphs as the optimal value depends on the graph's structure.

5.7. Cryptanalysis of T-Store

T-Store implements confidentiality of RDF graphs by restricting access to the graph's triples to authorized users only. Accessing the triples is considered to be authorized if a user has received a set of corresponding authorization keys from the data owner. Obtaining any other triples which are not explicitly permitted by the data owner is considered to be unauthorized. This section describes how T-Store achieves confidentiality of RDF graphs. First, the cryptographic security of creating and applying authorization keys is analyzed. This analysis follows directly from the design of T-Store as covered in Section 5.5. Afterwards, an attack model is presented which introduces different attacks on this design. Finally, this section analyzes how the attacks are prevented by T-Store.

5.7.1. Achieving Confidentiality of RDF Graphs

Figure 5.4 divides the overall process of T-Store into four different phases which include the encryption phase, the authorization phase, and the query phase. In the encryption phase, all triples $t \in G$ of a plaintext graph $G \in \mathbb{G}$ are encrypted for all eight triple pattern variants to create the ciphertext graph $G_C \in \mathbb{G}_C$. The encryption is conducted with symmetric encryption keys $ek \in \mathbb{K}_e$ which encode all bound parts of a SPARQL triple pattern. In the authorization phase, the data owner creates an authorization key $ak \in \mathbb{K}_a$ for each allowed triple pattern and sends it to a user. An authorization key is a preliminary query key $qk \in \mathbb{K}_q$ and encodes a data owner's restriction pattern $r \in \mathbb{P}$. In the query phase, a user combines a received authorization key ak with a self-defined user pattern $u \in \mathbb{P}_q$ to create a query key qk. A query key encodes a query pattern which contains all bound parts of a SPARQL triple pattern. It is also a symmetric key and created similarly to an encryption key. A query key qk is applied to a ciphertext graph G_C by decrypting all compatible ciphertext triples $c \in G_C$. A ciphertext triple is compatible with a query key if its corresponding plaintext triple t matches the query pattern encoded in the key. The result of this application is a set of all matching plaintext triples. In order to ensure the confidentiality of a plaintext graph G, the set must not contain any triple $t \in G$ which does not match the query pattern. This implies that decrypting a ciphertext triple c with a query key qk must only be successful if qk encodes the same query pattern as the encryption key ek which was used for creating the triple c.

If the two keys encode different query patterns, decrypting the ciphertext triple c must not be successful. Otherwise, a user would receive an unauthorized plaintext triple $t \in G$ which would directly contradict with the confidentiality of the plaintext graph G. In the following, the individual steps of the encryption phase, the authorization phase, and the query phase are analyzed in more detail. As running example, all steps are conducted on a plaintext triple $t = (s, p, o) \in G$ which is processed for the pattern variant `++-`. Please note that this example is without loss of generality and can be applied to other triples and other pattern variants as well.

In the encryption phase, each plaintext triple t is encrypted for all eight pattern variants with a corresponding encryption key ek. Encryption keys are created from basic keys bk as defined in Table 5.1. Encrypting the plaintext triple (s, p, o) for the pattern variant `++-` requires an encryption key $ek_{\texttt{sp-}} \in \mathbb{K}_e$. The key encodes the subject and the predicate of the triple and is created as follows:

$$ek_{\texttt{sp-}} = \lambda\left(\varrho\left(bk_{\texttt{++-}}, \{w_{\texttt{s}}||s, w_{\texttt{p}}||p\}\right)\right) \tag{5.22}$$

$$= \lambda\left(\mathsf{bit}\left(\mathsf{int}\left(bk_{\texttt{++-}}\right)^{\mathsf{int}(\lambda(w_{\texttt{s}}||s))\cdot\mathsf{int}(\lambda(w_{\texttt{p}}||p))} \bmod N\right)\right) \tag{5.23}$$

The prefix $w_{\texttt{s}}$ identifies the triple's subject and $w_{\texttt{p}}$ marks its predicate. The resulting encryption key $ek_{\texttt{sp-}}$ is used for encrypting the triple (s, p, o) as defined in Table 5.2. The key $ek_{\texttt{sp-}}$ corresponds to a bit string of length d as it is $\mathbb{K}_e \subset \{0,1\}^d$. In the authorization phase, the data owner creates a restriction pattern $r = (s_r, \texttt{?}, \texttt{?}) \in \mathbb{P}_q$ and combines it with a basic key $bk_{\texttt{++-}} \in \mathbb{K}_b$ to create an authorization key $ak_{\texttt{ru?}} \in \mathbb{K}_a$. As defined in Table 5.8, the key $ak_{\texttt{ru?}}$ is computed as follows:

$$ak_{\texttt{ru?}} = \varrho\left(bk_{\texttt{++-}}, \{w_{\texttt{s}}||s_r\}\right) \tag{5.24}$$

$$= \mathsf{bit}\left(\mathsf{int}\left(bk_{\texttt{++-}}\right)^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))} \bmod N\right) \tag{5.25}$$

Again, the prefix $w_{\texttt{s}}$ identifies the triple's subject. The data owner sends the resulting authorization key $ak_{\texttt{ru?}}$ to the user. In the query phase, the user creates a user pattern $u = (\texttt{?}, p_u, \texttt{?})$ and combines it with the received authorization key $ak_{\texttt{ru?}}$ into a query key $qk_{\texttt{ru?}} \in \mathbb{K}_q$. As defined in Table 5.9, the query key is created as follows:

$$qk_{\texttt{ru?}} = \lambda\left(\varrho\left(ak_{\texttt{ru?}}, \{w_{\texttt{p}}||p_u\}\right)\right) \tag{5.26}$$

$$= \lambda\left(\mathsf{bit}\left(\mathsf{int}\left(ak_{\texttt{ru?}}\right)^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_u))} \bmod N\right)\right) \tag{5.27}$$

$$= \lambda\left(\mathsf{bit}\left(\left(\mathsf{int}\left(bk_{\texttt{++-}}\right)^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))} \bmod N\right)^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_u))} \bmod N\right)\right) \tag{5.28}$$

$$= \lambda\left(\mathsf{bit}\left(\mathsf{int}\left(bk_{\texttt{++-}}\right)^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))\cdot\mathsf{int}(\lambda(w_{\texttt{p}}||p_u))} \bmod N\right)\right) \tag{5.29}$$

Similar to an encryption key, the resulting query key $qk_{\texttt{ru?}}$ is also a bit string of length d as it is $\mathbb{K}_q \subset \{0,1\}^d$. The query key $qk_{\texttt{ru?}}$ is identical to the encryption key $ek_{\texttt{sp-}}$ if the data owner's restriction pattern r is defined as $r = (s, \texttt{?}, \texttt{?})$ and if the user chooses a user pattern $u = (\texttt{?}, p, \texttt{?})$. In this case, it is $s_r = s$ and $p_u = p$ and both keys are mapped to

the same bit string. Thus, the query key $qk_{\texttt{ru?}}$ can be used for decrypting the ciphertext triple which was created with the encryption key $ek_{\texttt{sp-}}$.

Ensuring the confidentiality of the plaintext graph requires that different query keys qk do not collide with each other. Query keys collide with each other if they encode different query patterns but are mapped to the same bit string. Similarly, different encryption keys ek must not collide with each other as well. T-Store reduces the possibility of such collisions by ensuring that each step of creating encryption keys and query keys is collision-resistant as well. First, the data owner chooses eight different basic keys bk when encrypting a plaintext graph. As these keys form the foundation for creating encryption keys and query keys, choosing different basic keys reduces the possibility of key collisions. Second, the data owner chooses three distinct prefixes $w_{\mathtt{s}}$, $w_{\mathtt{p}}$, and $w_{\mathtt{o}}$. These prefixes are used for combining the bound parts of a query pattern with a basic key and indicate the position of each bound part in the pattern as shown in Equations 5.23 and 5.29. Using distinct prefixes for each bound part ensures that query patterns which share the same bound parts at different positions result in different keys. For example, the query patterns $(x, \texttt{?}, y)$ and $(y, \texttt{?}, x)$ are both of type `+-+` and contain the two values x and y. However, the patterns differ in their bound subject and object. The prefixes $w_{\mathtt{s}}$ and $w_{\mathtt{o}}$ ensure that they are mapped to different encryption keys and query keys as long as it is $w_{\mathtt{s}} \neq w_{\mathtt{o}}$. In this case, it is $\mathsf{int}(\lambda(w_{\mathtt{s}}||x)) \cdot \mathsf{int}(\lambda(w_{\mathtt{o}}||y)) \neq \mathsf{int}(\lambda(w_{\mathtt{s}}||y)) \cdot \mathsf{int}(\lambda(w_{\mathtt{o}}||x))$ which results in different exponents as they are used in Equations 5.23 and 5.29. Third, the data owner uses a collision-resistant basic hash function λ for creating encryption keys and query keys. As described in Section 4.5.4, the function λ is collision resistant if it is difficult to find any two different input values with the same hash value [56]. A collision resistant basic hash function prohibits collisions between encryption keys and query keys which encode different query patterns and ensures that such keys are mapped to different bit strings. As described in Section 5.6.1, T-Store uses SHA-2 with an output length of 256 bit as the basic hash function λ. Although collision attacks on reduced versions of this hash function have already been published [258, 158, 201], an attack on complete SHA-2 has not been found yet. Thus, this basic hash function can still be considered as collision resistant. Finally, T-Store ensures that the combination of all bound parts of a query pattern does not produce any collisions as well. For example, finding a collision of two different query patterns $(s_1, \texttt{?}, o_1)$ and $(s_2, \texttt{?}, o_2)$ corresponds to solving the following equation:

$$\mathsf{int}(bk)_{\texttt{+-+}}^{\mathsf{int}(\lambda(w_{\mathtt{s}}||s_1)) \cdot \mathsf{int}(\lambda(w_{\mathtt{o}}||o_1))} \bmod N = \mathsf{int}(bk_{\texttt{+-+}})^{\mathsf{int}(\lambda(w_{\mathtt{s}}||s_2)) \cdot \mathsf{int}(\lambda(w_{\mathtt{o}}||o_2))} \bmod N \quad (5.30)$$

Computing such collisions is similar to finding collisions in MuHash [29]. As described in Section 4.5.9, MuHash is considered to be collision resistant if the basic hash function λ is collision-resistant and the modulus N is large enough. In this case, the cryptographic security of MuHash can be reduced to the discrete logarithm problem which is considered to be hard to solve [197]. As described in Section 5.6.1, the combining function ϱ of T-Store uses a modulus N with a size of 2048 bit which is sufficiently large. Since the used basic hash function λ can also be considered as collision resistant, solving Equation 5.30 in order to find two colliding query patterns can be considered as hard.

In summary, it can be stated that the possibility of colliding encryption keys and query keys is sufficiently small. Thus, the ciphertexts stored in a ciphertext graph G_C can only be decrypted with the correct query key qk. As long as correct query keys can only be created by authorized users, the confidentiality of the plaintext graph is ensured.

5.7.2. Attack Model

An *attacker* tries to retrieve triples from the plaintext graph without having received a proper authorization from the data owner. Although an attacker may try to retrieve other information about the graph as well such as its size and connectivity, this information is only used as an intermediate step to obtain particular triples. Thus, accessing triples is the ultimate goal of an attacker while all other attacks are used to achieve this goal. Attackers can generally be distinguished between authorized users and unauthorized parties. Authorized users have received legitimate authorization keys from the data owner. In their attacks, they want to retrieve more triples from the plaintext graph than the data owner has originally intended. In contrast, unauthorized parties do not have any authorization key. However, they have access to the ciphertext graph and possibly some general knowledge about the plaintext graph as well. As authorized users also have access to this information, all attacks which can be conducted by an unauthorized party can also be executed by authorized users. This section introduces relevant attacks on T-Store for authorized users and unauthorized parties. Each of these attacks is analyzed in more detail in the subsequent sections.

AC.1: Guessing basic keys

An attacker guesses a valid basic key $bk \in \mathbb{K}_b$ by trying all possible basic keys of the key space $\mathbb{K}_b$. The goal of this attack is to find a basic key which can be used for creating arbitrary query keys $qk \in \mathbb{K}_q$. This attack corresponds to a *brute force attack* [270]. It can be conducted by unauthorized parties and only requires access to the ciphertext graph.

AC.2: Guessing query keys

An attacker guesses a valid query key $qk \in \mathbb{K}_q$ by trying all possible query keys of the key space $\mathbb{K}_q$. The goal of this attack is to find a query key which can be used for decrypting ciphertext triples $c \in \mathbb{T}_C$. This attack is also a brute force attack and can be conducted by unauthorized parties with access to the ciphertext graph.

AC.3: Extracting basic keys

An attacker has access to an authorization key $ak \in \mathbb{K}_a$ and extracts the encoded basic key $bk \in \mathbb{K}_b$. The goal of this attack is to retrieve a valid basic key which can be used for creating different authorization keys and query keys for a particular query pattern variant. The attack can be conducted by authorized users and requires a valid authorization key ak.

AC.4: Computing basic keys

An attacker has access to two different authorization keys $ak_1, ak_2 \in \mathbb{K}_a$ which encode the same basic key $bk \in \mathbb{K}_b$. The attacker computes the basic key bk by

utilizing the relationship between the keys ak_1 and ak_2. The goal of this attack is also to retrieve a valid basic key which can be used to create arbitrary authorization keys and query keys for a particular query pattern variant. The attack can be conducted by authorized users and requires two valid authorization key ak_1 and ak_2 which encode the same basic key bk.

AC.5: Reducing authorization keys

An attacker has access to an authorization key $ak \in \mathbb{K}_a$ which encodes a restriction pattern $r \in \mathbb{P}_q$ with two or three bound triple parts. The attacker reduces the authorization key by removing parts of its encoded restriction pattern so that the resulting key encodes only one bound triple part. The goal of this attack is to create an authorization key with less restrictions than the original key ak. This allows the attacker to define more flexible user patterns $u \in \mathbb{P}_q$ and thus to decrypt more ciphertext triples $c \in G_C$ from the encrypted graph $G_C \in \mathbb{G}_C$. The attack can be conducted by authorized users and requires access to a valid authorization key ak.

AC.6: Analyzing the ciphertext frequency

An attacker knows the frequency of potential plaintext triples and compares it with the frequency of the ciphertext triples in the encrypted graph G_C. The attacker matches the ciphertext triples with those plaintext triples that share the same number of occurrences. The goal of this attack is to identify the contents of ciphertext triples with certain probability without decrypting them first [296]. For example, two predicates which are most used in many RDF graphs are `rdf:type` and `rdfs:seeAlso` [149]. An attacker searches for the most frequent ciphertext triples and identifies their predicates as being either `rdf:type` or `rdfs:seeAlso` with certain probability. The attack requires deterministic encryption which encrypts identical plaintexts to identical ciphertexts. In this case, the distribution of plaintext triples is identical to the distribution of their corresponding ciphertext triples. The attack is known as a *frequency-based attack* [296]. It can be conducted by unauthorized parties and requires access to the ciphertext graph G_C as well as knowledge about the frequency of potential plaintext triples.

AC.7: Analyzing the ciphertext size

An attacker knows the size of potential plaintext triples and compares it with the size of the ciphertext triples in the encrypted graph G_C [296]. Here, size covers the number of bits in the bit string representation of plaintext triples and ciphertext triples. The attack is similar to a frequency-based attack but compares the size instead of the frequency. The goal of this attack is to identify the contents of ciphertext triples without decrypting them. An attacker can exploit her knowledge about widely-used classes and properties in order to guess with certain probability the contents of a ciphertext graph G_C. The attack requires that the size of a ciphertext is proportional to the size of its corresponding plaintext. This is the case if the encryption function ξ produces larger ciphertexts for larger plaintexts and smaller ciphertexts for smaller plaintexts. The attack is known as a *size-based*

attack [296]. It can be conducted by unauthorized parties and requires access to the ciphertext graph G_C as well as knowledge about the size of potential plaintext triples.

AC.8: Reasoning on query results
An attacker has access to a limited set of plaintext triples $t \in G$ which may be retrieved by applying legitimate queries to the ciphertext graph G_C. The attacker conducts reasoning on the plaintext triples in order to obtain additional triples from the plaintext graph G. The goal of this attack is to retrieve such triples t which are not authorized by the data owner. The attack can be conducted by authorized users who have access to a valid authorization key ak to apply legitimate queries to the ciphertext graph G_C.

AC.9: Analyzing the graph's characteristics
An attacker analyzes a ciphertext graph G_C in order to obtain structural information about the graph such as its size or connectivity. The attacker can exploit this information to achieve different goals. The information can be used to decide whether or not two different ciphertext graphs encrypt the same plaintext graph. In addition, the attacker can also compare the characteristics of a ciphertext graph G_C with the characteristics of a given plaintext graph G in order to decide whether or not the graph G_C is an encryption of the graph G. The attack can be conducted by unauthorized parties and requires access to a ciphertext graph G_C and possibly to a plaintext graph G as well.

The attacks **AC.1** to **AC.5** are cryptographic attacks on a ciphertext graph whereas the attacks **AC.6** to **AC.8** exploit an attacker's knowledge about plaintext triples. In contrast, the attack **AC.9** assesses the characteristics of a ciphertext graph as a whole and does not necessarily reveal any triples of its corresponding plaintext graph.

5.7.3. Guessing Basic Keys (AC.1)

The data owner creates eight different basic keys $bk \in \mathbb{K}_b$ which are kept secret from all other parties. As described in Section 5.6.1, T-Store uses a key space of $\mathbb{K}_b = \{0,1\}^{256}$ which corresponds to 2^{256} possible basic keys. The probability of guessing a valid basic key is therefore $\frac{8}{2^{256}} \approx 6.91 \cdot 10^{-77}$. As this probability is sufficiently small, T-Store can be considered as secure against brute force attacks **AC.1** for guessing basic keys.

5.7.4. Guessing Query Keys (AC.2)

A query key $qk \in \mathbb{K}_q$ is created from a basic hash function λ and encodes a particular query pattern. As described in Section 5.6.1, T-Store uses SHA-2 as the basic hash function with an output length of 256 bits. The set of all query keys $\mathbb{K}_q$ is therefore $\mathbb{K}_q = \{0,1\}^{256}$ which corresponds to 2^{256} possible query keys. A query key is used for retrieving all plaintext triples from the ciphertext graph G_C which satisfy the query pattern encoded in the key. Since each plaintext triple is encrypted for each of the eight

basic keys, encrypting a plaintext graph G with $n \in \mathbb{N}$ triples results in a ciphertext graph G_C with $8n$ ciphertexts. Each valid query key retrieves at least one and at most n plaintext triples from the graph G_C. Thus, the maximum number of possible query keys per graph is $8n$. This results in a probability of $\frac{8n}{2^{256}} = \frac{n}{2^{253}}$ for an attacker to guess a valid query key. For a plaintext graph with $n = 10^6$ triples, this corresponds to $\frac{8 \cdot 10^6}{2^{256}} \approx 6.91 \cdot 10^{-71}$ which is sufficiently small. In practice, this probability is usually smaller since a particular query key can decrypt multiple ciphertexts in a graph. This also results in less query keys for the graph. T-Store can therefore be considered as secure against brute force attacks **AC.2** for guessing query keys.

5.7.5. Extracting Basic Keys (AC.3)

As defined in Section 5.5.3, an authorization key $ak \in \mathbb{K}_a$ is created from a basic key $bk \in \mathbb{K}_b$ and a data owner's restriction pattern $r \in \mathbb{P}_q$ by applying the combining function ϱ as defined in Equation 5.19. In order to extract the basic key from an authorization key ak, an attacker must eliminate the restriction pattern from the key ak. Assume without loss of generality that an attacker has access to the authorization key $ak_{\texttt{ru?}}$. The key is created from the basic key $bk_{\texttt{++-}}$ and the restriction pattern $r = (s_r, \texttt{?}, \texttt{?})$ as follows:

$$ak_{\texttt{ru?}} = \varrho(bk_{\texttt{++-}}, \{w_{\texttt{s}} || s_r\}) \tag{5.31}$$

$$= \mathsf{bit}\left(\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_{\texttt{s}} || s_r))} \bmod N\right) \tag{5.32}$$

The function int transforms a bit string into an integer and bit transforms an integer into a bit string. The modulus $N \in \mathbb{N}$ is a product of two large primes $p, q \in \mathbb{N}$ which are only known to the data owner. Extracting the basic key $bk_{\texttt{++-}}$ from the authorization key $ak_{\texttt{ru?}}$ requires that the attacker is able to eliminate the exponent $\mathsf{int}(\lambda(w_{\texttt{s}} || s_r))$ in Equation 5.32.

The combining function ϱ is based on the RSA encryption algorithm [251] and shares the same security. The security of RSA is based on two mathematical problems which are assumed to be hard to solve. These problems are the factorization of large integers and the RSA problem [174]. An attacker who wants to eliminate the restriction pattern from an authorization key must solve one of these problems and thus has two different options. For the authorization key $ak_{\texttt{ru?}}$, the first option is to compute an $x \in \mathbb{N}$ which satisfies the following equation:

$$\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_{\texttt{s}} || s_r)) \cdot x} \bmod N = \mathsf{int}(bk_{\texttt{++-}}) \bmod N \tag{5.33}$$

This is equivalent to computing an $x \in \mathbb{N}$ which is inverse to $\mathsf{int}(\lambda(w_{\texttt{s}} || s_r)) \bmod (p-1) \cdot (q-1)$ [57]. Thus, the attacker must compute x so that it satisfies the following equation:

$$x \cdot \mathsf{int}(\lambda(w_{\texttt{s}} || s_r)) \equiv 1 \bmod (p-1) \cdot (q-1) \tag{5.34}$$

In order to solve this equation and to retrieve the value of x, the attacker must know the values of the two prime factors p and q. However, these values are only known to

the data owner. Furthermore, factorizing the modulus N is assumed to be hard to solve for large values of N [174]. The implementation of T-Store uses a value of N with a size of 2048 bit as recommended by NIST [217]. As an efficient way to factorize a modulus of this size has not been found yet, an attacker cannot compute the two primes p and q and apply Equation 5.34 in order to compute x. Consequently, the attacker can also not apply Equation 5.33 to extract the basic key bk from the authorization key ak.

The second option for the attacker to eliminate the restriction pattern from an authorization key is to compute a discrete root of an exponent. For the authorization key $ak_{\texttt{ru?}}$, the attacker must solve the following equation:

$$\sqrt[\mathsf{int}(\lambda(w_\mathtt{s}||s_r))]{\mathsf{int}(ak_{\texttt{ru?}})} \bmod N = \sqrt[\mathsf{int}(\lambda(w_\mathtt{s}||s_r))]{\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_\mathtt{s}||s_r))}} \bmod N \tag{5.35}$$

$$= \mathsf{int}(bk_{\texttt{++-}}) \bmod N \tag{5.36}$$

Solving this equation corresponds to solving the RSA problem which is also assumed to be hard to solve [174]. In summary, eliminating a restriction pattern r from an authorization key ak and extracting its basic key bk is hard as long as the factorization problem and the RSA problem are hard to solve as well. When using a value of N with a size of 2048 bit, T-Store can be considered as secure against attacks of type **AC.3**.

5.7.6. Computing Basic Keys (AC.4)

The combining function ϱ creates authorization keys $ak \in \mathbb{K}_a$ from basic keys $bk \in \mathbb{K}_b$ and restriction patterns $r \in \mathbb{P}$. As described in the previous section, the function ϱ is based on the RSA encryption algorithm [251]. RSA encrypts a plaintext message with a public exponent and a public modulus N. In the combining function ϱ, the plaintext message corresponds to the basic key bk and the public exponent corresponds to the bound parts of the restriction pattern r. The ciphertext resulting from an RSA encryption corresponds to the authorization key ak. The modulus N is the same as in RSA and used for all authorization keys of a particular graph.

Although RSA is considered to be secure as long as the factorization problem and the RSA problem are hard to solve, it is still vulnerable to the common modulus attack [271]. This attack exploits the relationship of two different RSA key pairs which share the same modulus N and encrypt the same plaintext message. It allows an attacker to retrieve the plaintext message from the two ciphertexts by using only the public keys which were used to create them. The common modulus attack can also be applied to the combining function ϱ as its design is similar to the RSA encryption algorithm. In order to conduct the attack on the combining function, an attacker must have access to two different authorization keys $ak_1, ak_2 \in \mathbb{K}_a$ which encode the same basic key bk. In addition, the attacker must know the corresponding restriction patterns $r_1, r_2 \in \mathbb{P}$ which are encoded into the keys ak_1 and ak_2, respectively, as well as the modulus N. Assume without loss of generality that the two authorization keys ak_1 and ak_2 encode the restriction

patterns $r_1 = (s_1, ?, ?)$ and $r_2 = (s_2, ?, ?)$, respectively, and the basic key bk_{+--}. The keys ak_1 and ak_2 are created with the combining function ϱ as follows:

$$ak_1 = \varrho(bk_{+--}, \{w_s || s_1\}) \tag{5.37}$$
$$= \mathsf{bit}\left(\mathsf{int}(bk_{+--})^{\mathsf{int}(\lambda(w_s||s_1))} \bmod N\right) \tag{5.38}$$
$$ak_2 = \varrho(bk_{+--}, \{w_s || s_2\}) \tag{5.39}$$
$$= \mathsf{bit}\left(\mathsf{int}(bk_{+--})^{\mathsf{int}(\lambda(w_s||s_2))} \bmod N\right) \tag{5.40}$$

The attacker uses the bound parts $s_1 \in \mathbb{R} \cup \mathbb{B}$ and $s_2 \in \mathbb{R} \cup \mathbb{B}$ of the restriction patterns r_1 and r_2 to compute the greatest common divisor of the two the exponents $\mathsf{int}(\lambda(w_s||s_1))$ and $\mathsf{int}(\lambda(w_s||s_2))$ as well as the coefficients of Bézout's identity [36]. This can be achieved by using the extended Euclidian algorithm [84]. I. e., the attacker applies this algorithm such that she receives the coefficients $c_1, c_2 \in \mathbb{Z}$ in the following equation:

$$c_1 \cdot \mathsf{int}(\lambda(w_s||s_1)) + c_2 \cdot \mathsf{int}(\lambda(w_s||s_2)) = \gcd\left(\mathsf{int}(\lambda(w_s||s_1)), \mathsf{int}(\lambda(w_s||s_2))\right) \tag{5.41}$$

Since the values $\mathsf{int}(\lambda(w_s||s_1))$ and $\mathsf{int}(\lambda(w_s||s_2))$ are created with a basic hash function λ, they are similar to pseudo-random values. Thus, they are very likely to be co-prime, i. e., it is $\gcd\left(\mathsf{int}(\lambda(w_s||s_1)), \mathsf{int}(\lambda(w_s||s_2))\right) = 1$ with high probability. The attacker can exploit this relationship between the two coefficients c_1 and c_2 and use them to compute the basic key bk_{+--} from the authorization keys ak_1 and ak_2 as follows:

$$\mathsf{int}(ak_1)^{c_1} \cdot \mathsf{int}(ak_2)^{c_2} \bmod N \tag{5.42}$$
$$= \left(\mathsf{int}(bk_{++-})^{\mathsf{int}(\lambda(w_s||s_1))} \bmod N\right)^{c_1} \cdot \left(\mathsf{int}(bk_{++-})^{\mathsf{int}(\lambda(w_s||s_2))} \bmod N\right)^{c_2} \bmod N \tag{5.43}$$
$$= \left(\mathsf{int}(bk_{++-})^{\mathsf{int}(\lambda(w_s||s_1))}\right)^{c_1} \cdot \left(\mathsf{int}(bk_{++-})^{\mathsf{int}(\lambda(w_s||s_2))}\right)^{c_2} \bmod N \tag{5.44}$$
$$= \mathsf{int}(bk_{++-})^{c_1 \cdot \mathsf{int}(\lambda(w_s||s_1))} \cdot \mathsf{int}(bk_{++-})^{c_2 \cdot \mathsf{int}(\lambda(w_s||s_2))} \bmod N \tag{5.45}$$
$$= \mathsf{int}(bk_{++-})^{c_1 \cdot \mathsf{int}(\lambda(w_s||s_1)) + c_2 \cdot \mathsf{int}(\lambda(w_s||s_2))} \bmod N \tag{5.46}$$
$$= \mathsf{int}(bk_{++-}) \bmod N \tag{5.47}$$

These computations utilize the laws of exponentiation and the properties of multiplication in modular arithmetic. The attacker can apply the resulting basic key bk_{+--} to create arbitrary authorization keys of type ++− and use these keys to decrypt corresponding ciphertext triples. Implementations of RSA usually prevent the common modulus attack by ensuring that all created RSA key pairs use different moduli N. This prevents an attacker from exploiting any relationship between different key pairs and from decrypting the plaintext message. In T-Store, however, using a different modulus N for each authorization key ak is not possible. T-Store requires that all plaintext triples which share the same bound parts for the same query type are encrypted with the same encryption key $ek \in \mathbb{K}_e$. This implies that the encryption key is created from the same bound parts and the same modulus N. Thus, T-Store is vulnerable to the attack **AC.4** which allows an attacker to compute the basic key from two authorization keys of the

same query type. In order to successfully prevent the attack, the combining function ϱ must be replaced by a secure function. Designing, implementing, and evaluating an alternative combining function is left for future work.[1]

5.7.7. Reducing Authorization Keys (AC.5)

In general, an authorization key $ak \in \mathbb{K}_a$ may be reduced to a different authorization key which shares the same basic key $bk \in \mathbb{K}_b$ but has a less restrictive restriction pattern $r \in \mathbb{P}_q$. Assume without loss of generality that an attacker has access to the authorization key $ak_{\texttt{rr?}}$ which encodes the basic key $bk_{\texttt{++-}}$ and the data owner's restriction pattern $r = (s_r, p_r, \texttt{?}) \in \mathbb{P}_q$. This restriction pattern already specifies all bound parts of a query pattern of type `++-`. Thus, an attacker can only define a user pattern $u \in \mathbb{P}_q$ as $u = (\texttt{?}, \texttt{?}, \texttt{?})$. In order to specify a bound subject or predicate herself, the attacker first must reduce the authorization key $ak_{\texttt{rr?}}$ to the authorization key $ak_{\texttt{ru?}}$ or to the key $ak_{\texttt{ur?}}$. In the following, the process of reducing the key $ak_{\texttt{rr?}}$ to the key $ak_{\texttt{ru?}}$ is described in more detail. A reduction to the authorization key $ak_{\texttt{ur?}}$ is similar and omitted for reasons of brevity. The key $ak_{\texttt{rr?}}$ is reduced to $ak_{\texttt{ru?}}$ as follows:

$$
\begin{aligned}
ak_{\texttt{rr?}} &= \varrho(bk_{\texttt{++-}}, \{w_{\texttt{s}}||s_r, w_{\texttt{p}}||p_r\}) && (5.48)\\
&= \mathsf{bit}\left(\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))\cdot\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))} \bmod N\right) && (5.49)\\
&= \mathsf{bit}\left(\left(\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))}\right)^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))} \bmod N\right) && (5.50)\\
&= \mathsf{bit}\left(\left(\mathsf{int}(bk_{\texttt{++-}})^{\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))} \bmod N\right)^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))} \bmod N\right) && (5.51)\\
&= \mathsf{bit}\left(\mathsf{int}(ak_{\texttt{ru?}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))} \bmod N\right) && (5.52)
\end{aligned}
$$

Again, these computations are based on the laws of exponentiation and on the the properties of multiplication in modular arithmetic. In order to extract the authorization key $ak_{\texttt{ru?}}$ from the key $ak_{\texttt{rr?}}$, the attacker must eliminate the exponent $\mathsf{int}(\lambda(w_{\texttt{s}}||s_r))$ from Equation 5.52. As described in Section 5.7.5, this would either require a factorization of the modulus N or the computation of a discrete root which are both hard to solve [174]. Thus, the authorization key $ak_{\texttt{rr?}}$ cannot be reduced to the less restrictive authorization keys $ak_{\texttt{ru?}}$ and $ak_{\texttt{ur?}}$ as long as these two problems are hard to solve. In this case, T-Store is able to prevent attacks of type **AC.5**.

5.7.8. Analyzing Ciphertext Frequency (AC.6)

Frequency-based attacks allow an attacker to infer the contents of ciphertext triples without decrypting them first. Conducting a frequency-based attack requires a deterministic

[1] This weakness of T-Store was discovered only recently with the help of a colleague of the author in February, 2016. Due to time constraints, it was not possible to apply and evaluate a new combining function before submitting this thesis.

encryption and knowledge about the distribution of potential plaintext triples. An attacker can acquire such knowledge by assessing the usage of widely-used properties and classes such as `rdf:type` or `foaf:Person` [149]. Deterministic encryption maps identical plaintexts to identical ciphertexts. In order to prohibit a frequency-based attack, it is necessary to encrypt identical plaintext triples as different ciphertext triples.

As described in Section 5.5.1, a single plaintext triple t is encrypted as eight different ciphertexts which correspond to each of the eight query pattern variants. Each ciphertext depends on a particular basic key $bk \in \mathbb{K}_b$ and on the complete plaintext triple. It encodes the triple's subject, predicate, and object as either bound or unbound. The bound parts are combined with the basic key into an encryption key ek and the unbound parts are encrypted with the key ek by using the encryption function ξ. The data owner chooses a different basic key for each of the eight query pattern variants. Thus, the eight ciphertexts created from a single plaintext triple are distinguished from each other by their unique combination of bound parts, unbound parts, and basic keys. A plaintext graph G is a set of plaintext triples $t \in G$ which contains each triple at most once. Therefore, the corresponding ciphertext graph G_C also contains each ciphertext triple $c \in G_C$ at most once. In summary, all ciphertexts stored in the ciphertext graph G_C are unique which prohibits frequency-based attacks **AC.6**.

5.7.9. Analyzing Ciphertext Size (AC.7)

Size-based attacks require that the size of a plaintext roughly corresponds to the size of its respective ciphertext. Here, size refers to the length of the bit string representation of plaintexts and ciphertexts. In order to prevent size-based attacks in T-Store, all ciphertexts of an encrypted graph G_C must have the same size. This prohibits an attacker from distinguishing between different ciphertext triples $c \in G_C$ and relating them to their respective plaintext triples $t \in G$ in the plaintext graph G.

In its current design, T-Store does not normalize the size of different ciphertexts and is therefore vulnerable to size-based attacks **AC.7**. However, these attacks can easily be eliminated by slightly modifying the encryption of the plaintext triples. In particular, all plaintext triples t must be transformed to the same size before encrypting them. This can be done similarly to splitting the sets id_S of ciphertext identifiers into arrays id_A as described in steps 6 and 7 of Section 5.5.2. A plaintext triple is first split into multiple fragments so that each fragment covers x bytes of data with $x \in \mathbb{N}$. Fragments which contain less than x bytes are padded accordingly. Each resulting fragment is associated with a unique ciphertext identifier $id_c \in \mathbb{N}$. Fragments of the same plaintext are related to each other by prefixing them with the ciphertext identifier of the next fragment. The prefixed fragments of the same plaintext are then encrypted separately with the same encryption key ek. The details of this modification are as follows: Let $y \in \mathbb{N}$ be the size of each ciphertext to be stored in the encrypted graph G_C. Each ciphertext encrypts 5 bytes of meta data and $x = y - 5$ bytes of content data which stores a fragment of the actual plaintext triple. The meta data describes how the content bytes are to be interpreted. The first byte of meta data states whether or not there are more ciphertexts

storing additional fragments of the same plaintext triple[2]. The value 1 states that this is the case and 0 indicates that this is not the case. The next 4 bytes of meta data store a 32 bit integer. The meaning of this integer is determined by the value of the first byte. If the first byte is 1, the integer contains the ciphertext identifier id_c of another ciphertext which stores the next fragment of the plaintext triple. In this case, all remaining x bytes of ciphertext data contain encrypted plaintext data. In contrast, if the first byte of meta data is 0, the integer defines the amount of actual content bytes. If these content bytes are less than $y - 5$, they are padded with random bytes.

For example, the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` has a length of 47 bytes. In order to encrypt this URI for the query pattern type `+-+` and the ciphertext size $y = 32$, it is first split into two different bit strings. The first bit string consists of $32 - 5 = 27$ bytes and the second bit string covers the remaining 20 bytes. Both bit strings are encrypted individually which results in two different ciphertexts. Let the first ciphertext be identified as 3 and the ciphertext identifier of the second ciphertext be 12. The ciphertexts c_3 and c_{12} are computed as follows with the encryption function ξ by using the encryption key $ek_{+-+} \in \mathbb{K}_e$ and $||$ as the concatenation operator:

$$c_3 := \xi(ek_{+-+}, \overbrace{\underbrace{1}_{1\text{ byte}} || \underbrace{12}_{4\text{ bytes}}}^{5\text{ bytes of meta data}} || \overbrace{\texttt{http://www.w3.org/1999/02/2}}^{x_3=27\text{ bytes of content}}) \tag{5.53}$$

$$c_{12} := \xi(ek_{+-+}, \underbrace{\underbrace{0}_{1\text{ byte}} || \underbrace{20}_{4\text{ bytes}}}_{5\text{ bytes of meta data}} || \underbrace{\texttt{2-rdf-syntax-ns\#type}}_{x_{12}=20\text{ bytes of content}} \underbrace{\texttt{.......}}_{\substack{7\text{ bytes of}\\\text{padding}}}) \tag{5.54}$$

Encrypting a single URI is necessary when encrypting a plaintext triple for the query pattern variant `+-+`. However, the described approach can also be used for encrypting plaintext triples for other query pattern variants. The presented modification of T-Store ensures that all ciphertexts stored in a ciphertext graph G_C are of equal size. This prevents an attacker from conducting a size-based attack **AC.7**.

5.7.10. Reasoning on Query Results (AC.8)

An authorized user has access to a limited set of authorization keys and can use these keys to apply queries to the ciphertext graph G_C. A corresponding query result consists different plaintext triples which contain A-box knowledge and/or T-box knowledge. A-box knowledge represents the factual data stored in the graph while T-box knowledge provides information about the ontology which was used for modeling the graph's triples [152]. More specifically, the T-box knowledge consists of URIs which identify properties and/or classes of the ontology. If the complete ontology of the plaintext graph is not already provided, a malicious user can guess the ontology based on these URIs. The user can then apply the ontology to reason about the A-box knowledge which

[2]Please note that a single bit would actually be sufficient for storing this information. However, T-Store is implemented in Java which does not allow to store individual bits.

she received with her query results as well. The reasoning process may provide additional triples from the plaintext graph which are not explicitly authorized by the data owner.

For example, an authorized user receives the triple `ex:Jay foaf:knows ex:Bob` after having applied a query to the ciphertext graph G_C. This triple contains both A-box knowledge and T-box knowledge. The A-box knowledge consists of the two URIs `ex:Jay` and `ex:Bob` and the T-box knowledge consists of the property `foaf:knows`. Based on this property, the user can infer that the FOAF ontology [50] was used for modeling at least one of the plaintext triples in the graph. The FOAF ontology defines several classes and properties for describing information about natural persons and their relationships. The user can apply these classes and properties to reason about the A-box knowledge of the query result. Using the property `foaf:knows` in a triple states that the subject and object of this triple are both of type `foaf:Person`. Furthermore, the FOAF ontology defines the class `foaf:Person` as being a subclass of `foaf:Agent`. By applying this T-box knowledge, the user can infer the four triples depicted in Listing 5.3 in addition to the single triple received from the actual query result.

```
1 ex:Jay rdf:type foaf:Person .    ex:Jay rdf:type foaf:Agent  .
2 ex:Bob rdf:type foaf:Person .    ex:Bob rdf:type foaf:Agent  .
```

Listing 5.3: Example triples resulting from reasoning.

T-Store is vulnerable to reasoning attacks **AC.8** as they are difficult to prevent in general. Preventing such attacks requires that the user does not have access to the used ontology and cannot apply it for reasoning. However, this would contradict with the principles of the Semantic Web [35] which is designed to support reasoning on machine-readable data. If the data owner does not want to prevent a user from reasoning on the query results, she must accept that the user can retrieve more information about the plaintext graph than the authorization keys allow. However, in many cases, the triples resulting from reasoning may not reveal any secret data about the plaintext graph and can therefore be considered as acceptable.

5.7.11. Analyzing the graph's characteristics (AC.9)

The characteristics of a graph cover several aspects such as the graph's size and connectivity as well as the number of all non-empty solution sequences, i. e., the number of query patterns which match at least one triple. Analyzing a graph's characteristics allows an attacker to distinguish between different ciphertext graphs and to relate a ciphertext graph to its corresponding plaintext graph. T-Store hides most characteristics with a graph's size being the only exception. As described in Section 5.5.1, encrypting a plaintext graph G with $n \in \mathbb{N}$ triples results in a ciphertext graph G_C with $8n$ ciphertexts. Thus, a ciphertext graph reveals the number of triples in its plaintext graph. Although the exact number can be concealed by inserting random bit strings as fake ciphertexts into the ciphertext graph and its index, the relative size of the plaintext graph is still revealed. This allows an attacker to distinguish between small graphs and large graphs.

As described in Section 5.7.8, T-Store ensures that each ciphertext in an encrypted graph G_C is unique. This prohibits an attacker from determining the contents of the plaintext triples $t \in G$ by relating their frequency to the frequency of the ciphertext triples $c \in G_C$. It also prevents an attacker from discovering the number of unique URIs, blank nodes, and literals in the plaintext graph G and from determining how often they occur at subject position, predicate position, or object position. This is necessary to determine the connectivity of the graph G which states how its triples are connected with each other. The connectivity can be used, e. g., for identifying the number of disjoint subgraphs or for distinguishing between loosely connected and strongly connected graphs. As an attacker cannot determine the frequency of individual triple parts, she cannot assess a graph's connectivity as well.

Neither a ciphertext graph G_C nor its corresponding index I reveal the number of query patterns which match at least one plaintext triple. The graph G_C is an unordered set of ciphertexts and does not provide any grouping of similar ciphertexts which match the same query pattern. Thus, the ciphertext graph does not reveal any information about the structure of the plaintext graph G. Although the index I provides a grouping of similar ciphertexts by using arrays of ciphertext identifiers as described in Section 5.5.2, these arrays are all of equal size and are scattered across the index. This prohibits an attacker from relating such arrays to each other which are associated with the same query pattern. Consequently, the attacker cannot determine the number of different ciphertexts matching a particular query pattern as well. In addition, the ciphertexts for all eight query pattern variants are stored in a single index. This prevents an attacker from determining the number of valid query patterns for each variant. In summary, T-Store only reveals the size of a plaintext graph but does hide any other structural information about it. Thus, the approach is only partially vulnerable to attacks **AC.9**.

5.8. Applications and Use Cases

This section describes how T-Store is used for implementing the scenarios introduced in Chapter 2. The first scenario focuses on the regulation of Internet communication as described in Section 2.1.3 and is covered in Sections 5.8.1 to 5.8.3. The second scenario is introduced in Section 2.2 and covers the secure management of medical data. Its implementation is addressed in Section 5.8.4.

5.8.1. Searching in Encrypted Log Files

The scenario for regulating Internet communication defines several computer networks which consist of communication end nodes such as client systems and intermediary nodes like application-level proxy servers. One of the computer networks introduced in Section 2.1.1 is the network of the German comprehensive school. It consists of several student computers which access the Internet via the school's proxy server. The proxy server acts as an enforcing system and implements the school's regulation policies which are further described in Section 3.4.4. In addition to this regulation, the server also records all Internet activities of the student computers in a log database. Each log entry

in the database represents one particular Internet communication. It contains the IP addresses and port numbers of a student computer and the contacted web server, the time and date of the communication, and the requested URL. The log entries stored by the proxy server are modeled with the Extended Log Format Ontology which is based on the Extended Log Format [138]. A detailed description of the ontology is provided in Appendix D. Thus, the log database corresponds to an RDF graph. Listing 5.4 shows an excerpt of the example log database depicted in Figure 2.3 which consists of two log entries. As depicted, each entry consists of seven different triples which share the same subject. The technical communication details are stored as the triples' objects. The first log entry is shown in lines 2 to 8 and represented as the blank node `_:entry47`. The entry states that the student computer with the IP address `192.168.2.103` and the port number `24154` requested the URL `http://www.porntube.com/` from the web server with the IP address `104.20.26.14` on Tuesday, October 21st, 2015, at 4:23:42 PM. The second entry is depicted in lines 10 to 16 and corresponds to the blank node `_:entry11`. The entry states that the same student computer requested the URL `https://www.tnaflix.com/` from the web server with the IP address `77.247.179.176` about six minutes later.

```
1  ...
2  _:entry47 log:date   "2015-10-21" .
3  _:entry47 log:time   "16:23:42" .
4  _:entry47 log:c-ip   "192.168.2.103" .
5  _:entry47 log:c-port "24154" .
6  _:entry47 log:s-ip   "104.20.26.14" .
7  _:entry47 log:s-port "80" .
8  _:entry47 log:cs-uri "http://www.porntube.com/" .
9
10 _:entry11 log:date   "2015-10-21" .
11 _:entry11 log:time   "16:29:12" .
12 _:entry11 log:c-ip   "192.168.2.103" .
13 _:entry11 log:c-port "3903" .
14 _:entry11 log:s-ip   "77.247.179.176" .
15 _:entry11 log:s-port "443" .
16 _:entry11 log:cs-uri "https://www.tnaflix.com/" .
17 ...
```

Listing 5.4: Excerpt from an example log database which is modeled with the Extended Log Format Ontology described in Appendix D.

The following sections describe how a supervisory school authority can securely query the log database depicted in Listing 5.4 using T-Store. The sections are based on the steps of storing and querying encrypted log entries as depicted in Figure 2.4 in Section 2.1.3.

5.8.2. Splitting Query Authorizations

In order to prohibit any unauthorized use of the log entries, the log database of the proxy server is encrypted with T-Store. The encryption is conducted by a hardware security module (HSM) [287] which is part of the proxy server. The HSM encrypts the triples of each log entry for each of the eight query pattern variants and stores the resulting ciphertexts triples in the log database $G_C \in \mathbb{G}_C$. To this end, the HSM

securely creates and stores eight different basic keys $bk \in \mathbb{K}_b$. Queries on the encrypted log database are collectively authorized by the school's administration (SA) and the parent's association (PA). Both parties only store a fragment of each basic key which cannot be directly used to create authorizations keys $ak \in \mathbb{K}_a$. Instead, the creation of a valid authorization key requires a separate authorization from both the school's administration and the parent's association. The fragments of basic keys stored by the school are defined as $sa = (sa_{---}, sa_{+--}, \ldots, sa_{+++}) \in \mathbb{J}$ and the fragments of the parent's association are $pa = (pa_{---}, pa_{+--}, \ldots, pa_{+++}) \in \mathbb{J}$ with $\mathbb{J}$ being the set of all tuples of key fragments. As a basic key bk is a bit string of length $d \in \mathbb{N}$, each fragment of a basic key is a bit string of length $\frac{d}{2} \in \mathbb{N}$. The set $\mathbb{J}$ is therefore defined as follows:

$$\mathbb{J} := \underbrace{\{0,1\}^{\frac{d}{2}} \times \{0,1\}^{\frac{d}{2}} \times \ldots \times \{0,1\}^{\frac{d}{2}}}_{8 \text{ times}} \tag{5.55}$$

The individual key fragments of the tuples sa and pa are chosen at random by the HSM when initializing the basic keys. Each basic key bk is then created by combining its respective fragments from the tuples sa and pa. The fragments are combined by multiplying their respective integer representations as follows:

$$bk_{---} = \mathsf{bit}\left(\mathsf{int}(sa_{---}) \cdot \mathsf{int}(pa_{---})\right) \tag{5.56}$$
$$bk_{+--} = \mathsf{bit}\left(\mathsf{int}(sa_{+--}) \cdot \mathsf{int}(pa_{+--})\right) \tag{5.57}$$
$$\ldots$$
$$bk_{-++} = \mathsf{bit}\left(\mathsf{int}(sa_{-++}) \cdot \mathsf{int}(pa_{-++})\right) \tag{5.58}$$
$$bk_{+++} = \mathsf{bit}\left(\mathsf{int}(sa_{+++}) \cdot \mathsf{int}(pa_{+++})\right) \tag{5.59}$$

Again, the function int maps bit strings to integers and bit transforms integers to bit strings. In order to apply a query to the encrypted log database, the supervisory school authority sends its query to the school's administration and the parent's association. Both parties collectively act as the data owner and compute a partial authorization key for each query pattern in the query. The partial authorization keys are based on a restriction pattern $r \in \mathbb{P}_q$ and on the basic key fragments stored by each party. The restriction patterns contain bound parts of each query pattern and the basic key fragments define the pattern variant. The computation of a partial authorization key is similar to the computation of a regular authorization key $ak \in \mathbb{K}_a$ as described in step 11 in Section 5.5.3. For example, in order to authorize a query pattern of type $\texttt{-++}$ with a restriction pattern $r = (\texttt{?}, p_r, o_r)$, the school's administration computes a partial authorization key $sa_{\texttt{?rr}}$ by using its basic key fragment $sa_{\texttt{-++}}$ as follows:

$$sa_{\texttt{?rr}} = \varrho(sa_{\texttt{-++}}, \{w_{\mathsf{p}}||p_r, w_{\mathsf{o}}||o_r\}) \tag{5.60}$$
$$= \mathsf{bit}\left(\mathsf{int}(sa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\mathsf{p}}||p_r)) \cdot \mathsf{int}(\lambda(w_{\mathsf{o}}||o_r))} \bmod N\right) \tag{5.61}$$

The function ϱ corresponds to the combining function defined in Equation 5.19 and N is a product of two large primes. Similarly, the parent's association computes its own partial authorization key $pa_{\texttt{?rr}}$ wit its basic key fragment $pa_{\texttt{-++}}$ as follows:

$$pa_{\texttt{?rr}} = \varrho(pa_{\texttt{-++}}, \{w_{\texttt{p}}||p_r, w_{\texttt{o}}||o_r\}) \tag{5.62}$$

$$= \mathsf{bit}\left(\mathsf{int}(pa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \bmod N\right) \tag{5.63}$$

As each of the two keys sa and pa only cover a part of a complete authorization key $ak_{\texttt{?rr}}$, they cannot be directly applied to the encrypted log database G_C. Both the school's administration and the parent's association send their partial authorization keys $sa_{\texttt{?rr}}$ and $pa_{\texttt{?rr}}$ to the supervisory school authority via a secure communication channel. The authority then computes the complete authorization key $ak_{\texttt{?rr}} \in \mathbb{K}_a$ by combining the two keys $sa_{\texttt{?rr}}$ and $pa_{\texttt{?rr}}$ as follows:

$$ak_{\texttt{?rr}} = \mathsf{bit}\left(\mathsf{int}(sa_{\texttt{?rr}}) \cdot \mathsf{int}(pa_{\texttt{?rr}}) \bmod N\right) \tag{5.64}$$

$$= \mathsf{bit}\left(\left(\mathsf{int}(sa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \bmod N\right) \cdot \left(\mathsf{int}(pa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \bmod N\right) \bmod N\right) \tag{5.65}$$

$$= \mathsf{bit}\left(\left(\mathsf{int}(sa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \cdot \mathsf{int}(pa_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))}\right) \bmod N\right) \tag{5.66}$$

$$= \mathsf{bit}\left((\mathsf{int}(sa_{\texttt{-++}}) \cdot \mathsf{int}(pa_{\texttt{-++}}))^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \bmod N\right) \tag{5.67}$$

$$= \mathsf{bit}\left(\mathsf{int}(bk_{\texttt{-++}})^{\mathsf{int}(\lambda(w_{\texttt{p}}||p_r))\cdot\mathsf{int}(\lambda(w_{\texttt{o}}||o_r))} \bmod N\right) \tag{5.68}$$

$$= \varrho(bk_{\texttt{-++}}, \{w_{\texttt{p}}||p_r, w_{\texttt{o}}||o_r\}) \tag{5.69}$$

These computations are based on the laws of exponentiation and on the properties of multiplication in modular arithmetic. The resulting authorization key $ak_{\texttt{?rr}}$ of Equation 5.69 is identical to an authorization key which is created from the basic key $bk_{\texttt{++-}}$ and the restriction pattern r as described in Table 5.8. The supervisory school authority uses the authorization key $ak_{\texttt{?rr}}$ to create a corresponding query key $qk_{\texttt{?rr}} \in \mathbb{K}_q$. It then applies this key to the encrypted log database G_C in the query phase as described in Section 5.5.4.

5.8.3. Analyzing the Log Database

According to §184 of the German Criminal Code [61], the comprehensive school must prohibit its students from accessing pornographic Internet content. The school implements this legal requirement by applying the regulation policy `alpp-1` which is expressed with the InFO policy language and further described in Section 3.4.4. The policy is technically enforced by the school's proxy server and contains two rules which prevent the school's students from accessing two web sites with pornographic content. The URLs of these web sites are `http://www.porntube.com/` and `http://www.fundorado.de/`.

Although the school's proxy server prohibits any access to these web sites, it still logs a student's attempts of accessing them. As depicted in the log database in Listing 5.4, the student computer with the IP address `192.168.2.103` tried to access the web site `http://www.porntube.com/`. This access request was blocked by the proxy server and stored in its log database. The student then tried to access a different web site with similar content several minutes later. The URL of this web site is `https://www.tnaflix.com/` which is not regulated according to the policy `alpp-1`.

The supervisory school authority suspects that the regulation policies of the school's proxy server might not cover all web sites providing pornographic content. Thus, it regularly analyzes the server's log database and searches for additional URLs to be included in the regulation as well. The authority assumes that students who unsuccessfully tried to access a blocked web site switched to an alternative web site with similar content. Based on this assumption, the authority searches the log database for all requests of blocked web sites. It then extracts the dates of the found log entries and searches for all URLs which were accessed on the same date, assuming that they may also lead to similar content. The query depicted in Listing 5.5 applies all these steps for the blocked web site `http://www.porntube.com/`. It consists of a single basic graph pattern with four different triple patterns and requires three join operations on the query variables `?logEntry1`, `?date`, and `?logEntry2`. A similar query must also be created for all other web sites which are blocked according to the proxy server's regulation policies.

```
1 SELECT ?URL
2 WHERE {
3   ?logEntry1 log:cs-uri "http://www.porntube.com/" .
4   ?logEntry1 log:date   ?date .
5   ?logEntry2 log:date   ?date .
6   ?logEntry2 log:cs-uri ?URL .
7 }
```

Listing 5.5: Example query of retrieving the URLs of all log entries which were created on the same date as the web site `http://www.porntube.com/` was requested.

As described in Section 5.5.5, the current design of T-Store only supports join operations on plaintext data. Each triple pattern of a query is applied to the ciphertext graph individually in order to create a plaintext solution sequence. The join operation is then conducted on all individual solution sequences. However, this process may reveal plaintext triples to the querying party that are not included in the final query result and are only part of an intermediary solution sequence. In order to protect the students' search interests and to prohibit any abuse of the logged Internet traffic, the supervisory school authority should only receive such plaintext triples that precisely match its query. Therefore, query processing is conducted differently than the process described in Section 5.5.5. In particular, queries with multiple triple patterns like the query depicted in Listing 5.5 are first split into different queries, each of which contains only a single triple pattern. The restriction patterns $r \in \mathbb{P}$ of these queries are chosen carefully so that their corresponding solution sequences are precisely as possible and do not provide any additional solution mappings which are irrelevant to the final query result. These queries

are then applied separately to the encrypted log database. Table 5.10 shows how the query of Listing 5.5 is split into four different queries. Each of these four queries must be authorized by the school's administration and the parent's association as described in Section 5.8.2. Although this increases the communication overhead between the two authorizing parties and the supervisory school authority, it also gives the authorizing parties more control over the queries that can be applied to the ciphertext graph.

Query	Description
Query 1	Retrieve all log entries containing a regulated URL
Query 2	Retrieve the date of all log entries found in query 1
Query 3	Retrieve all log entries for the dates found in query 2
Query 4	Retrieve the URL of all log entries found in query 3

Table 5.10.: Overview of the different queries for analyzing the encrypted log database.

The first query is depicted in Listing 5.6. Its triple pattern contains the URL of a blocked web site as a query parameter which is taken from the regulation policy `alpp-1`. The query retrieves the blank node identifiers of all log entries which contain a regulated URL. The query must be applied for all URLs which are blocked according to the proxy server's regulation policies. In the following, the overall query process is described for the URL `http://www.porntube.com/`. Querying other URLs is based on the same process. The first query requires an authorization key of type `-++` which encodes a restriction pattern $r_1 = (?, \texttt{log:cs-uri}, \texttt{http://www.porntube.com/})$. As this restriction pattern encodes all bound parts of the pattern variant `-++`, the supervisory school authority can only define a user pattern $u_1 = (?, ?, ?)$ for creating a query key qk_1. The authority applies this query key to the encrypted log database which results in a solution sequence containing all possible variable bindings of the query variable `?logEntry`. Applying the key qk_1 to the database shown in Listing 5.4 results in a solution sequence with a single variable binding which maps the variable `?logEntry` to the value `_:entry47`.

```
1 SELECT ?logEntry
2 WHERE { ?logEntry log:cs-uri "http://www.porntube.com/" . }
```

Listing 5.6: First example query of retrieving the blank node identifiers of all log entries which cover the requested URL `http://www.porntube.com/`.

The second query is depicted in Listing 5.4. It retrieves the dates of the log entries returned by the first query. The query uses the blank node identifier of the entries as a query parameter. A separate query must be created for each of these identifiers. In the example, the supervisory school authority only needs to create a single query for the log entry with the identifier `_:entry47`. Applying the query requires an authorization key of type `++-` which encodes a restriction pattern $r_2 = (?, \texttt{log:date}, ?)$. The authority combines this authorization key with a user pattern $u_2 = (\texttt{\_:entry47}, ?, ?)$ which defines the bound subject. Applying the resulting query key qk_2 to the log database of Listing 5.4

results in a solution sequence which contains a single variable binding for the query variable `?date`. The variable binding maps the variable to the value `2015-10-21`.

```
1 SELECT ?date
2 WHERE { _:entry47 log:date ?date . }
```

Listing 5.7: Second example query of retrieving the date of the log entry `_:entry47`.

The third query is depicted in Listing 5.8. It retrieves all log entries which were created on the dates returned by the second query. Again, a particular query must be created for each returned date. However, in the example, the solution sequence of the second query consists of a single variable binding only. Thus, the supervisory school authority must only create a single query which retrieves all log entries of the date `2015-10-21`. Creating a query key for this query requires an authorization key of type `-++` which encodes a restriction pattern $r_3 = (?, \texttt{log:date}, \texttt{2015-10-21})$. As the restriction pattern already contains all bound parts, the authority can only define a user pattern $u_3 = (?, ?, ?)$. Applying the resulting query key qk_3 to the example log database results in a solution sequence which contains two variable bindings for the query variable `?logEntry`. These variable bindings map the query variable to the values `_:entry47` and `_:entry11`.

```
1 SELECT ?logEntry
2 WHERE { ?logEntry log:date "2015-10-21" . }
```

Listing 5.8: Third example query of retrieving all log entries of the date `2015-10-21`.

The fourth query is depicted in Listing 5.9. It retrieves the URLs of all log entries which are returned by the third query. Again, the supervisory school authority must create a particular query for each returned log entry. In this case, however, the authority already knows the URL of the log entry `_:entry47` as this was the result of the first query. Therefore, it must only create a query for the log entry `_:entry11` which is depicted in Listing 5.9. The query requires an authorization key ak_4 of type `++-` which encodes a restriction pattern $r_4 = (?, \texttt{log:cs-uri}, ?)$. In order to complete the required bound parts, the authority defines a user pattern $u_4 = (\texttt{\_:entry11}, ?, ?)$ and combines it with the authorization key ak_4 to create a query key qk_4. Applying the query key qk_4 to the encrypted log database produces a solution sequence with a single variable binding that maps the variable `?URL` to the URL `https://www.tnaflix.com/`.

```
1 SELECT ?URL
2 WHERE { _:entry11 log:cs-uri ?URL . }
```

Listing 5.9: Fourth example query of retrieving the URL of the log entry `_:entry11`.

After having applied all four queries, the supervisory school authority further analyzes the URLs received from the fourth query. If a URL refers to pornographic content, the authority checks whether or not the URL is already blocked by the proxy server. If this is not the case, it suggests to add a new flow control rule to the server's regulation policies in order to regulate the URL as well.

The example queries demonstrate the general applicability of T-Store and its limitations. As the approach does not yet support join operations, each of the four queries is

applied separately to the encrypted log database. The restriction patterns of each query are chosen in such a way that they reduce the number of additional solution mappings which are returned by the query but are actually irrelevant for the final query result. However, even this querying process may still result in more solution mappings than the supervisory school authority requires for its analysis. In particular, the third query retrieves all log entries from the log database which were created on a specific date. These log entries may be created for different student computers and may also record the access of non-pornographic content which is irrelevant for the analysis of the supervisory school authority. In order to restrict the query's result, it should be redefined so that it only returns the log entries for a particular student computer on a particular date. Listing 5.10 shows a possible modification of the third query. The query only returns the log entries for the student computer with the IP address `192.168.2.103` on the date `2015-10-21` and conceals the log entries for all other computers on the same date. However, the query contains two different triple patterns whose solution sequences are joined on the query variable `?logEntry`. Thus, applying the query shown in Listing 5.10 on ciphertext graphs is not possible in the current version of T-Store without conducting a join operation on plaintext solution sequences. A further discussion of join operations and their implementation is provided in Section 5.9.4.

```
1 SELECT ?logEntry
2 WHERE {
3   ?logEntry log:c-ip "192.168.2.103" .
4   ?logEntry log:date "2015-10-21" .
5 }
```

Listing 5.10: Example query of retrieving all log entries of the date `2015-10-21` and the student computer with the IP address `192.168.2.103`.

5.8.4. Searching on Encrypted Medical Data

The second scenario covers the secure storage of medical data records and is introduced in Section 2.2. The scenario focuses on the patient Marla who manages her own personal health record (PHR) to monitor various medical information such as her weight, blood pressure, and pulse. Marla models her PHR as an RDF graph using an ontology which is based on the Health Level 7 reference information model (HL7 RIM). HL7[3] is a non-profit organization which develops several standards for managing electronic health data. HL7 RIM provides a generic language concept for such data which can be mapped to different serialization formats such as ontologies [163, 223, 228] or XML-based formats [94]. Marla encrypts her PHR with T-Store and stores the resulting ciphertext graph on a portable device. Whenever she consults a care delivery organization (CDO), she brings the device along with her. She then authorizes the CDO to apply queries to the encrypted PHR in order to support her medical treatment.

Marla acts as the data owner as she creates and encrypts all plaintext triples in her PHR herself. She also creates and securely stores the basic keys $bk \in \mathbb{K}_b$ required for

[3] `http://www.hl7.org`, last accessed: 01/21/16

creating all encryption keys $ek \in \mathbb{K}_e$. In order to flexibly define a CDOs access on her PHR, she replaces the basic keys after each year with new keys. Thus, Marla only uses a particular set of basic keys for encrypting all triples of a specific year. Table 5.11 shows the different basic keys which Marla has created in $n \in \mathbb{N}$ years. When Marla consults a CDO, she decides which triples of her PHR shall be accessible to the CDO. More specifically, she defines the queries the CDO can apply to the PHR and for which years the queries can be applied. She then creates a set of authorization keys $ak \in \mathbb{K}_a$ for each allowed query and year. Each authorization key is created from a restriction pattern $r \in \mathbb{P}$ and a basic key bk. In order to further restrict a CDO's access, Marla encodes all bound parts of an authorization key ak in its restriction pattern r. If Marla wants to allow the CDO to apply the same query pattern to her PHR for several years, she creates an authorization key ak for each year. The keys ak share the same restriction pattern r but differ in their basic key bk. For example, if she wants to authorize the query pattern $(\texttt{?}, r_p, r_o)$ for the last three years, she creates three different authorization keys ak of type `-++`. The keys ak are based on the basic keys $bk_{\texttt{-++},n-2}$, $bk_{\texttt{-++},n-1}$, and $bk_{\texttt{-++},n}$ with $n \in \mathbb{N}$ being the current year. After having created all authorization keys ak, Marla authorizes the CDO by handing over the keys together with the encrypted PHR stored on her portable device. The CDO then creates a query key $qk \in \mathbb{K}_q$ for each authorization key by combining it with a user pattern $u = (\texttt{?}, \texttt{?}, \texttt{?})$. It applies the query key qk to the encrypted PHR and uses the resulting plaintext triples to support Marla's medical treatment.

Key type	**Year** 1	**Year** 2	$\cdots$	**Year** $n-k$	$\cdots$	**Year** $n-1$	**Year** n
`---`	$bk_{\texttt{---},1}$	$bk_{\texttt{---},2}$	$\cdots$	$bk_{\texttt{---},n-k}$	$\cdots$	$bk_{\texttt{---},n-1}$	$bk_{\texttt{---},n}$
`+--`	$bk_{\texttt{+--},1}$	$bk_{\texttt{+--},2}$	$\cdots$	$bk_{\texttt{+--},n-k}$	$\cdots$	$bk_{\texttt{+--},n-1}$	$bk_{\texttt{+--},n}$
`-+-`	$bk_{\texttt{-+-},1}$	$bk_{\texttt{-+-},2}$	$\cdots$	$bk_{\texttt{-+-},n-k}$	$\cdots$	$bk_{\texttt{-+-},n-1}$	$bk_{\texttt{-+-},n}$
`--+`	$bk_{\texttt{--+},1}$	$bk_{\texttt{--+},2}$	$\cdots$	$bk_{\texttt{--+},n-k}$	$\cdots$	$bk_{\texttt{--+},n-1}$	$bk_{\texttt{--+},n}$
`++-`	$bk_{\texttt{++-},1}$	$bk_{\texttt{++-},2}$	$\cdots$	$bk_{\texttt{++-},n-k}$	$\cdots$	$bk_{\texttt{++-},n-1}$	$bk_{\texttt{++-},n}$
`+-+`	$bk_{\texttt{+-+},1}$	$bk_{\texttt{+-+},2}$	$\cdots$	$bk_{\texttt{+-+},n-k}$	$\cdots$	$bk_{\texttt{+-+},n-1}$	$bk_{\texttt{+-+},n}$
`-++`	$bk_{\texttt{-++},1}$	$bk_{\texttt{-++},2}$	$\cdots$	$bk_{\texttt{-++},n-k}$	$\cdots$	$bk_{\texttt{-++},n-1}$	$bk_{\texttt{-++},n}$
`+++`	$bk_{\texttt{+++},1}$	$bk_{\texttt{+++},2}$	$\cdots$	$bk_{\texttt{+++},n-k}$	$\cdots$	$bk_{\texttt{+++},n-1}$	$bk_{\texttt{+++},n}$

Table 5.11.: Basic keys bk created by Marla for n different years. All encryption keys ek created within a particular year are based on the basic keys of this year.

In its current implementation, T-Store uses a basic key size of 256 bits which corresponds to 32 bytes. A set of eight different basic keys requires $8 \cdot 32 = 256$ bytes of space. If Marla creates a different set of basic keys for each year, she must store $8n$ different basic keys which correspond to $256n$ bytes with $n \in \mathbb{N}$ being the number of years Marla manages her PHR. Even if Marla lives to be a hundred years old, she only has to store 800 basic keys for her entire life which require 204,800 bytes of space. This amount of space can be considered as negligible regarding a timespan of a hundred years.

5.9. Evaluation and Comparison with Existing Approaches

This section evaluates how the state of the art and related work presented in Section 5.1 as well as T-Store fulfill the requirements introduced in Section 5.2. In order to properly assess how the approaches for searching in encrypted relational databases and XML documents can be applied to RDF graphs, this section first defines a mapping from RDF graphs to XML documents and a mapping from RDF graphs to relational databases. The mappings are then used to assess the fulfillment of the different requirements. This section concludes with a discussion about join operations on encrypted data.

5.9.1. Encoding RDF Graphs

Listing 5.11 shows an example RDF graph which is serialized using Turtle [27]. The graph consists of three triples which are modeled with the FOAF ontology [50]. It defines the name and e-mail address of `ex:Tyler` and states that the person identified by this URI knows another person who is identified as `ex:Marla`. The graph can be used to process SPARQL queries such as the query depicted in Listing 5.12. The query retrieves the name of all persons with the e-mail address `tdurden@example.com`. It contains two triple patterns, each of which produces a different solution sequence. Both solution sequences are joined on the variable bindings of the query variable `?person`. Applying the query to the example graph results in a single solution mapping which binds the query variable `?name` to the value `Tyler Durden`.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2 @prefix ex:   <http://www.example.com/> .
3
4 ex:Tyler foaf:name  "Tyler Durden" .
5 ex:Tyler foaf:mbox  "tdurden@example.com" .
6 ex:Tyler foaf:knows ex:Marla .
```

Listing 5.11: Example RDF graph consisting of three triples. The graph is serialized using Turtle [27] and modeled with the FOAF ontology [50].

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name
3 WHERE {
4    ?person foaf:name ?name .
5    ?person foaf:mbox "tdurden@example.com" .
6 }
```

Listing 5.12: Example SPARQL query with two triple patterns. Applying the query to the example graph of Listing 5.11 results in a single solution mapping that maps the query variable `?name` to the value `Tyler Durden`.

RDF/XML [26] and OWL/XML [210] are two formats for encoding RDF graphs as XML documents. They are very flexible and support many different XML serializations of the same graph, e. g., by varying the sequence of the used XML elements [54, 10]. However, the formats are not designed for processing queries on the resulting XML documents with XML query languages like XPath [252]. As many approaches for searching

```xml
<graph>
  <triple>
    <subject>ex:Tyler</subject>
    <predicate>foaf:name</predicate>
    <object>Tyler Durden</object>
  </triple>

  <triple>
    <subject>ex:Tyler</subject>
    <predicate>foaf:mbox</predicate>
    <object>tdurden@example.com</object>
  </triple>

  <triple>
    <subject>ex:Tyler</subject>
    <predicate>foaf:knows</predicate>
    <object>ex:Marla</object>
  </triple>
</graph>
```

(a) Using child elements

```xml
<graph>
  <triple
    subject="ex:Tyler"
    predicate="foaf:name"
    object="Tyler Durden"
  />

  <triple
    subject="ex:Tyler"
    predicate="foaf:mbox"
    object="tdurden@example.com"
  />

  <triple
    subject="ex:Tyler"
    predicate="foaf:knows"
    object="ex:Marla"
  />
</graph>
```

(b) Using attributes

Listing 5.13: Two example XML encodings of the RDF graph depicted in Listing 5.11. Both XML encodings store each triple in the graph as a separate `triple` element. The first encoding uses child elements to store a triple's subject, predicate, and object whereas the second encoding uses attributes.

in encrypted XML documents support a subset of XPath queries, applying them to encrypted RDF graphs requires an XML encoding which can be used for query processing directly. Listing 5.13 shows two alternative XML encodings of the example graph of Listing 5.11 which are used in the following discussion. For reasons of brevity, all URIs are represented by their compact representation. Both encodings define a flat XML tree with `graph` as the root element and a separate `triple` element for each triple in the graph. The first encoding is depicted in Listing 5.13a and stores each part of a triple as a data value by using the three child elements `subject`, `predicate`, and `object`. In contrast, the second encoding depicted in Listing 5.13b uses XML attributes to store the triple's individual parts. The two encodings are semantically equivalent and can be easily converted into each other. They are both provided here as some approaches for searching in encrypted XML documents focus on processing data values while others are restricted to attribute values. Please note that the encodings are just two examples of how RDF graphs can be stored in XML files. They are only used in the discussion of this section and are not designed to be used in practice.

Both example encodings of Listing 5.13 can be used for processing XPath queries. Listing 5.14 shows an example XPath query which can be applied to the XML document of Listing 5.13a. The XPath query is equivalent to the SPARQL query of Listing 5.12. It encodes the two triple patterns of the original SPARQL query and contains all their bound parts. Each triple pattern is mapped to a path with starts at the root element

```
1 /graph/triple                                          Select all triples matching the first triple pattern.
2    [predicate/text() = "foaf:name"]                    Set the first pattern's predicate.
3    [subject/text() =                                   Join on the subjects of both patterns.
4       /graph/triple                                    Select all triples matching the second pattern.
5          [predicate/text() = "foaf:mbox"]              Set the second pattern's predicate.
6          [object/text() = "tdurden@example.com"]       Set the second pattern's object.
7          /subject/text()                               Return all variable bindings of the second pattern's subject.
8    ]
9    /object/text()                                      Return all variable bindings of the first pattern's object.
```

Listing 5.14: Example XPath query which corresponds to the SPARQL query of Listing 5.12. Applying the query to the XML document of Listing 5.13a results in one solution with the value `Tyler Durden`.

`graph` and ends at a particular part of a triple. The function `text()` retrieves the data value of an XML element such as `ex:Tyler`. It is used to compare the retrieved value with a bound part of the query and to conduct a join operation on the subjects of both triple patterns. The result of the XPath query is identical to the result of the SPARQL query of Listing 5.12. A similar XPath query can also be created for the second XML encoding of Listing 5.13b.

RDF graphs can be stored in relational databases by using, e. g., triple stores such as Sesame [54] and DB2RDF [45]. The two triple stores optimize query processing on an RDF graph by exploiting its internal structure. Sesame creates a different database table for each class of the graph's ontology and stores all instances of this class in the table. DB2RDF creates multiple tables and uses them to create a complex indexing system. However, utilizing the internal structure of an RDF graph may reveal the graph's characteristics. For example, creating a separate database table for each ontological class reveals how the instances are distributed across different classes. As described in Section 5.7.11, revealing a graph's characteristics may affect its confidentiality and must therefore be prevented. In order to conceal the characteristics of an RDF graph, the following discussion uses a single database table which contains all triples of the graph. Table 5.12 shows the database table `triples` which stores each triple of the example

Table 5.12.: Example database table storing the RDF graph of Listing 5.11. The table is named `triples` and stores each triple of the graph as a separate data record. It consists of three columns which store the triples' subjects, predicates, and objects.

Subject	Predicate	Object
ex:Tyler	foaf:name	"Tyler Durden"
ex:Tyler	foaf:mbox	"tdurden@example.com"
ex:Tyler	foaf:knows	ex:Marla

graph of Listing 5.11 as a separate data record. The table consists of three columns, each of which represents one part of a triple.

The database table of Table 5.12 can be used for querying the encoded RDF graph. Listing 5.15 shows an SQL query which is semantically equivalent to the SPARQL query of Listing 5.12. The `SELECT` clause corresponds to the query form and defines the query result, `FROM` sets the queried table, and the `WHERE` clause represents the query algebra and defines the matching conditions. Each triple pattern of the original SPARQL query is represented in the `FROM` clause with a different name. This allows it to distinguish between the different patterns and their respective parts. The `WHERE` clause contains all bound parts of the triple patterns and conducts a join operation on their subjects. Applying the SQL query to the database table shown in Table 5.12 provides the same result as applying the SPARQL query of Listing 5.12 to the RDF graph of Listing 5.11.

```
1 SELECT pattern1.object                                  Return all bindings of the first pattern's object.
2 FROM   triples pattern1, triples pattern2               Distinguish between the two triple patterns.
3 WHERE  pattern1.predicate = "foaf:name"                 Set the first pattern's predicate.
4 AND  pattern1.subject   = pattern2.subject              Join on the subjects of both patterns.
5 AND  pattern2.predicate = "foaf:mbox";                  Set the second pattern's predicate.
6 AND  pattern2.object    = "tdurden@example.com"         Set the second pattern's object.
```

Listing 5.15: Example SQL query which corresponds to the SPARQL query in Listing 5.12. Applying this query to the relational database table shown in Table 5.12 results in one solution with the value `Tyler Durden`.

In the following, the two XML encodings shown in Listing 5.13 and the relational database table depicted in Table 5.12 are used as a foundation to evaluate the state of the art and related work. The results of this evaluation are summarized in Table 5.13. The analysis is conducted with respect to the functional requirements **RC.F.1** to **RC.F.6** as well as the non-functional requirements **RC.N.1** to **RC.N.5.3**.

5.9.2. Evaluating the Functional Requirements

Many of the requirements defined in Section 5.2 are not specific to searching on encrypted RDF graphs and can be applied to any approach for searching in encrypted data. The only exceptions are the functional requirements **RC.F.1**, **RC.F.2**, and **RC.F.3** which are specific to SPARQL queries. Requirement **RC.F.1** covers the support of SPARQL triple patterns. A triple pattern can have none, one, two, or three bound parts which are the pattern's query parameters. In order to fulfill requirement **RC.F.1**, an approach must support queries with a varying number of query parameters as well. If an approach only allows a fixed number of query parameters, it does not completely fulfill requirement **RC.F.1**. The approaches for searching in encrypted relational databases can be applied to encrypted RDF graphs by storing the graph's triples in a single table as depicted in Table 5.12. Each triple corresponds to a database record with three different attributes. Approaches which use row-wise encryption map each plaintext triple to a single ciphertext whereas approaches with cell-wise encryption create three different ciphertexts for the triple's subject, predicate, and object. Although all approaches can process the resulting database table, they differ in their indexing, query processing,

Table 5.13.: Comparison of the capabilities of different approaches for searching in encrypted relational databases, encrypted XML documents, and encrypted graphs. The comparison is based on the requirements defined in Section 5.2. Rows correspond to different approaches and columns correspond to requirements. Requirements **RC.F.1** to **RC.F.6** are functional and **RC.N.1** to **RC.N.5.3** are non-functional. The letter y corresponds to a complete fulfillment of a requirement, l marks a partial fulfillment, n corresponds to no fulfillment, and – states that the requirement is not applicable.

	RC.F.1: SPARQL triple patterns	**RC.F.2**: SPARQL basic graph patterns	**RC.F.3**: SPARQL query forms	**RC.F.4**: Dynamic query authorizations	**RC.F.5**: Separating data owner and users	**RC.F.6**: Query templates	**RC.N.1**: Offline query process	**RC.N.2**: No trusted component	**RC.N.3**: Exact query result	**RC.N.4**: Data indistinguishability	**RC.N.5.1**: Hide string length of triple parts	**RC.N.5.2**: Hide density and connectivity	**RC.N.5.3**: Hide graph size
Relational databases													
CryptDB [235]	y	y	l	y	y	n	n	n	y	n	n	n	n
Elovici et al. [98]	l	n	l	y	n	–	n	y	l	y	n	y	n
Evdokimov and Günther [104]	l	n	l	y	n	–	n	y	y	y	n	l	n
Hacıgümüş et al. [137]	y	y	n	y	n	–	n	y	n	y	y	y	n
Prob-RPE [259]	y	y	l	y	y	n	n	n	y	n	y	n	y
Wang et al. [309]	y	n	l	y	n	–	n	y	n	y	n	y	n
Y. Yang et al. [315]	l	n	n	y	y	n	n	y	y	y	n	y	n
Z. Yang et al. [316]	l	n	l	y	n	–	n	y	y	y	y	y	n

Continued on next page.

and their support of SPARQL queries. However, all approaches support SPARQL triple patterns with at least one query parameter. CryptDB and Prob-RPE use a trusted proxy server which intercepts all plaintext queries from authorized users and maps them to corresponding ciphertext queries which are processed by a separate database server. The proxy server can process queries of different complexity and supports triple pat-

Table 5.14.: Comparing T-Store with the related work. *Continued from previous page.*

	RC.F.1: SPARQL triple patterns	**RC.F.2**: SPARQL basic graph patterns	**RC.F.3**: SPARQL query forms	**RC.F.4**: Dynamic query authorizations	**RC.F.5**: Separating data owner and users	**RC.F.6**: Query templates	**RC.N.1**: Offline query process	**RC.N.2**: No trusted component	**RC.N.3**: Exact query result	**RC.N.4**: Data indistinguishability	**RC.N.5.1**: Hide string length of triple parts	**RC.N.5.2**: Hide density and connectivity	**RC.N.5.3**: Hide graph size
XML documents													
Bouganim et al. [46]	y	n	l	y	y	n	y	n	y	y	n	y	n
Brinkman 1 [52]	n	n	n	y	n	–	n	y	y	–	–	y	n
Brinkman 2 [53]	l	n	l	y	n	–	n	y	y	y	n	y	n
Jammalamadaka and Mehrotra [165]	y	y	n	y	n	–	n	y	n	y	y	y	n
Lin and Candan [191]	n	n	n	y	n	–	n	y	n	–	–	y	n
OPESS [304]	y	n	l	y	n	–	n	y	y	y	n	y	n
SemCrypt [274]	y	n	l	y	n	–	n	y	y	y	n	y	n
Graph structures													
Chase and Kamara [78]	l	n	l	y	n	–	n	y	y	y	y	y	l
CryptGraph [314]	n	n	n	y	n	–	n	y	y	y	–	y	l
PPGQ [69]	n	y	l	y	n	–	n	y	n	y	y	y	l
T-Store	y	n	y	y	y	y	y	y	y	y	y	y	l

terns with multiple query parameters as well as basic graph patterns. Thus, the two approaches fulfill requirements **RC.F.1** and **RC.F.2**. Hacıgümüş et al. and Wang et al. index each part of a triple separately, resulting in three index values per triple. When initiating a query, a user maps each query parameter to its index value and sends it to the server. Both approaches support triple patterns with multiple query parameters and fulfill requirement **RC.F.1**. Elovici et al. encrypt each part of a triple separately and index the resulting ciphertexts with an index tree. A separate index tree is created for each triple part. Triple patterns are processed by traversing the index tree for a single query parameter. As the traversal can only be conducted on a single tree, triple patterns with multiple query parameters are not supported. Evdokimov and Günther

as well as Z. Yang et al. map each triple to three different ciphertext tuples. Triple patterns correspond to trapdoors [42] which encode a single query parameter and are processed at the server by comparing them with all ciphertext tuples. As the approaches do not support triple patterns with multiple query parameters, they only partially fulfill requirement **RC.F.1**. Y. Yang et al. encrypt a triple as a single ciphertext and index it with three different index values based on its subject, predicate, and object. A user creates a triple pattern by signing a single query parameter with her private key and sends it to the server. As the used signature scheme does not support signing multiple query parameters at once, the approach only partially fulfills requirement **RC.F.1**. The approaches for searching in encrypted XML documents support different types of queries which assess different parts of the document such as its element names, data values, and attribute values. In order to apply an approach to RDF graphs, it must support arbitrary URIs, blank nodes, and literals as query parameters. Such an approach can be used for searching in encrypted RDF graphs by using one of the encodings provided in Listing 5.13. In contrast, approaches which only allow element queries cannot be used for SPARQL queries. The number of possible element names in an XML document is restricted by the document's schema [124]. This prohibits element names from storing arbitrary URIs, blank nodes, or literals and makes it impossible to use them as query parameters. Brinkman 1 as well as Lin and Candan are restricted to element queries and cannot be used for SPARQL queries. Thus, they do not fulfill requirements **RC.F.1**, **RC.F.2**, and **RC.F.3**. Bouganim et al., Jammalamadaka and Mehrotra, OPESS, and SemCrypt support triple patterns with multiple query parameters and fulfill requirement **RC.F.1**. Bouganim et al. use a secure processing unit at the user side which processes all queries on the encrypted XML document. This unit is also capable of processing triple patterns with multiple query parameters and can operate on both encodings of Listing 5.13. Each plaintext triple is mapped to three different ciphertexts which correspond to the triple's subject, predicate, and object. Jammalamadaka and Mehrotra are essentially the same as Hacıgümüş et al. and fulfill the same requirements. The approach requires the XML encoding of Listing 5.13b. It encrypts each `triple` element as single ciphertext and associates it with three index values for the triple's subject, predicate, and object. Query processing is identical to Hacıgümüş et al. OPESS supports path queries based on element names and data values. As the approach does not operate on attribute values, it requires the XML encoding of Listing 5.13a. Each triple in the resulting XML document is encrypted as three different ciphertexts and indexed based on its individual parts. A triple pattern corresponds to a path and is processed by encrypting its query parameters and sending it to the server. SemCrypt supports path queries based on element names, attribute values, and data values and can be used with both encodings of Listing 5.13. Again, each plaintext triple is mapped to three different ciphertexts. SemCrypt processes queries by exchanging messages between the user and the server. Although each message only covers a single query parameter, multiple parameters are possible by exchanging several messages which are further processed by the user. Brinkman 2 uses trapdoors and is similar to Z. Yang et al. Thus, the approach only partially fulfills requirement **RC.F.1**. As supported queries may contain attribute values and data values, the approach supports both encodings of Listing 5.13. The graph-based

approaches Chase and Kamara, CryptGraph, and PPGQ focus on abstract graphs and are not specifically designed to support RDF graphs. Chase and Kamara support queries which retrieve all edges between two nodes. These queries can be used for triple patterns of type `+-+` which retrieve all predicates between a subject and and object. Although the approach defines a flexible index structure which can generally be used to support other types of triple patterns as well, the authors themselves only provide a small set of query types. Thus, the approach only partially fulfills requirement **RC.F.1**. Although CryptGraph operates on labeled graphs, it does not support labeled edges. However, RDF graphs have labeled nodes and labeled edges, which correspond to the triples' predicates. Thus, CryptGraph cannot be used for searching in encrypted RDF graphs and does not fulfill requirements **RC.F.1**, **RC.F.2**, and **RC.F.3**. PPGQ is restricted to subgraph queries and cannot be used for applying a single triple pattern to an RDF graph. Thus, the approach does not fulfill requirement **RC.F.1**. T-Store natively supports all eight types of triple patterns by encrypting each triple in the plaintext graph separately for each of the pattern types.

Requirement **RC.F.2** covers SPARQL basic graph patterns which consist of several triple patterns. If the triple patterns share identical query variables, their solution sequences are combined via a join operation on the variables. Thus, an approach which fulfills requirement **RC.F.2** must support join operations as well. Conducting a join operation requires the identification of compatible solution sequences, i. e., solution sequences which share identical variable bindings. Hacıgümüş et al. divide the values of all three triple parts into disjoint buckets which are used for processing queries and join operations. When processing a basic graph pattern, the server first computes the solution sequence of each triple pattern. It then combines all solution sequences by conducting a join operation on all compatible variable bindings. Variable bindings are compatible with each other if they map the same query variable to the same bucket identifier. CryptDB uses deterministic encryption to support join operations. Deterministic encryption maps identical plaintext values to identical ciphertext values. A join operation can then be directly applied to the ciphertext values. Although Prob-RPE uses probabilistic encryption, it associates each ciphertext with an additional join identifier. Join identifiers are bit strings which are identical for equal plaintext values. They allow a server to detect compatible ciphertexts and to conducts join operations on them. None of the other approaches for searching in relational database support join operations. Instead, they require the user to split a basic graph pattern into multiple SQL queries, each of which consists of a single triple pattern. The server processes each triple pattern individually and returns a corresponding solution sequence. Subsequently, the user decrypts all received solution sequences and conducts a join operation on the resulting plaintext. Jammalamadaka and Mehrotra are similar to Hacıgümüş et al. and support join operations using bucket identifiers. In contrast, none of the other approaches which operate on XML documents can detect compatible variable bindings and do not fulfill requirement **RC.F.2**. Chase and Kamara are restricted to queries which consist of a single triple pattern only. Join operations cannot be conducted on the ciphertext graph and must be conducted by the user via combining several solution sequences. Thus, the approach does not fulfill requirement **RC.F.2**. PPGQ supports subgraph queries

which can be used for implementing basic graph patterns on RDF graphs. A basic graph pattern is encoded as a single subgraph query and does not require any join operation. PPGQ therefore fulfills requirement **RC.F.2**. Similar to Chase and Kamara, the current design of T-Store does not support join operations and requires the user to conduct such operations on all received solution sequences. A detailed discussion on join operations is provided in Section 5.9.4.

Requirement **RC.F.3** states that `SELECT`, `CONSTRUCT`, and `ASK` queries must be equally supported. As described in Section 5.1.4, `SELECT` queries and `CONSTRUCT` queries operate on variable bindings whereas `ASK` queries return a boolean value. In order to fulfill requirement **RC.F.3**, an approach must support two different types of queries. Queries of the first type are used in `SELECT` queries and `CONSTRUCT` queries and process the three parts of all matching ciphertext triples individually in order to create the necessary variable bindings. An approach which cannot process an encrypted triple's subject, predicate, or object separately and only operates on complete ciphertext triples neither supports `SELECT` queries nor `CONSTRUCT` queries. Queries of the second type support `ASK` queries and match a set of query conditions against the queried data. The resulting boolean value indicates whether or not the query conditions have at least one match in the encrypted data. The value does not reveal any additional information about the plaintext data [140]. An approach which does not provide boolean query results or reveals more information about the plaintext data does not support `ASK` queries. None of the approaches which operate on relational databases support `ASK` queries as they are not able to return boolean query results. However, most of them use cell-wise encryption and encrypt each part of a triple individually. This can be used for returning individual variable bindings as query results which is necessary to support `SELECT` queries and `CONSTRUCT` queries. Thus, approaches using cell-wise encryption partially fulfill requirement **RC.F.3**. In contrast, Hacıgümüş et al. and Y. Yang et al. apply row-wise encryption and always return complete ciphertext triples. As they cannot create individual variable bindings as query results, they do not fulfill requirement **RC.F.3**. Many of the approaches which operate on XML documents can retrieve individual triple parts as query results and thus support `SELECT` queries and `CONSTRUCT` queries. However, none of them can be used for `ASK` queries. Bouganim et al. and Brinkman 2 encrypt each part of a triple separately and support individual variable bindings. Similarly, OPESS and SemCrypt map each plaintext triple to three different XML paths which correspond to the triple's subject, predicate, and object. The different paths can be processed independently and support individual variable bindings. PPGQ is restricted to subgraph queries which can be used for implementing `ASK` queries on encrypted RDF graphs. As the approach does not provide any mechanism for implementing variable bindings, it only partially fulfills requirement **RC.F.3**. In contrast, Chase and Kamara are not designed for subgraph queries and are restricted to `SELECT` queries and `CONSTRUCT` queries which contain triple patterns of type `+−+`. Thus, they partially fulfill requirement **RC.F.3** as well. In its current design, T-Store is only able to retrieve individual variable bindings and natively supports `SELECT` queries and `CONSTRUCT` queries. Although Algorithm 5.4 describes a method for processing `ASK` queries, it requires the computation of all variable bindings to determine the boolean query results. This contradicts with the specification

of `ASK` queries since they do not reveal any additional information about the queried graph. However, T-Store can easily be extended to support `ASK` queries as well. The extension requires eight additional basic keys for each of the eight triple pattern variants. These basic keys are independent from the basic keys that are used for `SELECT` queries and `CONSTRUCT` queries. They are used for creating encryption keys as defined in Table 5.1. Each created encryption key encrypts only a single value which indicates that the query pattern encoded in the key has at least one match in the ciphertext graph. The value does not provide any additional information about the graph and thus corresponds to the boolean query result of an `ASK` query. The ciphertexts resulting from such an encryption can be processed in the same way as the ciphertexts of `SELECT` queries and `CONSTRUCT` queries. They can also be stored in the same ciphertext graph and be indexed in its corresponding index. With this extension, T-Store is able to support `ASK` queries as well and fulfills requirement **RC.F.3**.

None of the discussed approaches is limited to a predefined set of queries. Instead, each approach allows the queries to be formulated after the plaintext graph has been encrypted. Thus, all approaches including T-Store fulfill requirement **RC.F.4**. Requirement **RC.F.5** demands that an approach must distinguish between a data owner who encrypts the plaintext graph and users who apply queries to the ciphertext graph. The data owner creates and permanently stores all encryption keys and a user can only access those keys which are explicitly authorized by the data owner. This requirement implies that an approach is not restricted to a single party which stores all cryptographic keys and can exclusively access the plaintext triples. If an approach even supports multiple users with different query authorizations, it implicitly fulfills requirement **RC.F.5**. Supporting multiple users necessitates a distinction between the users and the data owner who authorizes them. Most of the discussed approaches focus on data outsourcing [281] and consider the data owner to be identical to a single user. As this user is completely trusted and stores all cryptographic keys, the approaches do not fulfill requirement **RC.F.5**. The only exceptions are CryptDB, Prob-RPE, Y. Yang et al., and Bouganim et al. which support multiple users. CryptDB and Prob-RPE authorize each user by a trusted proxy server and Y. Yang et al. authenticate a user with a public key pair. Bouganim et al. require a trusted computing device at each user's side which authorizes and conducts all queries. T-Store distinguishes between the data owner and users and fulfills requirement **RC.F.5**. A data owner creates and permanently stores all basic keys which are used for creating particular query authorizations.

Requirement **RC.F.6** covers query templates which represent groups of similar query authorizations. Instead of authorizing each query individually, a query template combines similar queries into a single authorization. As query templates imply a distinction between data owners and users, the fulfillment of requirement **RC.F.6** necessitates the fulfillment of requirement **RC.F.5** as well. All approaches which support multiple users require an explicit authorization for each query. As they do not support query templates which combine similar queries, they do not fulfill requirement **RC.F.6**. In contrast, T-Store provides query templates via authorization keys. An authorization key is an incomplete query key which can be transformed into different query keys by an authorized user. Thus, T-Store fulfills requirement **RC.F.6**.

5.9.3. Evaluating the Non-Functional Requirements

Many of the discussed approaches require a server to conduct most of the query processing and do not fulfill requirement **RC.N.1**. The only exception is Bouganim et al. which use a secure processing unit for each user. The unit processes all queries without involving any online system. T-Store also allows a user to process all queries offline and fulfills requirement **RC.N.1** as well. Requirement **RC.N.2** states that an approach must not involve a trusted system when applying queries to ciphertext graphs. A system is considered to be trusted if it has access to plaintext triples but is not their final recipient. Examples of trusted systems are proxy servers which apply queries on behalf of authorized users. Most approaches only involve an authorized user and an untrusted server when processing a query. The server solely operates on ciphertext triples and cannot access any plaintext triples. The user is not a trusted system as she is the final recipient of the query results. Thus, these approaches fulfill requirement **RC.N.2**. In contrast, CryptDB, Prob-RPE, and Bouganim et al. require a trusted system for processing queries. CryptDB and Prob-RPE use a proxy server which acts as an intermediary between an authorized user and the database server and conducts the actual query processing. It is also able to access all plaintext triples returned by the database server. Bouganim et al. use a secure processing unit at the user's side. Similar to a proxy server, the unit intercepts all queries from the user, applies them to the ciphertext graph, and returns the resulting plaintext triples. As the user neither controls the proxy server nor the secure processing unit, she must trust that these systems do not abuse any received plaintext triples. Thus, CryptDB, Prob-RPE, and Bouganim et al. do not fulfill requirement **RC.N.2**. In contrast, T-Store does not use a trusted system and only involves the user who applies queries directly to a ciphertext graph.

An approach fulfills requirement **RC.N.3** if processing a query only reveals the matching triples while all other triples remain inaccessible to the querying user. CryptDB, Prob-RPE, and Bouganim et al. use a trusted system for query processing which conducts any necessary refining steps itself and returns the final solution sequence to the user. Thus, the three approaches fulfill requirement **RC.N.3**. Elovici et al. process a triple pattern by traversing an index tree. During the traversal, the user receives several tree nodes which point to ciphertext triples stored at the server. After having identified all matching triples, the user retrieves them in a second step. Although an honest user will only request matching ciphertext triples, the approach does not prevent a dishonest user from retrieving other triples as well. Thus, the approach only partially fulfills requirement **RC.N.3**. Evdokimov and Günther, Z. Yang et al., and Brinkman 2 encode triple patterns as trapdoors. Due to their design, trapdoors can only be applied to ciphertext triples which match the encoded triple pattern. Otherwise, the decryption fails and the corresponding plaintext triple remains inaccessible. Consequently, the three approaches return an exact solution sequence. Hacıgümüş et al. as well as Jammalamadaka and Mehrotra index ciphertext triples with the bucket identifiers of their individual parts. A user initiates a query by mapping its parameters to corresponding bucket identifiers and requesting all matching ciphertext triples from the server. As different plaintext values are mapped to the same bucket identifiers, the server may return

false positives which require an additional refining step. Wang et al. index each part of a ciphertext triple with a hash value using a non-collision-resistant hash function. Again, query processing requires a refining step which contradicts with requirement **RC.N.3**. Y. Yang et al. associate each ciphertext triple with three index values based on its subject, predicate, and object. As this allows the server to return exact query results, the approach fulfills requirement **RC.N.3**. Brinkman 1 applies a subtree query to an XML document by computing several polynomials. The user compares each polynomial with a predefined value. A query result is found if both values are equal. Otherwise, nothing is revealed about the plaintext data. Thus, the approach fulfills requirement **RC.N.3**. Lin and Candan apply path queries to XML documents by traversing the document tree. In order to conceal her actual query, the user retrieves multiple ciphertexts in each traversal step. She decrypts all ciphertexts, re-encrypts them with a new encryption key, and sends them to the server. As the user receives more plaintext data than necessary for answering her query, the approach does not fulfill requirement **RC.N.3**. OPESS generally supports path queries which retrieve complete subtrees of an XML document. As a received subtree forms a superset of the actual query result, an additional refining step is necessary. However, applying OPESS to an RDF graph is based on the encoding of Listing 5.13a. Each triple is then encrypted individually which supports exact query results. SemCrypt returns exact results for queries with a single query parameter. Multiple query parameters require a separate solution sequence for each parameter which must be combined afterwards. As this may remove incompatible solution mappings, SemCrypt only partially fulfills requirement **RC.N.3**. Chase and Kamara essentially map all possible combinations of query parameters to a corresponding solution sequence and return exact query results. CryptGraph processes subgraph queries by using homomorphic encryption [111] and does not reveal any intermediate query results. Instead, it only returns an encrypted value which states whether or not the subgraph is part of the queried graph. PPGQ supports subgraph queries in encrypted graph collections and indexes each graph with a feature vector describing its subgraphs. To reduce the size of the feature vector, similar subgraphs are mapped to the same feature. As this may result in false positives, the user must conduct an additional refining step. In contrast, T-Store returns exact query results since a query key can only decrypt those ciphertext triples which match the query pattern encoded in the key.

Requirement **RC.N.4** states that it must be computationally hard to detect identical URIs, blank nodes, or literals in the ciphertext graph. This implies that neither the ciphertext graph nor its index can be used for relating identical plaintext values to each other. CryptDB and Prob-RPE support join operations by identifying compatible ciphertext triples. CryptDB achieves this with deterministic encryption and Prob-RPE uses join identifiers. Both solutions allow it to detect identical plaintext values which directly contradicts with requirement **RC.N.4**. Elovici et al., Evdokimov and Günther, and Z. Yang et al. use probabilistic encryption and map each plaintext value to a different ciphertext. The resulting ciphertext triples prohibit the detection of identical plaintext URIs, blank nodes, or literals. Elovici et al. use an index which is also probabilistically encrypted and hides the frequency of the indexed values. In contrast, Evdokimov and Günther as well as Z. Yang et al. do not use an index at all. Thus, all three approaches

fulfill requirement **RC.N.4**. Hacıgümüş et al. and Y. Yang et al. use row-wise encryption which combines all three parts of a plaintext triple into a single ciphertext and conceals the distribution of individual plaintext values. In addition, Hacıgümüş et al. index each ciphertext triple with bucketization which maps different plaintext values to the same bucket identifier. Again, this hides the frequency of individual triple parts. Y. Yang et al. index a ciphertext triple with three index values created from the triple's subject, predicate, and object. If different triples share the same parts, a counter is added to the index value to ensure that all values are unique. Wang et al. apply cell-wise encryption and index each ciphertext using a hash function which is not collision-resistant. As the encryption can be conducted probabilistically, neither the ciphertexts nor the index values reveal the exact distribution of the plaintext values. Most of the approaches which operate on XML documents fulfill requirement **RC.N.4**. The only exceptions are Brinkman 1 as well as Lin and Candan which are restricted to element queries. As they do not operate on individual URIs, blank nodes, or literals, requirement **RC.N.4** cannot be applied to them. Bouganim et al. and Brinkman 2 apply probabilistic encryption and use an index which only operates on structural information. Thus, the index does not reveal any similarities between the ciphertext triples. OPESS probabilistically encrypts each XML element and transforms its index in such a way that the frequency of all indexed values is almost identical. Thus, the index cannot be used for determining the exact distribution of identical plaintext values. SemCrypt maps each path in an XML document to a unique identifier and stores this mapping at the user's side. The index is probabilistically encrypted and conceals identical plaintext URIs, blank nodes, or literals. All approaches for searching in encrypted graphs fulfill requirement **RC.N.4**. Chase and Kamara support triple patterns of type `+−+` by mapping tuples of subjects and objects to an encrypted list of predicates. Although the approach reveals the number of different subjects and objects in the graph, it does not reveal how often they occur in different plaintext triples. As this information is only provided by the encrypted predicates, the approach hides the frequency of individual URIs, blank nodes, and literals. CryptGraph represents a graph by its encrypted adjacency matrix. Similar to Chase and Kamara, CryptGraph reveals the number of different subjects and objects in the plaintext graph but hides all information about how they are connected. Thus, it neither reveals the number of different predicates between a subject and an object nor how often a particular value occurs at subject or object position. PPGQ encrypts a graph as a single ciphertext and thereby hides the individual URIs, blank nodes, and literals in the graph. As described in Section 5.7.8, T-Store encrypts a plaintext graph in such a way that all resulting ciphertexts are unique. This prevents the identification of identical plaintext URIs, blank nodes, or literals in the ciphertext triples.

Requirement **RC.N.5.1** demands that an approach hides the string length of individual URIs, blank nodes, and literals in the graph. Hacıgümüş et al. use row-wise encryption which hides the individual string length of each triple part. Prob-RPE supports different encryption schemes and allows it to transform all plaintext values to the same length before encrypting them. Z. Yang et al. apply cell-wise encryption after having padded each part of a triple to the same length. The resulting ciphertexts all share the same string length and do not reveal the length of the plaintext values.

Y. Yang et al. use row-wise encryption but associate each ciphertext triple with three index values. Different index values are created for the same plaintext value by using an additional counter. As this counter only slightly modifies the string length of the index value, the approach partially fulfill requirement **RC.N.5.1**. All other approaches which operate on relational databases use cell-wise encryption without any padding and map plaintext triples to ciphertext triples with roughly the same string length. Apart from Jammalamadaka and Mehrotra, none of the approaches for searching in encrypted XML documents fulfill requirement **RC.N.5.1**. Similar to Hacıgümüş et al. Jammalamadaka and Mehrotra encrypt each plaintext triple as a single ciphertext. Bouganim et al., OPESS, and SemCrypt encrypt each path in the XML document individually without using any padding. As a path also contains the data value of an element, the string length of a ciphertext reveals the size of its plaintext. Brinkman 2 encrypts each element together with its data value. Again, the approach does not use any padding and reveals the size of the encrypted plaintext values. As Brinkman 1 and Lin and Candan are restricted to element queries, requirement **RC.N.5.1** cannot be applied to them. Chase and Kamara use padding to transform all graph labels to the same size before encrypting them and fulfill requirement **RC.N.5.1**. As CryptGraph does not support labeled graphs, it cannot be evaluated for this requirement. PPGQ encrypts a graph as a single ciphertext and thereby hides the string length of all URIs, blank nodes, and literals in the graph. T-Store fulfills requirement **RC.N.5.1** by using the extension described in Section 5.7.9.

Requirement **RC.N.5.2** states that an approach must conceal the density and connectivity of a graph. The two characteristics can be used to distinguish between different ciphertext graphs or to relate a ciphertext graph to its corresponding plaintext graph. CryptDB uses deterministic encryption which allows it to analyze the frequency of all ciphertexts and to compute the density and connectivity of the plaintext graph. Similarly, Prob-RPE uses join identifiers which also allow such an analysis. Thus, the two approaches do not fulfill requirement **RC.N.5.2**. In contrast, all other approaches which operate on relational databases fulfill this requirement. Elovici et al. use probabilistic encryption for the plaintext triples and their index values. Thus, the resulting ciphertexts cannot be related to each other. Evdokimov and Günther as well as Z. Yang et al. also use probabilistic encryption and do not use any index which can be used for analyzing the ciphertext graph. The row-wise encryption applied by Hacıgümüş et al. prohibits an analysis of a graph's density and connectivity by using its ciphertext graph. Although the approach uses bucketization which associates different ciphertext triples with each other, the buckets do not provide precise information about a graph's internal structure. As different plaintext values are mapped to the same bucket, it is impossible to distinguish between ciphertexts triples which share the same plaintext values and ciphertext triples with different plaintext values. Wang et al. apply probabilistic encryption and use an index with colliding index values which does not reveal any information about the graph's characteristics. Y. Yang et al. uses row-wise encryption and create index values using a counter which is unknown to the database server. Thus, the counter cannot be used for analyzing the graph's internal structure. All approaches for searching in encrypted XML documents hide a graph's density and connectivity and fulfill require-

ment **RC.N.5.2**. Bouganim et al. and Brinkman 2 use probabilistic encryption and apply a structural index which does not provide any information about a graph's plaintext values. Brinkman 1 as well as Lin and Candan only support path queries based on element names and do not process URIs, blank nodes, or literals, which is necessary to analyze a graph's density and connectivity. Jammalamadaka and Mehrotra are similar to Hacıgümüş et al. OPESS uses probabilistic encryption and normalizes its index in such a way that it does not reveal the distribution of the encrypted URIs, blank nodes, or literals. SemCrypt uses probabilistic encryption for the plaintext triples and the index. All approaches for searching in encrypted graphs fulfill requirement **RC.N.5.2** as they do not provide complete information about a graph's structure. Chase and Kamara map pairs of subjects and objects to lists of all connecting predicates. These lists are padded to the same length before encrypting them. Predicates which occur in many triples are therefore indistinguishable from predicates which are only rarely used. CryptGraph represent a graph by its encrypted adjacency matrix which hides all connections between two nodes in the graph and prevents any analysis of the graph's structure. PPGQ encrypts a graph as a single ciphertext and thereby hides its internal structure. The feature vector used for indexing a graph maps different subgraphs to the same feature and does not reveal the density and connectivity of the graph. As described in Section 5.7.11, T-Store also hides these characteristics of a graph.

Requirement **RC.N.5.3** demands that a ciphertext graph does not reveal the number of different triples in the plaintext graph. Most of the approaches for searching in relational databases do not fulfill this requirement. Approaches using row-wise encryption map a plaintext triple to a single ciphertext triple. Thus, the number of triples in the ciphertext graph is identical to the number of triples in the plaintext graph. Approaches applying cell-wise encryption create three ciphertexts for each plaintext triple. Again, the number of ciphertexts reveals the number of plaintext triples. The only exception is Prob-RPE which maps a single plaintext triple to a varying number of ciphertext triples and thereby hides the size of the plaintext graph. All approaches which operate on XML documents reveal the number of plaintext triples in the ciphertext graph. Most of them create the same number of ciphertexts for each plaintext triple. Thus, the number of all ciphertexts in the ciphertext graph is a multiple of all plaintext triples. Brinkman 1 maps a plaintext graph to a hierarchically organized polynomial. However, the structure of this polynomial is essentially the same as the structure of the original XML document and reveals the number of all XML elements. The approaches for searching in encrypted graphs partially fulfill requirement **RC.N.5.3**. Chase and Kamara as well as CryptGraph index a graph by using its subjects and objects. Thus, the two approaches reveal the number of different URIs, blank nodes, and literals at subject and object position. However, they still hide the number of different predicates in the graph. PPGQ encrypts each graph as a single ciphertext and hides the exact number of plaintext triples. As the approach does not normalize the size of a graph, it maps larger plaintext graphs to larger ciphertext graphs and smaller plaintext graphs to smaller ciphertext graphs. This can be used for guessing the number of triples of a ciphertext graph. As described in Section 5.7.11, T-Store reveals the exact number of plaintext triples in the ciphertext graph. The section also suggests to insert random bit strings as fake ciphertexts into the cipher-

text graph and its index. Although this minor modification hides the exact number of triples in the plaintext graph, it still does not eliminate the possibility of distinguishing between large graphs and small graphs. Thus, T-Store fulfills requirement **RC.N.5.3** only partially even after implementing this modification.

5.9.4. Conducing Join Operations on Encrypted Data

As implied by requirement **RC.F.2**, processing a SPARQL query requires a join operation if two or more triple patterns share the same query variables. A join operation is then conducted on the solution sequences of the triple patterns by combining all compatible solution mappings. Solution mappings are compatible if they map the same query variables to the same values. As described in the previous sections, only some approaches for searching in encrypted data support join operations to be conducted on ciphertext data directly and fulfill requirement **RC.F.2**. In contrast, other approaches require a join operation on plaintext data. Even approaches that do support join operations on ciphertext data achieve this by not fulfilling other requirements instead. This section serves three different purposes. First, it outlines the general circumstances and difficulties of conducting join operations on ciphertext data. Second, it describes how and to what extent different approaches for searching in encrypted data solve these issues. Finally, it presents a theoretical approach for conducing join operations on ciphertext data which provides confidentiality of a plaintext graph at the expense of efficiency.

Conducting join operations requires two different steps which are the combination of the solution sequences to be joined and the identification of all compatible solution mappings [243]. These two steps can be processed in arbitrary order or even executed simultaneously. They apply to join operations in general and are necessary for processing both plaintext data and ciphertext data. In order to speed up the joining process, many join algorithms which operate on plaintext data use an index to distinguish between compatible and incompatible solution mappings. This allows a join processor to quickly detect identical plaintext values and to reduce the number of solution mappings to be evaluated. When conducting a join on ciphertext data, using such an index would allow the join processor to distinguish between ciphertext triples that share the same URIs, blank nodes, and literals and ciphertext triples that are completely different from each other. However, this directly contradicts with requirement **RC.N.4** which states that it must be computationally hard to detect identical plaintext values in the ciphertext graph. Thus, using a join index for ciphertext triples would affect the confidentiality of the plaintext graph. On the other hand, fulfilling requirement **RC.N.4** implies that each occurrence of the same plaintext value is mapped to a different ciphertext value. If an index is provided, it must conceal identical plaintext values as well. Although this prevents frequency-based attacks **AC.6** as described in Section 5.7.8, it also prohibits an efficient execution of join operations on ciphertext triples.

The approaches which support join operations on ciphertext data solve this general contradiction at the expense of the fulfillment of other requirements. Apart from the approaches for searching in encrypted data that support join operations, there are also other approaches which focus solely on joins without providing a complete querying en-

vironment. Such approaches include Li and Chen [189], Carbunar and Sion [70], and Furukawa and Isshiki [115]. Li and Chen use a trusted hardware module to conduct join operations on two or more solution sequences. The module requires an encrypted cross product of all sequences as input. It successively decrypts all entries of the cross product to detect and remove incompatible solution mappings. The remaining compatible mappings are returned as the join result. Carbunar and Sion use Bloom filters [41] to detect potentially compatible solution mappings. A Bloom filter is a bit string which represents a set of elements by mapping each element to a bit string and combining all bit strings with binary OR. The resulting bit string allows to quickly determine if an element is possibly part of the set or definitely not in the set. As the design of a Bloom filter produces false positives, using them for joining produces incompatible solution mappings which must be removed in an additional refining step conducted by the user. Furukawa and Isshiki require a join identifier to detect compatible solution mappings.

Current approaches that allow join operations to be conducted on ciphertext data can be divided into four different categories as shown in Table 5.15. Approaches which only support plaintext joins such as T-Store are not listed in the table as they do not fulfill requirement **RC.F.2**. Such approaches require a user to split a query into different sub-queries, process them separately, and join their plaintext solution sequences. This allows the user to access plaintext triples which are not part of the final query result and are removed during the joining process. Thus, approaches which do not fulfill requirement **RC.F.2** cannot ensure the confidentiality of all triples in the graph. Each of the four categories depicted in Table 5.15 has its own drawbacks and does not fulfill one of the requirements **RC.N.2**, **RC.N.3**, and **RC.N.4**. If an approach fulfills one requirement, it achieves this at the expense of another requirement. Approaches which use deterministic encryption or join identifiers do not require a trusted system for query processing or conducting join operations and fulfill requirement **RC.N.2**. They operate on individual ciphertext triples and return them as the final query result without needing an additional refining step (**RC.N.3**). However, the frequency of deterministically encrypted ciphertext triples and join identifiers is identical to the frequency of the plaintext values. This supports frequency-based attacks **AC.6** and contradicts with requirement **RC.N.4**. Approaches based on filtering and refining prevent these attacks and fulfill requirement **RC.N.4**. Instead of operating on individual triples, they combine several ciphertext triples into sets. If at least one triple of a set fulfills a query, all triples of the same set are returned as a preliminary query result. False positives are removed by the user in a refining step. However, the confidentiality of the removed triples is no longer protected. A querying user has access to these triples although they are not part of the final query result as implied by requirement **RC.N.3**. Approaches which require a trusted hardware module for processing joins ensure the confidentiality of all triples. Such a module can operate on probabilistic encryption (**RC.N.4**) and return exact query results (**RC.N.3**). However, the module must be protected against attacks from third parties [116] so that its cryptographic keys and processing memory is not accessible to anyone. Otherwise, the confidentiality of the processed triples cannot be guaranteed. As completely protecting a trusted hardware module is generally difficult [17], using such a module is not recommended (**RC.N.2**).

Table 5.15.: Different approaches for conducting join operations on ciphertext data. None of the current approaches fulfills all three requirements **RC.N.2**, **RC.N.3**, and **RC.N.4**. The table only lists such approaches which support join operations on ciphertext data. Approaches which require join operations on plaintext data such as T-Store are not listed.

Join method	Examples	**RC.N.2**: No trusted component	**RC.N.3**: Exact query result	**RC.N.4**: Data indistinguishability
Deterministic encryption	CryptDB [235]	y	y	n
Join identifiers	Furukawa and Isshiki [115] Prob-RPE [259]	y	y	n
Filtering and refining	Carbunar and Sion [70] Hacıgümüş et al. [137] Jammalamadaka and Mehrotra [165] PPGQ [69]	y	n	y
Secure join hardware	Li and Chen [189]	n	y	y

The analysis of the related work suggests that it is generally difficult to fulfill all three requirements **RC.N.2**, **RC.N.4**, and **RC.N.3** at the same time. In order to support join operations on encrypted graphs without affecting the graph's confidentiality, the solution sequences of all triple patterns must be combined before removing all incompatible solution mappings. Otherwise, the confidentiality of the plaintext triples cannot be guaranteed. In the following, a general concept of a confidential join is outlined which fulfills all three requirements **RC.N.2**, **RC.N.4**, and **RC.N.3**. The concept only defines an abstract process without a particular implementation. It primarily focuses on the confidentiality of the triples and not on the efficiency of its application. It serves as a basis for discussion and demonstrates that it is generally difficult to process join operations on ciphertext data in a practical way. The join is conducted after having determined the solution sequences of all triple patterns and consists of two steps. These steps correspond to the general steps of a join algorithm as described earlier in this sec-

tion. In order to protect the confidentiality of the plaintext triples, all solution sequences are still encrypted. In the first step, a complete cross product of all solution sequences is computed. This step does not distinguish between compatible and incompatible solution mappings and may produce many mappings which are not included in the final join result. The result of this first step is also encrypted and does not reveal any information about the plaintext triples. In the second step, a decryption key is applied to each combined solution mapping. The decryption must only be successful if the solution mapping is a valid join result, i. e., if it contains compatible plaintext triples. Otherwise, the decryption must fail and must not reveal any information about the processed mapping. The second step identifies all compatible solution mappings and creates the final join result. In order to protect the plaintext graph's confidentiality, it is necessary to process all solution mappings from the first step. Any optimization which only processes a subset of the solution mappings would directly contradict with requirement **RC.N.3**.

Although the outlined join process is generally possible, it is highly inefficient. If a query with $m \in \mathbb{N}$ triple patterns is applied to a graph with $n \in \mathbb{N}$ triples, the runtime complexity of processing the query is $\mathcal{O}(n^m)$ as each triple pattern may return n triples at most. In worst case, n^m decryption operations have to be conducted in order to retrieve the final join result. Although most SPARQL queries used in practice contain less than four triple patterns [203, 14], applying a query with three triples patterns to a graph with $100,000$ triples still requires 10^{15} decryption operations. Even if a single decryption operation required only 1 nanosecond to compute, processing 10^{15} decryption operations would still take about 11.5 days. Thus, a join operation based on the sketched concept is not practical unless the decryption itself is very efficient. However, finding a suitable encryption algorithm which is efficient even for larger graphs is very unlikely. On the other hand, any optimization of the concept would violate the confidentiality of the plaintext graph. In conclusion, it is currently almost impossible to provide a join algorithm which is both efficient and confidential at the same time.

5.9.5. Summary

Most approaches for searching in encrypted relational databases, encrypted XML documents, and encrypted graphs can also be used for searching in encrypted RDF graphs. However, none of them fulfills all functional and non-functional requirements defined in Section 5.2. T-Store fulfills most of these requirements with **RC.F.2** being the only exception. Fulfilling requirement **RC.F.2** necessitates the ability to conduct join operations on ciphertext graphs. As discussed in Section 5.9.4, some approaches support join operations at the expense of fulfilling other requirements. The section has also described the general difficulty of providing a join algorithm that can be applied to a ciphertext graph directly and is still efficient enough to be used in practice. Therefore, developing a suitable compromise between efficiency and confidentiality is left for future work.

5.10. Limitations and Future Extensions

As described in the previous section, T-Store fulfills most of the functional and non-functional requirements defined in Section 5.2. In addition to the identified drawbacks, T-Store has other limitations as well when querying encrypted RDF graphs. This section first outlines these limitations and describes their effect on the query process. Afterwards, possible extensions of T-Store are discussed which may be added in the future.

5.10.1. Replacing the Combining Function ϱ

As described in Section 5.7.6, T-Store is vulnerable to attack **AC.4** and allows an attacker to compute a basic key from two authorization keys of the same query type. The attack exploits the relationship of the two authorization keys and their construction via the combining function ϱ. In order to prevent the attack, the combining function must be replaced by a secure alternative. The alternative combining function must share the same features as the current function which is defined in Equation 5.19. In particular, the alternative combining function must satisfy the following conditions:

$$\varrho(b, \{\}) = b, \qquad \forall b \in \{0,1\}^* \tag{5.70}$$

$$\varrho(b, \{b_1, b_2\}) = \varrho(\varrho(b, \{b_1\}), \{b_2\}), \quad \forall b, b_1, b_2 \in \{0,1\}^* \tag{5.71}$$

$$\varrho(\varrho(b, \{b_1\}), \{b_2\}) = \varrho(\varrho(b, \{b_2\}), \{b_1\}), \quad \forall b, b_1, b_2 \in \{0,1\}^* \tag{5.72}$$

These conditions cover the arithmetic capabilities of the combining function which are used by T-Store to create authorization keys ak from basic keys bk and to create query keys qk from authorization keys. As long as a function fulfills these conditions, it can be used with T-Store without changing anything else of the approach. In addition, the combining function must also prevent the attacks **AC.3**, **AC.4**, and **AC.5** as described in Section 5.7. This can be achieved by choosing a combining function ϱ which makes it difficult for an attacker to determine b when given b_1 and b_2 with $\varrho(b, \{b_1\}) = b_2$ and $b, b_1, b_2 \in \{0,1\}^*$. Finding a suitable combining function which prevents the attacks and fulfills the three conditions of Equations 5.70 to 5.72 is left for future work.

5.10.2. Query Results with Blank Nodes

An RDF graph represents blank nodes as blank node identifiers. The scope of these identifiers is restricted to a particular graph, i. e., the same identifier can be used in other graphs as well and refers to another blank node. Applying a SPARQL query to a plaintext RDF graph may produce a query result with blank node identifiers. In this case, the scope of the identifiers is restricted to the query result and the identifiers in the query result are independent from the blank node identifiers in the queried graph [140]. T-Store also supports blank node identifiers in query results. However, these identifiers are identical to the blank node identifiers in the queried graph and refer to the same blank nodes. This may reveal more information about the plaintext graph and its internal structure than the data owner has indented. It allows an authorized user to relate different query results to each other by analyzing their common blank node identifiers.

Such a relation is not necessary possible when applying a SPARQL query to a plaintext graph.

5.10.3. Distributing Authorization Keys

Users are allowed to apply queries to ciphertext graphs by receiving authorization keys from the data owner. The secure distribution of authorization keys is therefore crucial to ensure the confidentiality of a plaintext graph. Distributing the keys requires a secure communication channel between an authorized user and the data owner which ensures that the transmitted keys cannot be intercepted by any party. In addition, the data owner should verify the identity of a user before sending an authorization key. A public key certificate can be used for both creating the secure communication channel and for verifying the user's identity. It provides a mapping from a public key to its owner and is managed by a public key infrastructure (PKI). Further information about PKIs is provided in Section 4.9.4. For example, the data owner could apply a user's public key certificate to send an authorization key via an encrypted e-mail. Possible standards for e-mail encryption are S/MIME [245] and PGP [320]. Transmitting an authorization key in an encrypted e-mail by using a verified public key certificate ensures that only the legitimate owner of the certificate can access the key.

Even a secure communication channel cannot prevent a user from forwarding authorization keys to other parties. Forwarding such keys circumvents the data owner's query authorizations and affects the confidentiality of the plaintext graph. T-Store does not prohibit the re-distribution of authorization keys as they are not bound to individual users. Instead, an authorization key which encodes a specific restriction pattern is identical for all users. If several users have access to the same key and the key is sent to another party, the data owner cannot identify the malicious user. In order to detect the user, the data owner could create a different ciphertext graph with different basic keys for each authorized user. Although this does not prevent a user from both forwarding authorization keys together with the ciphertext graph, the data owner is at least able to determine the malicious user. However, using different keys for each user increases the number of keys that the data owner must manage.

5.10.4. Refining Query Authorizations

Query authorization in T-Store is currently restricted to distributing individual authorization keys to users. An authorization key defines the basic matching conditions of a single SPARQL triple pattern. However, as described in Section 5.1.4, SPARQL queries are more complex and do not only consist of a single triple pattern. Instead, they have a query algebra with an arbitrary number of triple patterns and a query form that defines the format of the query result. Consequently, when authorizing a user to apply queries to a ciphertext graph, the data owner should be able to precisely define the complete SPARQL query that is being authorized. In particular, the data owner should be able to specify all triple keys of a query algebra as well as a corresponding query form in such a way that all triple keys and the query form can only be used in combination with each

other. I. e., it must not be possible for a user to extract a triple key from one authorized query and include it in another query or to replace the query form of an authorized query with an alternative query form.

In order to support more precise query authorizations, the authorization mechanism of T-Store must be further improved. This includes the implementation and enforcement of join operations as described in Section 5.9.4 as well as the enforcement of query forms. Enforcing join operations prevents an authorized user from applying an individual triple key to a ciphertext graph without applying the other triple keys of the query as well. This prohibits the user from receiving any temporary query results. Enforcing a query form ensures that the query result does not provide any more information about the plaintext triples than the data owner has originally intended. For example, the query form of a `SELECT` query may only return the variable bindings of a single query variable and hide the bindings of all other variables that are defined in the triple keys. In summary, enforcing join operations and query forms ensures that a user can only apply a query to a ciphertext graph if this exact query has been authorized by the data owner.

5.10.5. Revoking Basic Keys and Ciphertexts

The data owner uses eight different basic keys for creating a ciphertext graph which is either published on the web or sent to all authorized users. If a basic key is compromised, it should be revoked since the confidentiality of the plaintext graph can no longer be guaranteed. In addition, the ciphertext graph should be revoked as well since the data owner can no longer manage the parties who apply queries to it. However, T-Store does not support the revocation of basic keys or ciphertext graphs. Once a ciphertext graph has been made available, it can no longer be revoked. Since basic keys are bound to a particular ciphertext graph, they cannot be revoked as well or replaced with new keys. In general, any offline approach for searching in encrypted data suffers from this drawback as it does not use a central system which regulates all access to the ciphertext data. In contrast, online approaches such as CryptDB [235], Prob-RPE [259], and Y. Yang et al. [315] support the revocation of query authorizations but require a central server for storing the data. To circumvent the problem of key revocation, both the data owner and all authorized users must ensure that their key material is safe and cannot be accessed by any unauthorized party.

5.10.6. Additional Support for the SPARQL Algebra

The current design of T-Store only supports a small fragment of the SPARQL algebra [140] and is limited to matching triple patterns against a ciphertext graph. However, SPARQL also provides additional features such as filters, solution sequence modifiers, and aggregates. A filter applies a boolean expression to each solution mapping of a solution sequence and removes all mappings which do not match the expression. Filters allow, e. g., value comparisons, type checking, and regular expressions. They can be integrated into T-Store by using functional encryption [43] which supports the application of arbitrary functions to ciphertext data. Solution sequence modifiers are evaluated on

all solution mappings of a solution sequence and affect the whole sequence. An example modifier is `ORDER BY` which sorts all solution mappings in ascending or descending order. Applying this modifier requires order-preserving encryption which ensures that the order of the ciphertexts is the same as the order of their corresponding plaintexts. To support `ORDER BY`, T-Store can be combined with approaches which use order-preserving encryption such as [304, 137]. Aggregates apply aggregation functions to all solution mappings of a solution sequence. Example aggregates are `SUM`, which adds up numerical literals, and `AVG`, which computes the mean value of such literals. Both aggregates can be supported by integrating approaches for homomorphic encryption [111] into T-Store. Example approaches which support aggregates on ciphertext data include [76, 126].

5.11. Summary

This chapter has presented T-Store, an approach for applying SPARQL queries to encrypted RDF graphs. A plaintext graph is encrypted by a data owner who authorizes different users to apply queries to the resulting ciphertext graph. T-Store supports query templates which represent sets of similar queries sharing the same query parameters. Instead of authorizing each query individually, the data owner can use a query template and allow a user to define the remaining query parameters herself. T-Store limits the communication between the data owner and an authorized user to a minimum and allows the user to conduct the query processing offline on her local system. Query processing is solely based on conducting cryptographic operations on the ciphertext graph and does not need a trusted computing device at the user's side. Query results are exact and do not require any post-processing conducted by the user. Apart from an authorized user, all other parties are prohibited from accessing any plaintext triples of the queried graph. Thus, T-Store provides confidentiality of the graph and answers research question **RQ.1**. Although the approach fulfills most of its requirements, it still has some drawbacks which may affect a graph's confidentiality. Eliminating these drawbacks and enhancing T-Store with additional features is left for future work.

Chapter 6.

Conclusion

This thesis has presented three different security mechanisms for achieving secure Semantic Web data management. The mechanisms implement the four security requirements confidentiality, integrity, authenticity, and compliant availability of Semantic Web graph data. The first security mechanism is InFO, a policy language for regulating information flow in open and distributed networks. A policy defines the conditions under which Semantic Web data is accessible and contains all details for technically enforcing them. InFO implements compliant availability of Semantic Web data and answers research question **RQ.4**. The second security mechanism is Siggi, a formal framework for digitally signing arbitrary Semantic Web graphs. A graph's signature is invalidated if the graph is modified by an unauthorized party. This affects the graph's integrity and authenticity since its original creator has not approved of the modifications. Thus, Siggi implements integrity and authenticity of Semantic Web data and answers research questions **RQ.2** and **RQ.3**. The third security mechanism is T-Store, an approach for searching in encrypted Semantic Web data. Only authorized parties are able to access particular triples of a plaintext graph by applying queries to its corresponding ciphertext graph. At the same time, all other triples remain inaccessible. Thus, T-Store implements confidentiality of Semantic Web graphs and answers research question **RQ.1**. The rest of this chapter describes how the three security mechanisms are used for implementing the scenarios presented in Chapter 2, summarizes their main contributions, and outlines possible future work.

6.1. Implementing the Scenarios

The scenario for regulating Internet communication defined in Section 2.1 involves different authorities which create regulation policies and distribute them between each other. The policies are created with InFO and describe allowed and prohibited communication flow. In order to ensure the integrity and authenticity of a policy when transmitting it to another authority, each policy is signed by its creator using the graph signing framework Siggi. InFO policies are enforced by different communication systems which maintain a log database of all regulated transmissions. The log database is encrypted with T-Store in order to protect the log entries' confidentiality and to ensure that only authorized parties can retrieve particular entries. The scenario for securing medical data records in

electronic healthcare is covered in Section 2.2 and focuses on a patient who manages her own medical records. These records are encrypted with T-Store in order to ensure their confidentiality. Only medical professionals can access specific records after being authorized by the patient. In addition to the patient's medical records, medical professionals create medical records as well which are transmitted between different medical institutions. Before transmitting a medical record, it is signed with Siggi in order to ensure the record's integrity and authenticity and prevent it from any unauthorized modifications. InFO is used to ensure that the signed records are only sent to other medical institutions via an encrypted communication channel. At the same time, InFO blocks any other transmission of medical records.

6.2. Summary of the Main Contributions

InFO, Siggi, and T-Store achieve confidentiality, integrity, authenticity, and compliant availability of Semantic Web data in open and distributed networks such as the Internet. Such networks are not regulated by a single authority which defines specific restrictions on the network's components and usage. Instead, they have a heterogeneous environment with different computer systems and data formats. InFO is designed for creating policies which are implemented in networks without a central regulatory authority. In contrast to other policy language, InFO policies can be enforced by various communication systems such as application-level proxy severs, name servers, and routers. This allows it to apply InFO policies directly to the Internet without requiring any specific regulation systems. InFO's modular and extensible design allows it to support additional enforcing systems as well. An InFO policy is enriched by human-readable background information which covers the policy's organizational motivation and its legal justification. In order to precisely describe such information, InFO allows to integrate different legal ontologies which provide a rich vocabulary for various legal scenarios. Although some other policy languages also allow organizational background information to be included into a policy as well, their support for legal information is very limited. InFO supports a conflict resolution mechanism for resolving conflicts of contradicting rules within one or more policies. This is especially important in open and distributed networks in which policies are created by different authorities with different intentions.

Siggi defines a generic signature pipeline which can be configured with various algorithms to achieve different features. The generic design of the pipeline also supports the creation of new algorithms by adopting the framework's formal specification. The flexible configurability of Siggi allows it to be used in heterogeneous environments where there is no standard set of algorithms that are used by all authorities. Instead, each authority may use its own configuration of the framework. Siggi supports iterative signing of graph data by signing already signed data again. This can be used for provenance tracking in which each party receives a signed graph, signs it again, and sends it to the next party. A signature created with Siggi only covers a signed graph's semantics and not its syntax. Therefore, the signature does not rely on a particular serialization of the graph and is still valid after re-encoding the graph with a different serialization format.

For this reason, the signature is permanently attached to the signed data and does not rely on a secure communication channel. The signature value can be accompanied with additional metadata which further describes its creation such the date and time the graph was signed or the name of the signing party.

T-Store distinguishes between a data owner who encrypts a plaintext graph and authorized users who perform queries on the resulting ciphertext graph. The approach requires only little communication between the data owner and a user. Instead of authorizing each query individually, similar queries can be combined into query templates. A query template is an incomplete query which must be further specified by an authorized user with additional query parameters before applying it to the ciphertext graph. Query processing is conducted offline by the user without involving any online system or a trusted system located at the user's side.

The main contributions of this thesis are available online at `https://github.com/akasten/`. InFO is provided as a set of OWL ontologies which also include the three domain ontologies for application-level proxy severs, name servers, and routers. In addition, the mapping ontologies of the legal ontologies and the content classification schemes, which are further described in Appendix B, are available as well. Siggi is available as the source code of its prototypical Java implementation which covers both the generic signing framework and the four example configurations discussed in Section 4.4. Furthermore, the signature ontology which is used by the assembly function of Siggi and further described in Appendix C is also provided. T-Store is available as the source code of its two prototypical Java implementations which cover the two variants T-Store BSC and T-Store NDX. In addition, the SPARQL queries used for evaluating the two variants as described in Section 5.6.1 as well as the extended log format ontology, which is used in the scenario of T-Store and further described in Appendix D, are also available.

6.3. Outlook and Future Work

All three security mechanisms have been prototypically implemented, applied to two example scenarios, and evaluated against their respective requirements. However, all three mechanisms can still be improved and further analyzed. InFO can be evaluated by simulating a large computer network with several different communication end nodes such as web servers and client systems as well as intermediary communication nodes like routers and application-level proxy servers. The intermediary nodes can be used as enforcing systems to implement different regulation policies. Such a simulation can assess the scalability of InFO policies and its enforcing systems in a larger network. In addition, it can provide general insights into large scaled network regulations and possible side-effects. Siggi can be used as a basis to provide various applications for securing integrity and authenticity of Semantic Web graphs. An example application is provenance tracking of signed graph data which allows it to trace the graph's path along different authorities. Furthermore, Siggi can be used for applications which aim at providing trust between different interacting parties. Finally, the framework can also be integrated into ontology editors to provide permanent signatures for the created graph

data. Similar to Siggi, T-Store can also be used as a foundation to provide different applications for securing Semantic Web data. Example applications are outlined in this thesis and cover an encrypted log database and a confidential storage for medical records. In addition, T-Store's prototypical implementation can be further extended to create a generic triple store which supports arbitrary SPARQL queries on encrypted graphs and restricts access to particular triples to authorized users only.

Appendix A.

Algorithms and Domain Ontologies of the InFO Policy Language

This appendix provides additional details on the InFO policy language which is described in Chapter 3. In particular, it describes different algorithms for resolving conflicts between two or more flow control rules. These algorithms are generic and can be used within different application domains. Thus, they are part of InFO's Technical Regulation patterns. Apart from these algorithms, this appendix also provides further details on the three domain-specific extensions for implementing InFO policies on routers, name servers, and proxy servers. These details cover specific flow control rules as well as additional algorithms for resolving conflicts.

A.1. Generic Algorithms for Resolving Conflicts

The Technical Regulation of InFO defines six different types of algorithms for resolving conflicts between flow control rules of one or more flow control policies. The conflicts are distinguished between modality conflicts and application specific conflicts [192]. Modality conflicts exist between two flow control rules if one rule allows a particular communication flow and the other rule prohibits it. In contrast, application specific conflicts exist between an enforcing system and any flow control rule which cannot be completely interpreted by this enforcing system. The algorithms of the Technical Regulation are used in the Flow Control Policy Pattern described in Section 3.3.4 and in the Flow Control Meta-Policy Pattern described in Section 3.3.5. They are part of a flow control policy or a flow control meta-policy. The different types of algorithms are represented by the classes `LocalNonApplicabilityAlgorithm`, `LocalConflictResolutionAlgorithm`, `GlobalNonApplicabilityAlgorithm`, `GlobalConflictResolutionAlgorithm`, `RulePriorityAlgorithm`, and `PolicyPriorityAlgorithm`. Particular algorithms are modeled as subclasses of these generic classes. In order to better distinguish between the different algorithms and to avoid name clashes, each type of algorithm is associated with a unique namespace. All algorithms are evaluated by the enforcing system which implements the flow control regulation. The detailed process of resolving conflicts is described in Section 3.3.5.

A `LocalNonApplicabilityAlgorithm` solves application-specific conflicts within a single flow control policy. The Technical Regulation of InFO defines the class `DiscardNonApplicableRuleAlgorithm` as a particular instance of such an algorithm. The algorithm removes all flow control rules from the policy which cannot be interpreted by the policy's enforcing system. Other algorithms for resolving application-specific conflicts of a single policy are not provided. However, it is also possible to add new algorithms due to InFO's open design. `GlobalNonApplicabilityAlgorithm`s solve application-specific conflicts of one or more flow control policies. Particular instances are `DiscardNonApplicableRuleAlgorithm` and `DiscardNonApplicablePolicyAlgorithm`. The first algorithm has the same semantics as its local counterpart[1] and removes all conflicting rules while leaving all other rules intact. On the other hand, the second algorithm removes all policies which contain at least one conflicting rule.

`LocalConflictResolutionAlgorithm`s resolve modality conflicts between contradicting flow control rules of the same policy. InFO's Technical Regulation provides the class `DiscardConflictingRulesAlgorithm` as an example of such algorithms. The algorithm resolves modality conflicts by removing all affected rules from a policy. Similarly, algorithms of type `GlobalConflictResolutionAlgorithm` resolve modality conflicts between contradicting flow control rules of the one or more policies. Specific algorithms are `DiscardAffectedRulesAlgorithm` and `DiscardAffectedPoliciesAlgorithm`. The first algorithm removes all conflicting rules and leaves all other rules intact while the second algorithm removes all policies which contain at least one conflicting rule.

`PolicyPriorityAlgorithm`s define the order of evaluating different flow control policies by the same enforcing system. `PreferLatestPolicyAlgorithm` states that newer policies must be preferred to older policies and `PreferOldestPolicyAlgorithm` prefers older policies instead of newer policies. The algorithm `EvaluatePolicyOrderingAlgorithm` requires an explicit order between policies with the properties `follows` and/or `precedes` and states that this order shall be evaluated. Similarly, `RulePriorityAlgorithm`s define the order of evaluating the rules of a particular policy. `EvaluateRuleOrderingAlgorithm` requires an explicit order between the rules and evaluates this order.

A.2. Details of the Router Ontology

The Router Ontology is a domain-specific extension of the InFO policy language for routers. It provides several concepts and axioms for describing flow control policies which shall be enforced by these network nodes. The concepts cover router-specific flow control rules and rule priority algorithms. New patterns are not defined in the ontology.

A.2.1. Flow Control Rules

The Router Ontology refines the generic allowing flow control rule and the denying flow control rule of the Flow Control Rule Pattern and the redirecting flow control rule of

[1] Please note that both algorithms share the same local name but have different URIs. Therefore, their fully qualified name is unique and name clashes are prevented.

the Redirecting Flow Control Rule Pattern with router-specific concepts. Each of the refined rules requires a router as its enforcing system and may also require additional parameters. An allowing flow control rule explicitly allows a sender to communicate with a receiver. Further refinements of the generic class `AllowingFlowControlRuleMethod` define how the sender and receiver shall be identified. The Router Ontology defines the following allowing flow control rules:

`IPAddressAllowingRuleMethod`
: This rule explicitly permits a communication between a sender and a receiver based on the IP addresses of the two systems. The rule requires at least one IP address.

`PortNumberAllowingRuleMethod`
: This rule explicitly permits a communication between a sender and a receiver based on the port numbers they use for establishing their communication channel. As each of the two communication systems choses their own port number, the port number is directly associated with each system and not with the communication channel. The rule requires at least one port number.

`SocketAddressAllowingRuleMethod`
: This rule explicitly permits a communication between a sender and a receiver based on their sockets of the communication channel. A socket address consists of an IP address and a port number. The rule requires at least one IP address and one port number which form the socket address.

`TransportLayerProtocolAllowingRuleMethod`
: The rule explicitly permits a communication between a sender and a receiver based on the transport layer protocol of their communication channel. Example transport layer protocols are TCP and UDP. The rule requires at least one transport layer protocol to be specified.

A denying flow control rule prevents a sender from communicating with a receiver. Further refinements of the generic class `DenyingFlowControlRuleMethod` define how the sender and receiver shall be identified. The Router Ontology defines the following denying flow control rules:

`IPAddressBlockingRuleMethod`
: This rule prohibits a communication between a sender and a receiver based on their IP addresses. The rule requires at least one IP address.

`PortNumberBlockingRuleMethod`
: This rule prohibits a communication between a sender and a receiver based on the port numbers they use for establishing their communication channel. The rule requires at least one port number to be specified.

`SocketAddressBlockingRuleMethod`
: This rule prohibits a communication between a sender and a receiver based on the socket addresses which they use for establishing their communication channel.

The rule requires at least one IP address and one port number which form a socket address.

`TransportLayerProtocolBlockingRuleMethod`
This rule prohibits a communication between a sender and a receiver based on the transport layer protocol of their communication channel. The rule requires at least one transport layer protocol.

A redirecting flow control rule prevents a sender from communicating with a receiver by redirecting it to a different receiver. Further refinements of the generic class `RedirectingFlowControlRuleMethod` define the parts of the original receiver's address that shall be replaced with other address information. The Router Ontology defines the following redirecting flow control rules:

`IPAddressRedirectingRuleMethod`
This rule prohibits a communication between a sender and a receiver by redirecting the sender to a different receiver with another IP address. The rule requires at least one IP address as the redirection target.

`PortNumberRedirectingRuleMethod`
This rule prohibits a communication between a sender and a receiver by redirecting the sender to a different receiver with another port number. The IP address of the original receiver may stay the same. The rule requires at least one port number as the redirection target.

`SocketAddressRedirectingRuleMethod`
This rule prohibits a communication between a sender and a receiver by redirecting the sender to a different receiver with another IP address and another port number. The rule requires at least one IP address and one port number which form the socket address of the redirection target.

A.2.2. RulePriorityAlgorithms

The Router Ontology defines two additional rule priority algorithms which can be used together with the generic algorithms described in Appendix A.1. Other types of algorithms are not provided by the Router Ontology. The provided algorithms are `PreferShortestSubnetMaskAlgorithm` and `PreferLongestSubnetMaskAlgorithm` which are both modeled as subclasses of `RulePriorityAlgorithm`. The two algorithms operate on network addresses and thus only affect flow control rules which regulate the access to computer networks. Single network systems such as clients and servers are not affected. The first algorithm sorts all flow control rules in ascending order according to the subnet mask of the regulated network's IP address. The first element in the resulting order has the shortest subnet mask. The shorter the subnet mask of a computer network is, the more nodes the network contains. Thus, the algorithm prefers flow control rules which cover larger computer networks to such rules which regulate access to smaller networks. The second algorithm is inverse to the first algorithm. It also sorts all flow control rules

in descending order according to the subnet mask of the regulated network. However, the algorithm prefers longer subnet masks and thus smaller networks.

A.3. Details of the Name Server Ontology

The Name Server Ontology is a domain-specific extension of the InFO policy language for name servers. It provides several concepts and axioms for describing flow control policies which shall be enforced by such systems. The concepts cover name server-specific flow control rules and rule priority algorithms. New patterns are not defined in the ontology. The flow control rules and algorithms of the Name Server Ontology operate on domains and domain names. A domain is a hierarchically organized tree in the domain name system and consists of several domain names. Each domain name corresponds to an individual nodes in the tree. Domains are associated with computer networks while individual domain names represent single computer systems. A domain is identified by a domain name which corresponds to the root node of the tree.

A.3.1. Flow Control Rules

The Name Server Ontology refines the generic allowing flow control rule and the denying flow control rule of the Flow Control Rule Pattern and the redirecting flow control rule of the Redirecting Flow Control Rule Pattern with name server-specific concepts. Each of the refined rules requires a name server as its enforcing system and may also require additional parameters. An allowing flow control rule explicitly allows a sender to communicate with a receiver. Further refinements of the generic class `AllowingFlowControlRuleMethod` define how the sender and receiver shall be identified. The Name Server Ontology defines the following allowing flow control rules:

`DomainAllowingRuleMethod`
: This rule explicitly permits a communication between a sender and a receiver based on their domains. The rule requires at least one domain name which is the domain's root node.

`DomainNameAllowingRuleMethod`
: This rule explicitly permits a communication between a sender and a receiver based on their domain names. The rule requires at least one domain name.

A denying flow control rule prevents a sender from communicating with a receiver. Further refinements of the generic class `DenyingFlowControlRuleMethod` define how the sender and receiver shall be identified. The Name Server Ontology defines the following denying flow control rules:

`DomainBlockingRuleMethod`
: This rule prohibits a communication between a sender and a receiver based on their domains. The rule requires at least one domain name which corresponds to the domain's root node.

DomainNameBlockingRuleMethod
: This rule prohibits a communication between a sender and a receiver based on their domain names. The rule requires at least one domain name.

A redirecting flow control rule prevents a sender from communicating with a receiver by redirecting to a different receiver. Further refinements of the generic class `RedirectingFlowControlRuleMethod` define the parts of the original receiver's domain or domain name that shall be replaced with other address information. The Name Server Ontology defines the following redirecting flow control rules:

DomainRedirectingRuleMethod
: This rule prohibits a communication between a sender and a receiver by redirecting the sender to a different domain. The original receiver is identified by its domain which is represented by the domain name of its root node. Apart from this domain name, the rule also requires another domain name which identifies the domain used as the redirection target.

DomainNameRedirectingRuleMethod
: This rule prohibits a communication between a sender and a receiver by redirecting the sender to a different domain name. The original receiver is identified by its domain name. Apart from this domain name, the rule also requires another domain name which is used as the redirection target.

A.3.2. RulePriorityAlgorithms

The Name Server Ontology extends the set of generic rule priority algorithms described in Appendix A.1 with four additional algorithms. Other types of algorithms are not provided. The ontology provides the following rule priority algorithms:

PreferDomainNameToDomainAlgorithm
: This algorithm states that flow control rules regulating domains names shall be preferred to such rules which cover whole domains. Thus, the algorithms prefers rules which regulate the access to single computer systems to such rules which cover whole networks.

PreferDomainToDomainNameAlgorithm
: This algorithm states that flow control rules regulating whole domains shall be preferred to such rules which cover single domain names. Thus, the algorithm prefers rules which regulate the access to computer networks. Using this algorithm is not recommended as it may lead to overblocking by discarding more precise flow control rules. It is mainly included in the Name Server Ontology for reasons of completeness.

PreferLongestDomainNameAlgorithm
: This algorithm sorts all flow control rules which are based on domain names in descending order according to the length of their domain name. Thus, the first

rule of the resulting order has the longest domain name. The length of a domain name is measured by the number of labels it contains. A label corresponds to the string between two dots in a domain name. The algorithm basically prefers more precise rules which affect less computer systems.

`PreferShortestDomainNameAlgorithm`
: This algorithm sorts all flow control rules which are based on domain names in ascending order according to the length of their domain name. Thus, the first rule of the resulting order has the shortest domain name. This algorithm essentially prefers more imprecise rules which affect more computer systems. Using this algorithm is not recommended as it may lead to overblocking. It is included in the Name Server Ontology for reasons of completeness.

A.4. Details of the Application-Level Proxy Ontology

The Application-Level Proxy Ontology is a domain-specific extension of the InFO policy language for proxy servers. It provides several concepts and axioms for describing flow control policies which shall be enforced by these servers. The concepts cover proxy server-specific flow control rules and rule priority algorithms. The ontology also defines a new pattern for describing denying flow control rules which modify the original content.

A.4.1. Flow Control Rules

The Application-Level Proxy Ontology refines the generic allowing flow control rule and the denying flow control rule of the Flow Control Rule Pattern and the replacing flow control rule of the Replacing Flow Control Rule Pattern with proxy server-specific concepts. Each of the refined rules requires a proxy server as its enforcing system and may also require additional parameters. Furthermore, the ontology also introduces the Content Modifying Rule Pattern as a specialization of the Flow Control Rule Pattern. An allowing flow control rule explicitly allows a sender to communicate with a receiver by transmitting content via a channel. Further refinements of the generic class `AllowingFlowControlRuleMethod` define how the transmitted content shall be identified. The Application-Level Proxy Ontology defines the following allowing flow control rules:

`URLAllowingRuleMethod`
: This rule explicitly permits the transmission of content that is identified by its URL. The rule requires at least one URL.

`HashValueAllowingRuleMethod`
: This rule explicitly permits the transmission of content that is identified by its hash value. The rule may be used for regulating access to content independent from its URL. Even if the same content is provided at different URLs, its hash value remains the same. Thus, this rule is more robust than rules based on URLs. The rule requires at least one hash value.

FileExtensionAllowingRuleMethod
This rule explicitly permits the transmission of content based on its file extension. Example file extensions are `.jpg` for JPEG images and `.mp3` for MP3 audio files. The rule requires at least one file extension.

MIMETypeAllowingRuleMethod
This rule explicitly permits the transmission of content based on its MIME type. A MIME type [112] consists of a type and a subtype and can be used for describing the format of a file. Example MIME types are `image/jpeg` for JPEG images and `audio/mp3` for MP3 audio files. The rule requires at least one MIME type.

A denying flow control rule prohibits the transmission of content between a sender and a receiver. Further refinements of the generic class `DenyingFlowControlRuleMethod` define how transmitted content shall be identified. The Application-Level Proxy Ontology defines the following denying flow control rules:

URLBlockingRuleMethod
This rule prohibits the transmission of content that is identified by its URL. The rule requires at least one URL.

Hash value blocking RuleMethod
This rule prohibits the transmission of content that is identified by its hash value. The rule requires at least one hash value.

FileExtensionBlockingRuleMethod
This rule prohibits the transmission of content based on its file extension. The rule requires at least one file extension.

MIMETypeBlockingRuleMethod
This rule prohibits the transmission of content based on its MIME type. The rule requires at least one MIME type.

A replacing flow control rule prohibits the transmission of content between a sender and a receiver by exchanging the original content with other content. The Application-Level Proxy Ontology defines the class `FileReplacingRuleMethod` as a subclass of the generic class `ReplacingFlowControlRuleMethod`. The rule prohibits the transmission of the original content by replacing the file to be transmitted with a different, predefined file. This file is fixed and must exist before the rule is applied. The rule may be used for replacing certain images with another image providing an explanation for the reasons of the image blocking. Other rules for replacing the transmitted content are not provided by the Application-Level Proxy Ontology. However, the ontology also defines a pattern for modifying the original content to be transmitted. This pattern is the Content Modifying Rule Pattern which is depicted in Figure A.1. The pattern allows to deny a particular communication flow by modifying the original content with a specified modification algorithm. In contrast to the `FileReplacingRuleMethod`, a content modification rule does not use a predefined file as the replacement target. Instead, it requires the

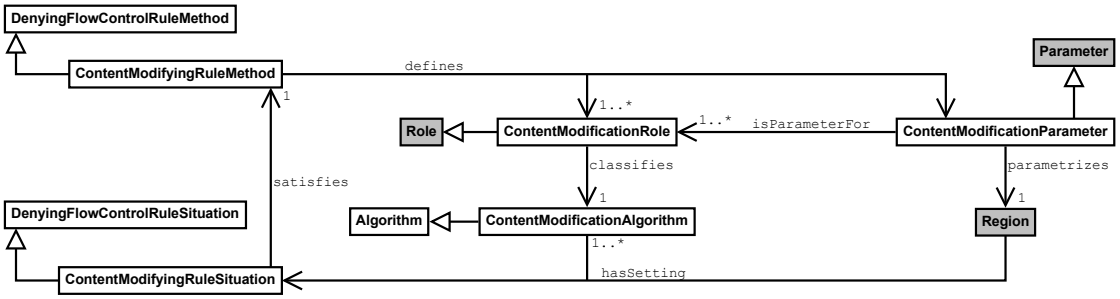


Figure A.1.: Content Modifying Rule Pattern.

computation of such a target for each content file. The Content Modifying Rule Pattern extends the Flow Control Rule Pattern with a content modification algorithm and optional modification parameters. These classes allow to describe the algorithm used for modifying the original content as well as a list of additional modification parameters. The Application-Level Proxy-based Flow Control Ontology defines the following content modification rules:

`ImageRescalingRuleMethod`
: This rule prohibits the transmission of the original image by rescaling it with a corresponding image rescaling algorithm. The transmitted content must be an image file. Other files are not affected by this rule. The rule requires at least one image rescaling algorithm.

`ImageBlurringRuleMethod`
: This rule prohibits the transmission of the original image by blurring it with a corresponding image blurring algorithm. The transmitted content must be an image file. This rule may be used to conceal the content of adult images without having to block the web page containing this image. The rule requires at least one image blurring algorithm.

A.4.2. RulePriorityAlgorithms

The Application-Level Proxy Ontology extends the set of generic rule priority algorithms described in Appendix A.1 with several additional algorithms. Other types of algorithms are not provided by the Application-Level Proxy Ontology. Most of the provided rule priority algorithms operate on URLs or URL fragments such as query strings. Rules which are not based on URLs are not affected by these algorithms. The ontology provides the following rule priority algorithms:

`PreferSingleFileToWebSiteAlgorithm`
: This algorithm states that flow control rules regulating single web files shall be preferred to such rules which covers whole websites. Websites are collections of several related web pages. Thus, the algorithm basically states that more precise rules shall be preferred to rules which affect too many web pages.

PreferWebSiteToSingleFileAlgorithm

This algorithm states that flow control rules regulating whole websites shall be preferred to such rules which only covers single web files. This algorithm essentially prefers more imprecise rules which affect more web pages. Using this algorithm is not recommended as it may lead to overblocking. It is mainly included in the Application-Level Proxy Ontology for reasons of completeness.

PreferLongestDomainNameAlgorithm

This algorithm sorts the flow control rules in descending order according to the length of their URL's domain name. Thus, the first rule of the resulting order has the longest domain name. The length of a domain name corresponds to the number of labels it contains. The algorithm basically prefers more precise rules which affect less web pages. The algorithm is semantically equivalent with the algorithm of the Name Server Ontology of the same name.

PreferShortestDomainNameAlgorithm

This algorithm sorts the flow control rules in ascending order according to the length of their URL's domain name. Thus, the first rule of the resulting order has the shortest domain name. This algorithm essentially prefers more imprecise rules which affect more web pages. Using this algorithm is not recommended as it may lead to overblocking. It is included in the Application-Level Proxy Ontology for reasons of completeness. The algorithm is semantically equivalent with the algorithm of the Name Server Ontology of the same name.

PreferLongestPathAlgorithm

This algorithm sorts the flow control rules in descending order according to the length of their URL's local path. Thus, the first rule of the resulting order has the longest path. The length of a path corresponds to the number of directory sections it contains. The algorithm basically prefers more precise rules which affect less web pages.

PreferShortestPathAlgorithm

This algorithm sorts the flow control rules in ascending order according to the length of their URL's local path. Thus, the first rule of the resulting order has the shortest path. The algorithm essentially prefers more imprecise rules which affect more web pages. Using this algorithm is not recommended as it may lead to overblocking. It is included in the Application-Level Proxy Ontology for reasons of completeness.

PreferLongestQueryStringAlgorithm

This algorithm sorts the flow control rules in descending order according to the length of their URL's query string. Thus, the first rule of the resulting order has the longest query string. The length of a query string corresponds to the number of query parameters. The algorithm basically prefers more precise rules which affect less web content.

`PreferShortestQueryStringAlgorithm`

This algorithm sorts the flow control rules in ascending order according to the length of their URL's query string. Thus, the first rule of the resulting order has the shortest query string. The algorithm essentially prefers more imprecise rules which affect less web content. Using this algorithm is not recommended as it may lead to overblocking. It is included in the Application-Level Proxy Ontology for reasons of completeness.

Appendix B.

Integrating External Vocabularies into the InFO Policy Language

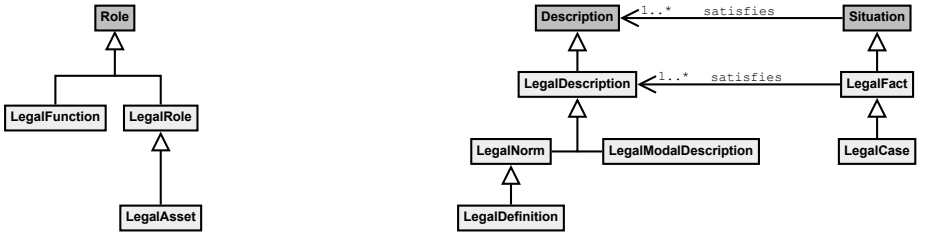
This appendix describes how legal ontologies and content labeling schemes are integrated into the InFO policy language which is introduced in Chapter 3. InFO policies focus on the technical implementation details for regulating information flow and associate them with human-readable background information. This information can be further enriched by integrating external legal ontologies and content labeling schemes into InFO which support more elaborate descriptions. In particular, legal ontologies provide a vocabulary for annotating a flow control policy with its organizational motivation and legal justification whereas content labeling schemes further describe the regulated content.

B.1. Integrating Legal Ontologies into InFO

This section describes how legal ontologies can be integrated into the InFO pattern system. The discussed legal ontologies are the Core Legal Ontology (CLO) [123, 118] and the Legal Knowledge Interchange Format (LKIF) [146, 147]. The ontologies are integrated by extending the patterns of the Legal Regulation and the Organizational Regulation of InFO as introduced in Section 3.3.6. These patterns are the Code of Conduct Pattern, the Flow Regulation Norm Pattern, and the Legislation Pattern. The integration of CLO and LKIF is implemented by using the mapping ontologies CLOMapping and LKIFMapping, respectively, as depicted in Figure 3.4. Each mapping ontology imports the InFO patterns and the legal ontology to be integrated. It defines subclass and subproperty relationships between the concepts and properties of InFO and the legal ontology. Where necessary, the mapping ontology also defines additional classes and properties as well as new ontology design patterns. The integration is conducted according to the general process described in Section 3.3.7.

B.1.1. Integrating the Core Legal Ontology

The Core Legal Ontology (CLO) [123, 118] is a legal core ontology which provides various classes, properties, and design patterns for modeling different legal aspects such as legal documents or legal norms. It is based on DOLCE+DnS Ultralite (DUL) [119] and



(a) Hierarchy of legal roles **(b)** Hierarchy of legal descriptions and situations

Figure B.1.: Important classes of the Core Legal Ontology. The classes are modeled as subclasses of corresponding DUL classes. Dark gray identifies classes from DUL whereas light gray marks classes of the Core Legal Ontology.

extends many classes and patterns of DUL. The most important classes of CLO are depicted in Figure B.1. Figure B.1a shows how CLO extends the class `Role` with more specific subclasses such as `LegalFunction` or `LegalAsset`. `LegalFunction` is a legal role which is played by `LegalSubject`s. A `LegalSubject` is a particular type of social agent such as an organization or institution and is part of a legal context. In contrast, the class `LegalAsset` represents non-agentive objects such as the content which is regulated by a flow control rule. Figure B.1b shows different specializations of the DUL classes `Description` and `Situation` that are used in the DnS pattern [120]. CLO also uses this pattern to describe different legal states of affairs. It defines the classes `LegalDescription` and `LegalFact` as direct subclasses of `Description` and `Situation`, respectively, and further specifies them for particular use cases. The individual subclasses are used in the following patterns which are part of the CLOMapping ontology.

The Code of Conduct Pattern of InFO is depicted in Figure 3.11. It models an organizational code of conduct which motivates a flow control policy. The CLOMapping ontology extends this pattern with additional details and aligns it to the classes of CLO. The extended pattern is based on DUL's DnS pattern and depicted in Figure B.2. It defines a code of conduct as a collection of basic principles. Each principle corresponds to a behavioral rule which is adopted by the organization that is also the creator of the code of conduct. Thus, the organization acts as a `CodeOfConductCreator` and imposes

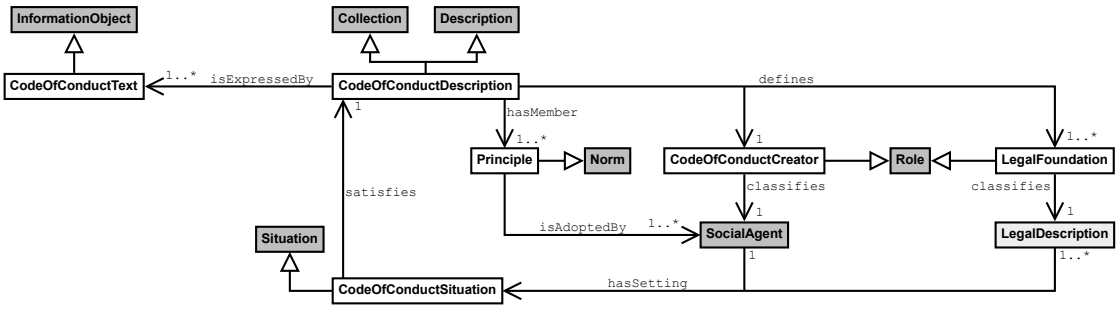
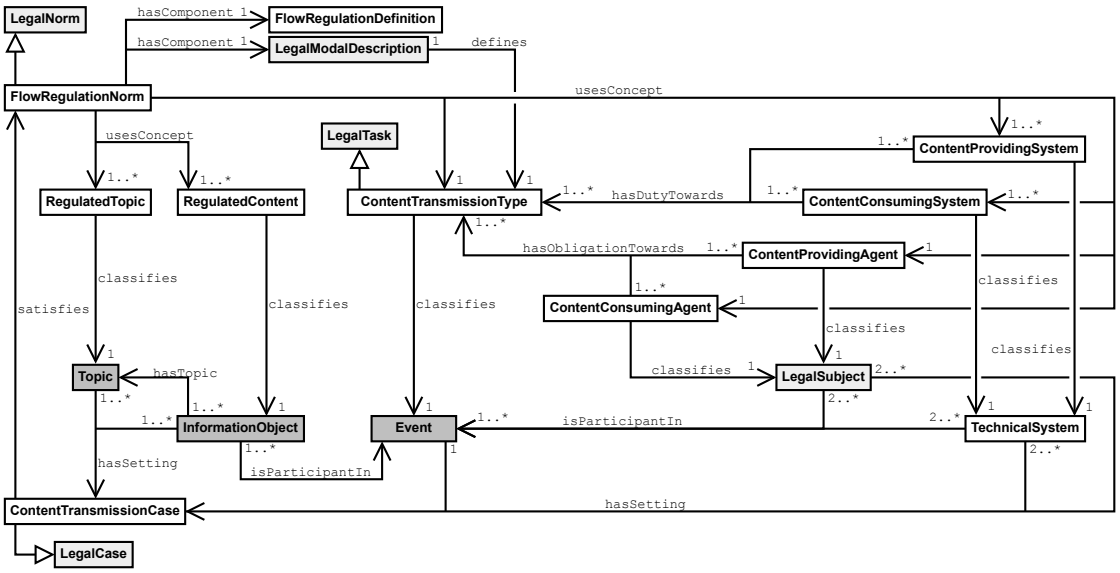


Figure B.2.: Integrating the Code of Conduct Pattern into the Core Legal Ontology.

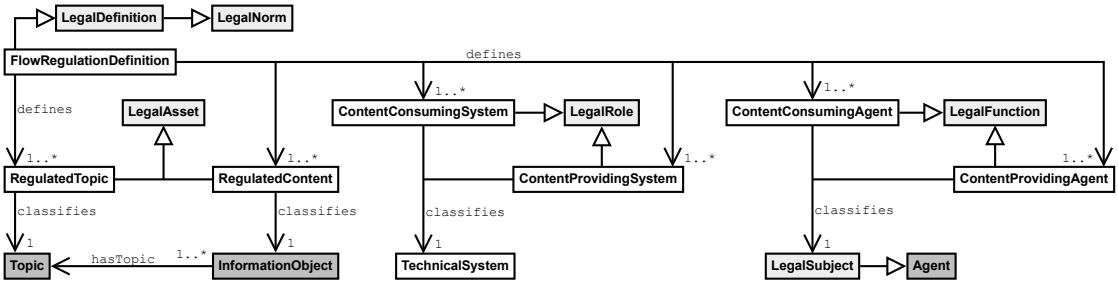
the obligation to follow its code of conduct on itself. The legal foundation of a code of conduct is a `LegalDescription` such as a legal norm. A code of conduct is expressed by a `CodeOfConductText` which is a particular type of `InformationObject`. According to DUL's information realization pattern as depicted in Figure 3.2g, the code's textual representation is realized as an `InformationRealization`. The conceptualization of a code of conduct is represented by the pattern's main class `CodeOfConductDescription` which is satisfied by a corresponding `CodeOfConductSituation`.

The Flow Regulation Norm Pattern of InFO is depicted in Figure 3.12 and describes the legal background of a flow control policy. Figure B.3a shows the pattern after having aligned it to CLO. The modified pattern extends the Norm↔Case pattern of CLO which distinguishes between legal norms and their application in a legal case. Similar to the DnS pattern of DUL, a norm defines the different roles and parameters which are relevant for a legal setting whereas a legal case relates these roles and parameters to particular entities such as `Agents` or non-agentive objects. The modified Flow Regulation Norm Pattern defines `FlowRegulationNorm` as a subclass of `LegalNorm` and replaces the classes of DUL with more specific classes of CLO. In particular, `EventType` is replaced by its subclass `LegalTask` and `LegalSubject` is used instead of `SocialAgent`. A flow regulation norm models a particular communication flow to be either legal or illegal by using corresponding subclasses of `LegalModalDescription`. The class `FlowRegulationNorm` only represents a conceptualization of a legal norm which regulates the legality of the communication flow. A particular communication setting involving real-world entities



(a) Integrating the Flow Regulation Norm Pattern into the Core Legal Ontology.

Figure B.3.: Flow Regulation Norm Pattern and Regulation Definition Pattern. The Regulation Definition Pattern defines concepts which are used by the Flow Regulation Norm Pattern.



(b) Regulation Definition Pattern.

Figure B.3.: Flow Regulation Norm Pattern and Regulation Definition Pattern. *Continued from previous page.*

is modeled by the class `ContentTransmissionCase` which is directly related to all such entities. Such a case can be checked for its legality by comparing it with a corresponding flow regulation norm. This allows it to decide whether nor not the execution of this particular case (and thus executing the communication flow) is actually legal or illegal. CLO generally distinguishes between the usage of legal concepts and their definition. The Flow Regulation Norm Pattern uses different legal concepts which are defined in the Regulation Definition Pattern depicted in Figure B.3b. The Regulation Definition Pattern essentially provides a vocabulary for the Flow Regulation Norm Pattern.

CLO defines a law as a collection of several legal norms which regulate the same social context [123]. An example law is the German Criminal Code [62] which consists of different legal norms regulating the handling of crimes. One of these legal norms is §86 which prohibits the distribution of neo-Nazi material. The Legislation Pattern of InFO models the process of altering a legal norm and supports different modifications such as passing the norm, updating the norm, or suspending it. Figure B.4 shows how this pattern is integrated into CLO. Its main class `LegislationNorm` is associated with all concepts relevant for passing or modifying a legal norm. Its component `Legislation-`

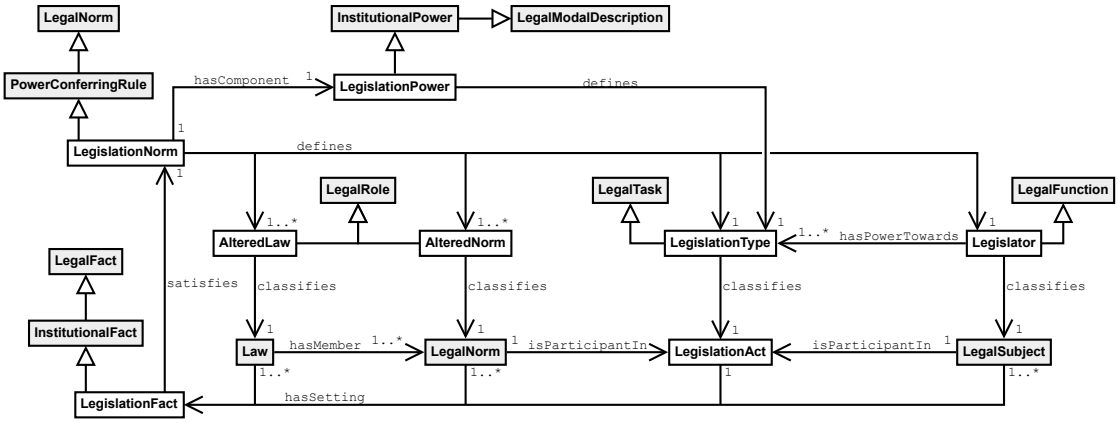


Figure B.4.: Integrating the Legislation Pattern into the Core Legal Ontology.

Power states that the legal subject acting as the legislator has the right to alter the law. The actual situation in which the law is altered is expressed as a LegislationFact. This class is related to all real-world entities involved in the altering process.

B.1.2. Integrating the Legal Knowledge Interchange Format

The Legal Knowledge Interchange Format (LKIF) [146, 147] is a legal core ontology which provides a foundation for translating legal knowledge between different sources. In contrast to CLO, LKIF is not based on an upper ontology such as DUL and defines all classes and properties itself. These classes include different types of roles, legal norms, and legal documents. Figure B.5 depicts a fragment of LKIF's role hierarchy. LKIF distinguishes between Legal_Roles, Social_Roles, and Functions. Legal_Roles and Social_Roles are played by Agents which are either natural persons or organizations. All other entities are not represented as agents in LKIF. Functions refer to physical objects which are not agents and may be used to classify technical systems. Legal_Roles are primarily used to classify agents which are active in a legal context such as judges or lawyers whereas Social_Roles refer to social activities. Examples of social roles are Organization_Roles which can be used for describing a position *within* an organization. However, the complete organization can have both legal roles and social roles. The different roles and legal norms of LKIF are used in the following patterns which align the organizational patterns and legal patterns of InFO to the legal ontology. The alignment is implemented in the LKIFMapping ontology. As LKIF does not define any particular patterns itself, the alignment focuses on introducing subclass relationships between classes of the InFO patterns and LKIF classes. In the following, classes marked in dark gray refer to classes of DUL, light gray identifies classes of LKIF, and white classes correspond to classes which are either part of InFO or the LKIFMapping ontology.

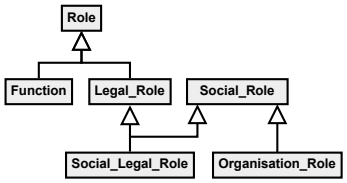


Figure B.5.: Hierarchy of different legal roles in LKIF.

The Code of Conduct Pattern of InFO introduces the class CodeOfConductDescription which defines the context of all relevant classes for describing an organizational code of conduct. Figure B.6 shows how the pattern is aligned to LKIF. CodeOfConductDescription is modeled as a subclass of LKIF's Norm which is used for allowing or disallowing different things. Norm is a subclass of Mental_Object which is similar to SocialObject in DUL. The actual contents of a code of conduct are encoded as a CodeOfConductText which is a particular type of legal document. As a code of conduct is a self-imposed set of rules of a social entity such as an organization, its creator is represented as a Social_Role rather than a Legal_Role. The properties for relating a CodeOfConductDescription to its different entities are modeled according to the

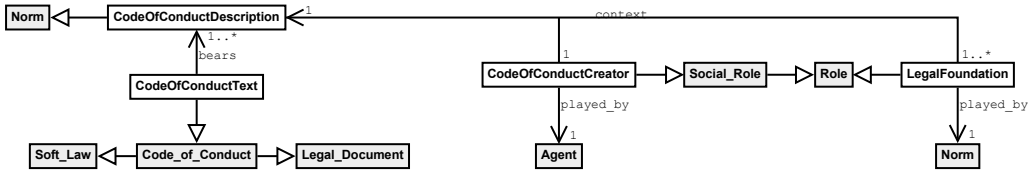


Figure B.6.: Integrating the Code of Conduct Pattern into LKIF.

properties provided in LKIF. In particular, the property `defines` of DUL is replaced by `context` and `bears` corresponds to the inverse property of `isExpressedBy`.

The Flow Regulation Norm pattern of InFO defines the legal state of a particular communication flow. Figure B.7 depicts the pattern after having aligned it to LKIF. A `FlowRegulationNorm` is a specialization of `Norm` which models how the transmission of content of a specific topic is regulated by law. The norm either allows or disallows the transmission by using corresponding subclasses of `Normatively_Qualified`. Transmitting regulated content is represented by two different actions which cover the content's consumption by a consumer and its offer by a provider. Each action requires an `Agent` such as a natural person and an additional technical system such as a client computer in order to be executed. The distinction between providing content and consuming content and splitting both steps of the content's transmission is necessary as LKIF restricts an action to a single agent. However, a transmission usually involves two agents which are the sender and the receiver. As LKIF does not support such complex actions, both steps of the transmission are split into individual parts. In each part, the agent plays a `Legal_Role` whereas the technical system provides a function to fulfill the action. For reasons of brevity, Figure B.7 only depicts the consuming action and its related entities. The providing action is modeled similarly.

The Legislation Pattern of InFO describes how a legal norm is created, altered, or suspended. Figure B.8 shows how the pattern is integrated into LKIF. The main class of the pattern is `LegislationNorm` which covers all entities for describing how a legal norm is modified. The norm essentially allows a legislator to perform the modification and is

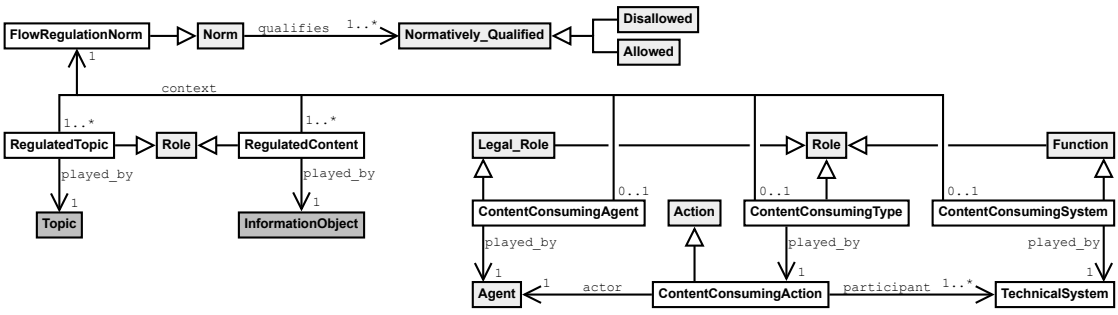


Figure B.7.: Integrating the Flow Regulation Norm Pattern into LKIF. For reasons of brevity, the figure only shows how the pattern models the consumption of content. Providing the content is modeled similar to its consumption.

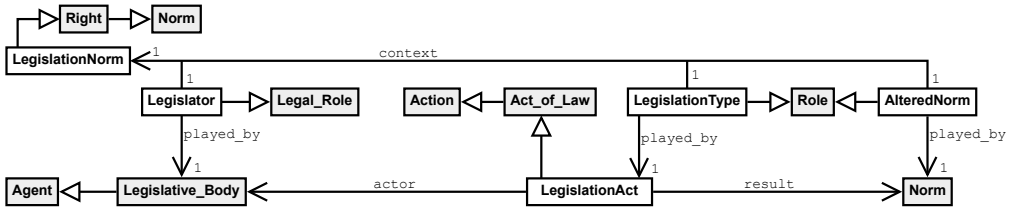


Figure B.8.: Integrating the Legislation Pattern into LKIF.

therefore modeled as a subclass of `Right`. LKIF provides the class `Legislative_Body` as a particular type of `Agent` which is specifically designed to represent legislators. Similar to the original pattern, a `LegislationAct` corresponds to an action in which both the legislative body and the altered norm participate. The legislative body conducts the action which results in the altered norm.

B.2. Integrating Content Labeling Schemes into InFO

This section describes how different content labeling schemes are integrated into the InFO policy language. The discussed content labeling schemes are the RTA label, age-de.xml [264], and PICS [183]. The integration mainly focuses on the classification of regulated Internet content by using the labeling schemes. It does not map every feature of a labeling scheme into InFO as most of the additional features can already be expressed with InFO's vocabulary. The three content labeling schemes are integrated using the mapping ontologies RTAMapping, AgeDeXmlMapping, and PICSMapping as depicted in Figure 3.4. The integration of the different labeling schemes is conducted according to the general process outlined in Section 3.3.8.

B.2.1. Integrating the RTA Label

The Restricted to Adults (RTA) label[1] is a simple label for classifying web content as adult content. The label is embedded into the classified web page directly or included in the HTTP response when requesting the web page from a server. Figure B.9 shows how the RTA label is integrated into InFO by using the RTAMapping mapping ontology. As the RTA label is only a single string which is identical for all web content, the mapping ontology defines a single instance of the class `Topic` from the Ontopic core ontology[2]. This instance is identified as `rtat-1` and is expressed by a word which contains the string representation of the RTA label. Web content is classified as adult content by associating it with the topic `rtat-1` using the topic pattern of the Ontopic ontology as depicted in Figure 3.3. Figure B.10 shows an example usage of the topic `rtat-1` for classifying the two web sites `wst-1` and `wst-2` which are taken from the example regulation of Section 3.4.4. Both web sites are also classified with the topic `pt-1` which

[1] `http://www.rtalabel.org`, last accessed: 01/21/16
[2] `http://ontologydesignpatterns.org/ont/dul/ontopic.owl`, last accessed: 01/21/16

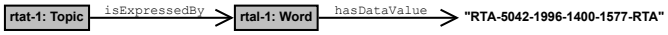


Figure B.9.: RTAMapping ontology which defines an individual `Topic`.

Figure B.10.: Example usage of the RTAMapping ontology.

represents pornographic content. As most pornographic content is also adult content, both topics overlap with each other.

B.2.2. Integrating age-de.xml

age-de.xml [264] uses a single XML file for classifying all web content of a web site with corresponding age categories. In contrast to the RTA label, age-de.xml is not restricted to particular age categories and supports the definition of arbitrary categories. An age category indicates the minimum age that a person must have when requesting the related web content. The XML file can contain different categories for each part of a web site such as individual web pages, subdomains, or directories. Age categories defined with age-de.xml are used by child protection software to regulate the access to web content based on the age of the content consumer, i. e., the human Internet user. If the Internet user is younger than the minimum age of the requested web content, access to the content is blocked. In addition to the default blocking behavior, age-de.xml also allows to redirect users of a particular age to another web page. This allows it to present underage users and adult users different web pages even if their request is identical.

age-de.xml is integrated into InFO by using the AgeDeXmlMapping mapping ontology. The ontology defines the class `AgeLabel` as a subclass of `Topic` for defining arbitrary age labels. It also contains five different instances of this class which represent the age categories of the German rating system organization *Voluntary Self Regulation of the*

Table B.1.: Example age categories of the AgeDeXmlMapping ontology. The categories are based on the German FSK rating system and are instances of `AgeLabel`.

Age category	Instance of `AgeLabel`
FSK ab 0	`fsk-ab-0`
FSK ab 6	`fsk-ab-6`
FSK ab 12	`fsk-ab-12`
FSK ab 16	`fsk-ab-16`
FSK ab 18	`fsk-ab-18`

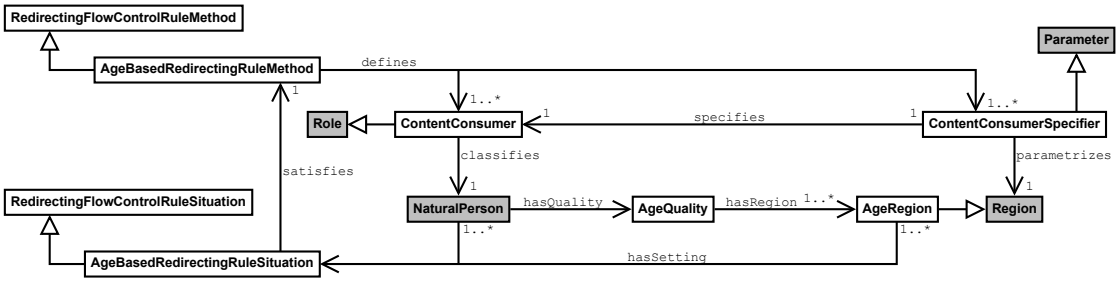


Figure B.11.: Age-Based Redirecting Flow Control Rule Pattern.

Movie Industry (Freiwillige Selbstkontrolle der Filmwirtschaft; FSK)[3]. The individual instances are depicted in Table B.1. The redirection ability of age-de.xml is mapped to the Age-Based Redirecting Flow Control Rule Pattern depicted in Figure B.11. The pattern extends the Redirecting Flow Control Rule Pattern of InFO described in Section 3.3.3 with additional attributes of the regulated communication. The attributes are represented by the classes `ContentConsumer` and `ContentConsumerSpecifier` which describe human consumers and their personal features such as their age. The pattern can be used to create redirecting rules for Internet users of different ages.

B.2.3. Integrating PICS

PICS [183] allows to describe digital resources with more complex labels than the RTA label or age-de.xml. A PICS label associates a digital resource with several ratings, each of which contains an arbitrary number of attribute-value pairs. Each attribute covers a particular feature of the labeled resource and the value defines the feature's intensity. PICS labels do not describe digital resources by associating them with static categories of similar resources like the RTA label or age-de.xml. Instead, resources are described by specifying their individual characteristics. This is similar to the DUL's qualities and quality region pattern [119] as described in Section 3.3.1. Thus, the integration of PICS into InFO is also based on this design pattern. It is implemented by using the PICSMapping mapping ontology which is depicted in Figure B.12. The ontology defines the two classes `PICSRatingAttribute` and `PICSRatingValue` as subclasses of `Quality` and `Region`, respectively. They represent a single PICS rating and associate it with the labeled resource. A separate entity for PICS labels is not defined as such a label is basically a collection of several ratings and does not provide any additional functions for describing a digital resource.

[3] `http://www.spio.de/`, last accessed: 01/21/16

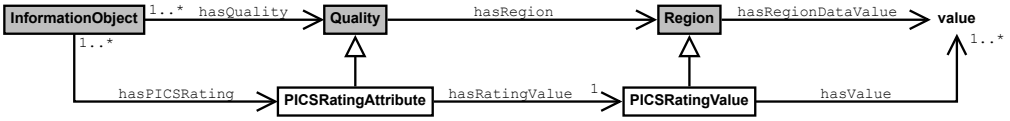


Figure B.12.: PICSMapping ontology which extends the qualities and quality region pattern of DUL.

Appendix C.

Signature Ontology

This appendix describes a lightweight Signature Ontology which can be used with the graph signing framework Siggi presented in Chapter 4. As described in Section 4.3, the Signature Ontology is used by the assembly function α_N to create a signature graph S. The signature graph contains the signature value of a set of signed Named Graphs and all information about how to verify this value. This includes the names of all sub-functions of the graph signing function σ_N involved in the signature's creation, the public key for verifying the signature value, the identifiers of all signed graphs, and the actual signature value. The design of the Signature Ontology is based on the XML signature standard [20] which defines an XML schema for describing signatures of XML documents. The Signature Ontology provides three different ontology design patterns [121] for describing this information. These patterns are the *Signature Pattern*, the *Graph Signing Method Pattern*, and the *Certificate Pattern*. The Signature Pattern covers the basic information about a signature value, the Graph Signing Method Pattern covers all functions that were used during its creation, and the Certificate Pattern describes the public key certificate used for verifying the signature value. The Signature Ontology is implemented using the Web Ontology Language (OWL) [301] and axiomatized using Description Logics [16]. In order to be compliant with the Information Flow Control Ontology described in Chapter 3, the patterns of the Signature Ontology are aligned to the upper ontology DOLCE+DnS Ultralite (DUL) [119]. However, the Signature Ontology can also be used separately from DUL.

C.1. Signature Pattern

The *Signature Pattern* is the main pattern of the Signature Ontology and depicted in Figure C.1. It covers the basic information about how to verify a signature value. The Signature Pattern links a signature value to the identifiers of the signed Named Graphs, the graph signing function σ_N that was used for creating the signature value, and the public key certificate which is used for its validation. The signature is validated by applying the specified graph signing function σ_N to the signature value, the set of Named Graphs, and on the public key k_p contained in the public key certificate. The detailed process of this verification is described in Section 4.3.9 as part of the graph signing formalization. Both the signature value and the public key certificate

are modeled as subclasses of DUL's class `InformationObject` as they correspond to abstract information entities. The Signature Pattern does not provide a specific class for the signed Named Graphs. Instead, the OWL class `Thing` is used directly.

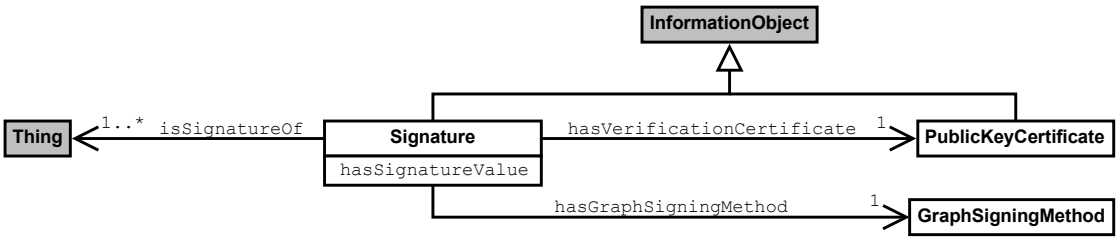


Figure C.1.: Signature Pattern. Gray entities are either taken from the upper ontology DOLCE+DnS Ultralite (DUL) [119] or from the OWL core vocabulary and white entities are part of the Signature Ontology.

C.2. Graph Signing Method Pattern

The Graph Signing Method Pattern describes the particular sub-functions of the graph signing function σ_N and is depicted in Figure C.2. The sub-functions of the graph signing function σ_N are the canonicalization function for graphs κ_N, the serialization function ν_N, the basic hash function λ and the hash function for graphs λ_N, the combining function for graphs ϱ_N, and the signature function φ. The Graph Signing Method Pattern defines all these functions as a component of the graph signing function σ_N by using DUL's componency pattern [119]. All functions are modeled as subclasses of DUL's class `Method`. Thus, their names end with `Method` instead of `Function`.

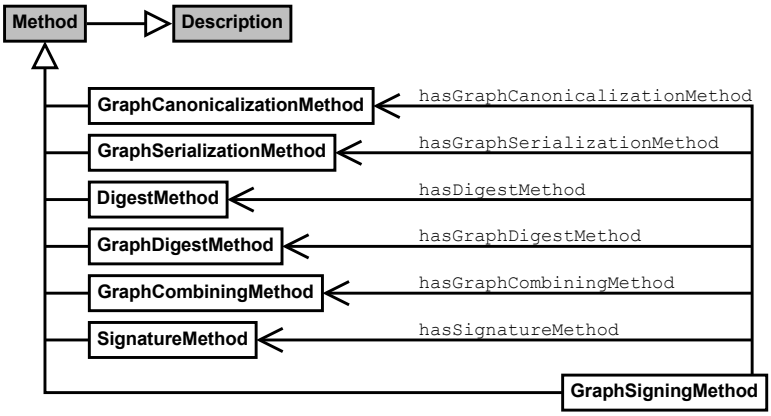


Figure C.2.: Graph Signing Method Pattern. The pattern models the sub-functions of the graph signing process as components of the graph signing function σ_N.

The Signature Ontology also defines several individuals, each of which represents a particular implementation of a sub-function in the graph singing process. Each individual

is identified by a unique URI and corresponds to exactly one algorithm. The individuals correspond to the related work for signing graphs discussed in Section 4.1 and are listed in Table C.1. The hash algorithm SHA-2 [218] is mapped to multiple individuals, each of which covers a specific bit length of the resulting hash value. For example, SHA-2 with an output length of 224 bits corresponds to `dm-sha224`.

Table C.1.: Identifiers used in the Signature Ontology for the sub-functions of the graph signing function σ_N. The identifiers correspond to the related work for signing graph data discussed in Section 4.1.

Function	Example	Identifier
Canonicalization Function κ_N	Carroll [72]	`gcm-carroll-2003`
	Fisteus et al. [110]	`gcm-fisteus-2010`
	Hogan [148]	`gcm-hogan-2015`
	Kuhn and Dumontier [184]	`gcm-kuhn-2014`
	Sayers and Karp [261]	`gcm-sayers-2004`
Serialization Function ν_N	N-Triples [25]	`gsm-n-triples`
	Turtle [27]	`gsm-turtle`
	N3 [34]	`gsm-n3`
	TriG [38]	`gsm-trig`
	TriX [75]	`gsm-trix`
	RDF/XML [26]	`gsm-rdf-xml`
	OWL/XML [210]	`gsm-owl-xml`
Basic Hash Function λ	MD5 [250]	`dm-md5`
	SHA-224 [218]	`dm-sha224`
	SHA-256 [218]	`dm-sha256`
	SHA-384 [218]	`dm-sha384`
	SHA-512 [218]	`dm-sha512`
Hash Function for Graphs λ_N	Melnik [200]	`gdm-melnik-2001`
	Carroll [72]	`gdm-fisteus-2010`
	Fisteus et al. [110]	`gdm-carroll-2003`
	Sayers & Karp [261]	`gdm-sayers-2004`
Combining Function for Graphs ϱ_N	sort and concatenate	`cm-sort-concat`
Signature Function φ	ElGamal [97]	`sm-elgamal`
	RSA [251]	`sm-rsa`
	DSA [214]	`sm-dsa`

C.3. Certificate Pattern

The Certificate Pattern is depicted in Figure C.3 and describes the details of the public key certificate used for verifying a signature of a set of graphs. The pattern introduces the

class PublicKeyCertificate as subclass of DUL's InformationObject. The class PublicKeyCertificate serves as a superclass for all different types of public key certificates. This includes the class PGPCertificates for expressing PGP certificates [320] and the class X509Certificates for expressing certificates of the X.509 format [82]. All public key certificates are owned by an Agent known as the certificate's subject. The subject of a certificate can be identified by a hierarchically structured distinguished name [317]. Other attributes of a PublicKeyCertificate depend on the certificate's type. For example, X.509 certificates are issued by certification authorities which are also identified by their distinguished names. A X.509 certificate can be uniquely identified by using its serial number and the distinguished name of its issuer.

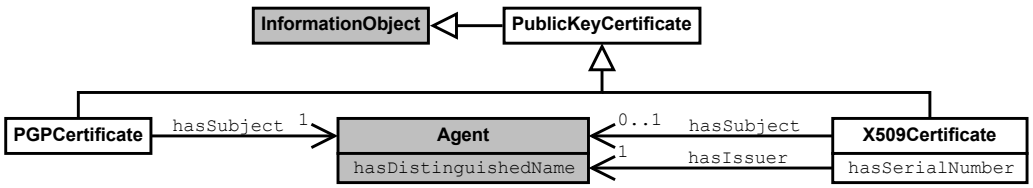


Figure C.3.: Certificate Pattern.

Appendix D.

Extended Log Format Ontology

This appendix describes a lightweight OWL ontology of the Extended Log Format [138]. The format is designed for monitoring HTTP communication between two different computer systems such as client computers, proxy servers, and web servers. It can describe all technical communication details of a single HTTP transmission. The format has an open design which can be extended with additional fields in order to further describe a recorded transmission. It can be used by web servers to monitor incoming client requests or applied by proxy servers to log all HTTP communication between several client computers and their contacted servers. A recorded HTTP transmission is represented by a log entry which describes the transmission's technical communication details. A log entry consists of several fields, each of which covers a particular detail such as the IP address of the client computer or the requested URL. Several log entries are combined into log files which are usually stored at the system that conducts the logging. Example web servers which support the Extended Log Format include the Microsoft web server IIS[1] and Oracle WebLogic [211].

The Extended Log Format Ontology provides an OWL ontology of the Extended Log Format by mapping each of the format's fields to ontological classes and properties. The ontology is used in the example application of T-Store, an approach for searching in encrypted RDF graphs which is covered in Chapter 5. The details of this application are described in Section 5.8. The ontology has a lightweight design which is primarily based on data properties. This allows the ontology to be used for creating simple log files without an unnecessary overhead. The ontology defines the three classes `LogFile`, `LogEntry`, and `SummaryEntry` as well as several data properties for specifying their attributes. The names of the data properties derive from the specification of the Extended Log Format [138]. Figure D.1 depicts the relations between all three classes and their most important data properties. In addition to the depicted classes and properties, the ontology can also be extended with additional properties. The class `LogFile` represents a log file which consists of several log entries. A log file can be further described by subproperties of `logFileProperty`. Examples of such properties are `logFileDate` and `logFileSoftware`. `logFileSoftware` describes the software which was used for creating the log file and `logFileDate` specifies the date and time when the

[1] See `https://msdn.microsoft.com/en-us/library/ms525807\%28v=vs.90\%29.aspx`, last accessed: 01/21/16

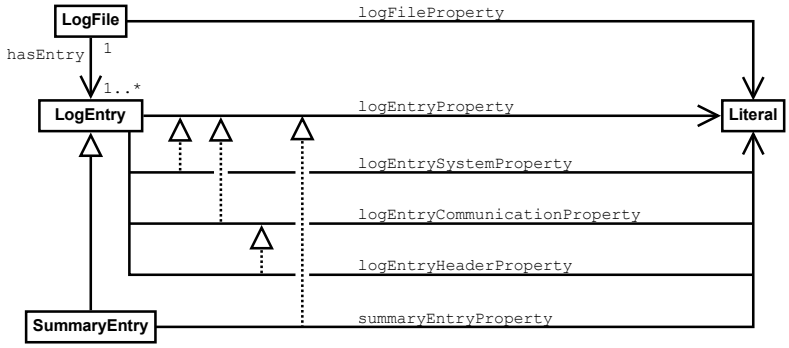


Figure D.1.: Fragment of the Extended Log Format Ontology. The ontology maps the vocabulary of the Extended Log Format [138] to different ontological classes and data properties. For reasons of brevity, the figure only shows the most important data properties. Each of the depicted data properties is further refined by more specific subproperties. Dashed arrows indicate subproperty relationships between data properties.

log file was created. For reasons of brevity, the specific subproperties are not depicted in Figure D.1.

`LogEntry` represents a particular log entry of a `LogFile` and is further described by subproperties of `logEntryProperty`. Example properties include `date` and `time` for storing the date and time of the entry, respectively. More specific properties are summarized by using the subproperties `logEntrySystemProperty`, `logEntryCommunicationProperty`, and `logEntryHeaderProperty`. `logEntrySystemProperty` covers all data properties for describing a single system involved in the recorded HTTP transmission. Examples of such properties are `cIp` for storing the IP address of the client computer and `sDns` for describing the domain name of the web server. The prefix `c` indicates that a property refers to the client whereas `s` refers to the server. `logEntryCommunicationProperty` covers data properties which are not restricted to a single system and affect both communicating systems at once. An example for such a property is `csMethod` which describes the HTTP method the client system has used when sending its request to the web server. The prefix `cs` indicates the direction of the transmitted message and states that the client system (`c`) is the sender while the web server (`s`) is the receiver. Example values of this data property are `GET` and `POST` [109]. `logEntryHeaderProperty` further specifies a `logEntryCommunicationProperty` and covers all data properties for describing a particular header field in an HTTP message header. Examples of such properties are `scSetCookie` and `csCookie` which are used for setting a cookie at the client system and sending it back to the server, respectively. Again, the prefixes `sc` and `cs` indicate the direction of the sent message. A `SummaryEntry` does not describe an individual HTTP transmission. Instead, it summarizes recurring events which are described with subproperties of `summaryEntryProperty`. An example property is `count` which defines how many times the described event occurred. Additional properties of `SummaryEntry` can be used for specifying the time frame of the counted events.

Bibliography

[1] Cristina L. Abad and Rafael I. Bonilla. An analysis on the schemes for detecting and preventing ARP cache poisoning attacks. In *27th International Conference on Distributed Computing Systems Workshops (ICDCSW '07)*, pages 60–60, 2007.

[2] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. Enabling advanced and context-dependent access control in RDF stores. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 1–14. Springer, Berlin Heidelberg, 2007.

[3] Hal Abelson, Ben Adida, Mike Linksvayer, and Nathan Yergler. ccREL: The Creative Commons rights expression language. W3C member submission, Creative Commons, 2008. `http://www.w3.org/Submission/ccREL/` (last accessed: 01/21/16).

[4] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*, chapter 6: Certificates and Certification, pages 69–87. In [8], 2nd edition, 2003.

[5] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*, chapter 3: The Concept of an Infrastructure, pages 21–36. In [8], 2nd edition, 2003.

[6] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*, chapter 9: Trust Models, pages 131–149. In [8], 2nd edition, 2003.

[7] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*, chapter 5: PKI-Enabled Services, pages 49–67. In [8], 2nd edition, 2003.

[8] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley, 2nd edition, 2003.

[9] Georgy Adelson-Velsky and Evgenii Mikhailovich Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[10] Waseem Akhtar, Jacek Kopecký, Thomas Krennwallner, and Axel Polleres. XSPARQL: Traveling between the XML and RDF Worlds – and avoiding the XSLT pilgrimage. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 432–447. Springer, Berlin Heidelberg, 2008.

[11] Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist*, chapter 10: Counting and Sets in OWL, pages 213–245. Morgan Kaufmann, 2011.

[12] Anne Anderson. A comparison of two privacy policy languages: EPAL and XACML. Technical report, Sun Microsystems Laboratories, Mountain View, CA, USA, 2005.

[13] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. Foundations of RDF databases. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 158–204. Springer, Berlin Heidelberg, 2009.

[14] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. In *Proceedings of the Workshop on Usage Analysis and the Web of Data (USEWOD'11)*, 2011.

[15] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise privacy authorization language (EPAL 1.2). W3C member submission, World Wide Web Consortium (W3C), 2003. `http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/` (last accessed: 01/21/16).

[16] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[17] Shane Balfe, Eimear Gallery, Chris J. Mitchell, and Kenneth G. Paterson. Challenges for trusted computing. *IEEE Security & Privacy*, 6(6):60–66, 2008.

[18] Alexander Balke, Felix Gorbulski, Artur Schens, and Erwin Schens. ϕ. Projektpraktikumsausarbeitung, Universität Koblenz-Landau, 2014.

[19] Jack M. Balkin, Beth Simone Noveck, and Kermit Roosevelt. Filtering the Internet: A best practices model. In Jens Waltermann and Marcel Machill, editors, *Protecting Our Children on the Internet: Towards a New Culture of Responsibility*. Bertelsmann Foundation Publishers, 2000.

[20] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia, and Ed Simon. XML signature syntax and processing. W3C recommendation, World Wide Web Consortium (W3C), 2008. `http://www.w3.org/TR/xmldsig-core/` (last accessed: 01/21/16).

[21] Cataldo Basile, Antonio Lioy, Salvatore Scozzi, and Marco Vallini. Ontology-based policy translation. *Computational Intelligence in Security for Information Systems*, 63:117–126, 2009.

[22] Richard Baskerville. What design science is not. *European Journal of Information Systems*, 17(5):441–443, 2008.

[23] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[24] Stefan Becker, Benjamin Hück, Katharina Naujokat, Arne Fritjof Schmeiser, and Andreas Kasten. ODRL 2.0 revisited. In *11th International Workshop for Technical, Economical, and Legal Aspects of Business Models for Virtual Goods*, 2013.

[25] Dave Beckett. N-Triples. W3C RDF core WG internal working draft, World Wide Web Consortium (W3C), 2001. `http://www.w3.org/2001/sw/RDFCore/ntriples/` (last accessed: 01/21/16).

[26] Dave Beckett. RDF/XML syntax specification. W3C recommendation, World Wide Web Consortium (W3C), 2004. `http://www.w3.org/TR/rdf-syntax-grammar/` (last accessed: 01/21/16).

[27] David Beckett and Tim Berners-Lee. Turtle. W3C team submission, World Wide Web Consortium (W3C), 2011. `http://www.w3.org/TeamSubmission/turtle/` (last accessed: 01/21/16).

[28] Mark Bedner and Tobias Ackermann. Schutzziele der IT-Sicherheit. *Datenschutz und Datensicherheit (DuD)*, 34(5):323–328, 2010.

[29] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *Advances in Cryptology (EUROCRYPT'97)*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192, Berlin Heidelberg, 1997. Springer.

[30] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.

[31] Tim Berners-Lee. Semantic Web. Presentation at XML2000, 2000. `http://www.w3.org/2000/Talks/1206-xml2k-tbl/` (last accessed: 01/21/16).

[32] Tim Berners-Lee. WWW past & future. Presentation at the Royal Society, 2003. `http://www.w3.org/2003/Talks/0922-rsoc-tbl/` (last accessed: 01/21/16).

[33] Tim Berners-Lee. Artificial intelligence and the Semantic Web. Keynote at the AAAI2006, 2006. `https://www.w3.org/2006/Talks/0718-aaai-tbl/Overview.html` (last accessed: 01/21/16).

[34] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. W3C team submission, World Wide Web Consortium (W3C), 2011. `http://www.w3.org/TeamSubmission/n3/` (last accessed: 01/21/16).

[35] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific american*, 284(5):28–37, 2001.

[36] Étienne Bézout. *Théorie générale des équations algébriques*. Ph.-D. Pierres, 1779.

[37] Matt Bishop. *Introduction to computer security*, chapter 1: An Overview of Computer Security, pages 1–25. Addison Wesley, 2006.

[38] Christian Bizer and Richard Cyganiak. RDF 1.1 TriG. W3C recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/trig/` (last accessed: 01/21/16).

[39] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data – the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[40] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[41] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM (CACM)*, 13(7):422–426, 1970.

[42] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology (EUROCRYPT'04)*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, Berlin Heidelberg, 2004.

[43] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, Berlin Heidelberg, 2011.

[44] Stefano Borgo and Claudio Masolo. Foundational choices in DOLCE. In Staab and Studer [283], chapter 17, pages 361–381.

[45] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 121–132, New York, NY, USA, 2013. ACM.

[46] Luc Bouganim, François Dang Ngoc, and Philippe Pucheral. Client-based access control management for XML documents. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, volume 30, pages 84–95. VLDB Endowment, 2004.

[47] Steve Bratt. Semantic Web, and other technologies to watch. Presentation at the INCOSE International Workshop, 2007. `http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/` (last accessed: 01/21/16).

[48] Tim Bray. An HTTP status code to report legal obstacles. Internet-draft, Network Working Group, 2015. Work in progress. `https://tools.ietf.org/html/draft-ietf-httpbis-legally-restricted-status-04` (last accessed: 01/21/16).

[49] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, World Wide Web Consortium (W3C), 2006. `http://www.w3.org/TR/xml11` (last accessed: 01/21/16).

[50] Dan Brickley and Libby Miller. FOAF vocabulary specification 0.99, 2014. `http://xmlns.com/foaf/spec/` (last accessed: 01/21/16).

[51] Dan Brickley and Ralph R. Swick. PICS rating vocabularies in XML/RDF. W3C note, World Wide Web Consortium (W3C), 2000. `http://www.w3.org/2000/02/rdf-pics/` (last accessed: 01/21/16).

[52] Richard Brinkman, Jeroen Doumen, and Willem Jonker. Using secret sharing for searching in encrypted data. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, volume 3178 of *Lecture Notes in Computer Science*, pages 18–27, Berlin Heidelberg, 2004. Springer.

[53] Richard Brinkman, Ling Feng, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. Efficient tree search in encrypted data. In *Proceedings of the 2nd International Workshop on Security In Information Systems (WOSIS'04)*, pages 126–135. Taylor & Francis, 2004.

[54] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In Ian Horrocks and James Hendler, editors, *Proceedings of 1st International Semantic Web Conference (ISWC'02)*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, Berlin Heidelberg, 2002.

[55] Mark R. Brown and Andrew C. Greenberg. On formally undecidable propositions of law: Legal indeterminacy and the implications of metamathematics. *Hastings Law Journal*, 43:1439, 1992.

[56] Johannes Buchmann. *Einführung in die Kryptographie*, chapter 12. Kryptografische Hashfunktionen, pages 191–202. Springer, 2008.

[57] Johannes Buchmann. *Einführung in die Kryptographie*, chapter 9.3: Das RSA-Verfahren, pages 137–148. Springer, 4th edition, 2008.

[58] Michel Buffa and Catherine Faron-Zucker. Ontology-based access rights management. In *Advances in Knowledge Discovery and Management*, volume 398 of *Studies in Computational Intelligence*, pages 49–61. Springer, Berlin Heidelberg, 2012.

[59] Bundesamt für Sicherheit in der Informationstechnik (BSI), editor. *IT-Grundschutz-Kataloge: Standardwerk zur Informationssicherheit.* Bundesanzeiger Verlag, 2014.

[60] Bundesrepublik Deutschland. Die Gesetzgebung des Bundes. In *Grundgesetz für die Bundesrepublik Deutschland*, chapter VII. Bundesrepublik Deutschland, 1949. `http://www.gesetze-im-internet.de/gg/` (last accessed: 01/21/16).

[61] Bundesrepublik Deutschland. §184 StGB: Verbreitung pornographischer Schriften, 1975. `www.gesetze-im-internet.de/stgb/__184.html` (last accessed: 01/21/16).

[62] Bundesrepublik Deutschland. §86 StGB: Verbreiten von Propagandamitteln verfassungswidriger Organisationen, 1975. `http://www.gesetze-im-internet.de/stgb/__86.html` (last accessed: 01/21/16).

[63] Bundesrepublik Deutschland. § 202c Vorbereiten des Ausspähens und Abfangens von Daten, 2007. `http://www.gesetze-im-internet.de/stgb/__202c.html` (last accessed: 01/21/16).

[64] Bundesrepublik Deutschland. Gesetz zur Bekämpfung der Kinderpornographie in Kommunikationsnetzen. *Bundesgesetzblatt*, (6):78–80, 2010. `http://www2.bgbl.de/Xaver/start.xav?startbk=Bundesanzeiger_BGBl&start=//*[@attr_id=%27bgbl110s0078.pdf%27]` (last accessed: 01/21/16).

[65] Ian Burnett, Rik van de Walle, Keith Hill, Jan Bormans, and Fernando Pereira. MPEG-21: Goals and achievements. *IEEE MultiMedia*, 10(4):60–70, 2003.

[66] Julian Busch. *A Brief Guide to CCC: China Compulsory Certification.* CreateSpace Independent Publishing Platform, 2013.

[67] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, chapter 1: Patterns, pages 1–24. Wiley, 1996.

[68] Jianneng Cao, Fang-Yu Rao, Mehmet Kuzu, Elisa Bertino, and Murat Kantarcioglu. Efficient tree pattern queries on encrypted XML documents. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops (EDBT '13)*, pages 111–120, New York, NY, USA, 2013. ACM.

[69] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *ICDCS'11*, pages 393–402. IEEE, 2011.

[70] Bogdan Carbunar and Radu Sion. Joining privately on outsourced data. In *Secure Data Management*, pages 70–86. Springer, 2010.

[71] Gavin Carothers. RDF 1.1 N-Quads. W3C recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/n-quads/` (last accessed: 01/21/16).

[72] Jeremy J. Carroll. Signing RDF graphs. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proceedings of 2nd International Semantic Web Conference (ISWC'03)*, volume 2870, pages 369–384, Berlin Heidelberg, 2003. Springer.

[73] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web (WWW'05)*, WWW '05, pages 613–622, New York, NY, USA, 2005. ACM.

[74] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web recommendations. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters (WWW Alt. '04)*, pages 74–83, New York, NY, USA, 2004. ACM.

[75] Jeremy J. Carroll and Patrick Stickler. RDF triples in XML. Technical Report HPL-2003-268, HP Laboratories, 2004.

[76] Claude Castelluccia, Einar Mykletun, and Gene Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. In *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'05)*, pages 109–117, 2005.

[77] Stefania Cavallar, Bruce Dodson, Arjen K. K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Jöel Marchand, François Morain, Alec Muffett, Chris and Craig Putnam, and Paul Zimmermann. Factorization of a 512-bit RSA modulus. In Bart Preneel, editor, *Advances in Cryptology (EUROCRYPT'00)*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Berlin Heidelberg, 2000.

[78] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT'10*, pages 577–594. Springer, 2010.

[79] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the great firewall of China. In George Danezis and Philippe Golle, editors, *Privacy*

Enhancing Technologies (PET'06), number 4258 in LNCS, pages 20–35. Springer, 2006.

[80] Common Criteria. Common Criteria for information technology security evaluation – part 1: Introduction and general model. CCMB-2012-09-001, 2012.

[81] Alissa Cooper. A survey of query log privacy-enhancing techniques from a policy perspective. *ACM Transactions on the Web (TWEB)*, 2(4):19:1–19:27, 2008.

[82] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and Tim Polk. Internet X.509 public key infrastructure. Request for Comments 5280, Network Working Group, 2008. `https://tools.ietf.org/html/rfc5280` (last accessed: 01/21/16).

[83] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 8.4: Bucket sort, pages 174–177. The MIT Press, 2nd edition, 2001.

[84] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 31.2: Greatest common divisor, pages 933–938. The MIT, 3rd edition, 2009.

[85] Frédéric Cuppens, Nora Cuppens-Boulahia, Thierry Sans, and Alexandre Miège. A formal approach to specify and deploy a network security policy. *Formal Aspects in Security and Trust*, 173:203–218, 2005.

[86] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories, 2005.

[87] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/rdf11-concepts/` (last accessed: 01/21/16).

[88] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In Morris Sloman, Emil C. Lupu, and Jorge Lobo, editors, *Proceedings of the Workshop on Policies for Distributed Systems and Networks (POLICY '01)*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38, Berlin Heidelberg, 2001. Springer.

[89] Ronald Deibert, John Palfrey, Rafal Rohozinski, and Jonathan Zittrain, editors. *Access Controlled: The Shaping of Power, Rights, and Rule in Cyberspace*. The MIT Press, 2010.

[90] Department of Health and Human Services. Health insurance reform: Security standards; final rule. *Federal Register*, 68(34):8334–8381, 2003. `http://www.hhs.gov/sites/default/files/ocr/privacy/hipaa/administrative/securityrule/securityrulepdf.pdf` (last accessed: 01/21/16).

[91] Deutsche Telekom AG. Our code of conduct: The way we work. Booklet, 2012. `http://www.telekom.com/code-of-conduct-en` (last accessed: 01/21/16).

[92] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. Request for Comments 5246, Network Working Group, 2008. `http://tools.ietf.org/html/rfc5246` (last accessed: 01/21/16).

[93] Digital Library Federation. Metadata encoding and transmission standard: Primer and reference manual. Technical report, Digital Library Federation, 2010. `http://www.loc.gov/standards/mets/METSPrimerRevised.pdf` (last accessed: 01/21/16).

[94] Robert H. Dolin, Liora Alschuler, Sandy Boyer, Calvin Beebe, Fred M. Behlen, Paul V. Biron, and Amnon Shabo Shvo. HL7 clinical document architecture, release 2. *Journal of the American Medical Informatics Association*, 13(1):30–39, 2006.

[95] Maximillian Dornseif. Government mandated blocking of foreign web content. In Jan von Knop, Wilhelm Haverkamp, and Eike Jessen, editors, *Security, E-Learning, E-Services: Proceedings of the 17. DFN-Arbeitstagung über Kommunikationsnetze*, Lecture Notes in Informatics, pages 617–648, Düsseldorf, 2004.

[96] Claudia Eckert. *IT-Sicherheit: Konzepte – Verfahren – Protokolle*, chapter 1.2: Schutzziele, pages 6–14. Oldenbourg Verlag, 2009.

[97] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Berlin Heidelberg, 1985.

[98] Yuval Elovici, Ronen Waisenberg, Erez Shmueli, and Ehud Gudes. A structure preserving database encryption scheme. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, volume 3178 of *Lecture Notes in Computer Science*, pages 28–40. Springer, Berlin Heidelberg, 2004.

[99] Orri Erling and Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In Roberto de Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[100] European Parliament and Council. Directive on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the European Union*, L 281(23/11/1995):31–50, 1995. `http://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:31995L0046` (last accessed: 01/21/16).

[101] European Parliament and Council. Directive on the protection of individuals with regard to the processing of personal data and on the free movement

of such data. *Official Journal of the European Union*, L 201(2002/07/31):37–47, 2002. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32002L0058:EN:HTML` (last accessed: 01/21/16).

[102] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*, chapter 4: Classification of Ontology Matching Techniques, pages 73–84. Springer, 2013.

[103] Christopher Evans, Clive D. W. Feather, Alex Hopmann, Martin Presler-Marshall, and Paul Resnick. PICSRules 1.1. W3C recommendation, World Wide Web Consortium (W3C), 1997. `http://www.w3.org/TR/REC-PICSRules/` (last accessed: 01/21/16).

[104] Sergei Evdokimov and Oliver Günther. Encryption techniques for secure database outsourcing. In Joachim Biskup and Javier López, editors, *Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 327–342. Springer, Berlin Heidelberg, 2007.

[105] Gunther Eysenbach. What is e-health? *Journal of Medical Internet Research*, 3(2), 2001.

[106] Göran Falkman, Marie Gustafsson, Mats Jontell, and Olof Torgersson. SOMWeb: A Semantic Web-based system for supporting collaboration of distributed medical communities of practice. *Journal of medical Internet research*, 10(3), 2008.

[107] Ling Feng and Willem Jonker. Efficient processing of secured XML metadata. In Robert Meersman and Zahir Tari, editors, *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 704–717, Berlin Heidelberg, 2003. Springer.

[108] José Luis Fernández-Alemán, Inmaculada Carrión Se nor, Pedro Ángel Oliver Lozoya, and Ambrosio Toval. Security and privacy in electronic health records: A systematic literature review. *Journal of Biomedical Informatics*, 46:541–562, 2013.

[109] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Request for Comments 2616, Network Working Group, 1999. `http://tools.ietf.org/html/rfc2616` (last accessed: 01/21/16).

[110] Jesus Arias Fisteus, Norberto Fernández García, Luis Sánchez Fernández, and Carlos Delgado Kloos. Hashing and canonicalizing Notation3 graphs. *Journal of Computer and System Sciences*, 76(7):663–685, 2010.

[111] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007:15:1–15:15, January 2007.

[112] Ned Freed and Nathaniel S. Borenstein. Multipurpose Internet mail extensions (MIME) part one: Format of internet message bodies. Request for Comments

2045, Network Working Group, 1996. `https://tools.ietf.org/html/rfc2045` (last accessed: 01/21/16).

[113] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The secure sockets layer (SSL) protocol version 3.0. Request for Comments 6101, Internet Engineering Task Force (IETF), 2011. `https://tools.ietf.org/html/rfc6101` (last accessed: 01/21/16).

[114] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. Request for Comments 1519, Network Working Group, 1993. `http://tools.ietf.org/html/rfc1519` (last accessed: 01/21/16).

[115] Jun Furukawa and Toshiyuki Isshiki. Controlled joining on encrypted relational database. In *Pairing 2012*, pages 46–64. Springer, 2013.

[116] Eimear Gallery and Chris J. Mitchell. Trusted computing: Security and applications. *Cryptologia*, 33(3):217–245, 2009.

[117] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*, chapter 1: Introduction, pages 1–31. Addison-Wesley, 1995.

[118] Aldo Gangemi. Design patterns for legal ontology construction. In Pompeu Casanovas, Maria Angela Biasiotti, Enrico Francesconi, and Maria Teresa Sagri, editors, *Proceedings of the 2nd Workshop on Legal Ontologies and Artificial Intelligence Techniques (LOAIT'07)*, volume 321, pages 65–85. CEUR Workshop Proceedings, 2007.

[119] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening ontologies with DOLCE. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 223–233. Springer, 2002.

[120] Aldo Gangemi and Peter Mika. Understanding the Semantic Web through descriptions and situations. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 689–706. Springer, 11 2003.

[121] Aldo Gangemi and Valentina Presutti. Ontology design patterns. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 221–243. Springer, Berlin Heidelberg, 2009.

[122] Aldo Gangemi and Valentina Presutti. Ontology design patterns. In Staab and Studer [283], chapter 11, pages 221–243.

[123] Aldo Gangemi, Maria-Teresa Sagri, and Daniela Tiscornia. A constructive framework for legal ontologies. In V. Richard Benjamins, Pompeu Casanovas, Joost Breuker, and Aldo Gangemi, editors, *Law and the Semantic Web*, volume 3369, pages 97–124. Springer, 2005.

[124] Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. W3C recommendation, World Wide Web Consortium (W3C), 2012. `http://www.w3.org/TR/xmlschema11-1/` (last accessed: 01/21/16).

[125] Dave Garets and Mike Davis. Electronic medical records vs. electronic health records: Yes, there is a difference. White paper, HIMSS Analytics, LLC, 2006.

[126] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM (CACM)*, 53(3):97–105, 2010.

[127] Aurona J. Gerber, Andries Barnard, and Alta J. van der Merwe. Towards a Semantic Web layered architecture. In *International Conference on Software Engineering*, 2007.

[128] Aurona J. Gerber, Alta J. van der Merwe, and Andries Barnard. A functional Semantic Web architecture. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 273–287. Springer, Berlin Heidelberg, 2008.

[129] Global Network Initiative. Principles on freedom of expression and privacy. Technical report, Global Network Initiative, 2011. `http://www.globalnetworkinitiative.org/principles/index.php` (last accessed: 01/21/16).

[130] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[131] Thomas Gottron, Malte Knauf, and Ansgar Scherp. Analysis of schema structures in the linked open data graph based on unique subject URIs, pay-level domains, and vocabulary usage. *Distributed and Parallel Databases*, pages 1–39, 2014.

[132] Jacob Grimm and Wilhelm Grimm, editors. *Deutsche Sagen*, volume 1. Nicolaische Buchhandlung, Berlin, 1816.

[133] Rüdiger Grimm, Daniela Simić-Draws, Katharina Bräunlich, Andreas Kasten, and Anastasia Meletiadou. Referenzmodell für ein Vorgehen bei der IT-Sicherheitsanalyse. *Informatik-Spektrum*, pages 1–19, 2014.

[134] Alessio Gugliotta, Liliana Cabral, John Domingue, Vito Roberto, Mary Rowlatt, and Rob Davies. A Semantic Web service-based architecture for the interoperability of e-Government services. *Web Information Systems Modeling Workshop (WISM'05)*, pages 21–30, 2005.

[135] Tracy D. Gunter and Nicolas P. Terry. The emergence of national electronic health record architectures in the United States and Australia. *Journal of Medical Internet Research*, 7(1), 2005.

[136] Sebastian Haas, Sven Wohlgemuth, Isao Echizen, Noboru Sonehara, and Günter Müller. Aspects of privacy for electronic health records. *International journal of medical informatics*, 80(2):e26–e31, 2011.

[137] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD'02)*, pages 216–227, New York, NY, USA, 2002. ACM.

[138] Phillip M. Hallam-Baker and Brian Behlendorf. Extended Log File Format. W3C Working Draft WD-logfile-960323, World Wide Web Consortium (W3C), 1996. `http://www.w3.org/TR/WD-logfile.html` (last accessed: 01/21/16).

[139] Volker Hammer, Ulrich Pordesch, and Alexander Roßnagel. KORA – eine Methode zur Konkretisierung rechtlicher Anforderungen zu technischen Gestaltungsvorschlägen für Informations-und Kommunikationssysteme. *Infotech/I+ G*, 5, 1993.

[140] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, World Wide Web Consortium (W3C), 2013. `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/` (last accessed: 01/21/16).

[141] Andreas Harth, Maciej Janik, and Steffen Staab. Semantic Web architecture. In John Domingue, Dieter Fensel, and James A. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 43–75. Springer, Berlin Heidelberg, 2011.

[142] Patrick J. Hayes and Peter F. Patel-Schneider. RDF 1.1 semantics. W3C recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/rdf11-mt/` (last accessed: 01/21/16).

[143] Martin Hepp, Frank Leymann, John Domingue, Alexander Wahler, and Dieter Fensel. Semantic business process management: A vision towards using Semantic Web services for business process management. In *International Conference on e-Business Engineering (ICEBE'05)*, pages 535–540. IEEE, 2005.

[144] Hessisches Kultusministerium. Verordnung über die Aufsicht über Schülerinnen und Schüler (Aufsichtsverordnung – AufsVO) vom 11. Dezember 2013. In *Amtsblatt des Hessischen Kultusministeriums*. A. Bernecker Verlag GmbH, 2013.

[145] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[146] Rinke Hoekstra, Joost Breuker, Marcello di Bello, and Alexander Boer. The LKIF core ontology of basic legal concepts. In Pompeu Casanovas, Maria Angela Biasiotti, Enrico Francesconi, and Maria Teresa Sagri, editors, *Proceedings of the 2nd Workshop on Legal Ontologies and Artificial Intelligence Techniques (LOAIT'07)*, volume 321, pages 43–63. CEUR Workshop Proceedings, 2007.

[147] Rinke Hoekstra, Joost Breuker, Marcello di Bello, and Alexander Boer. LKIF core: Principled ontology development for the legal domain. *Law, Ontologies and the Semantic Web: Channelling the Legal Information Flood*, 188:21, 2009.

[148] Aidan Hogan. Skolemising blank nodes while preserving isomorphism. In *Proceedings of the 24th International Conference on World Wide Web (WWW'15)*, pages 430–440. International World Wide Web Conferences Steering Committee, 2015.

[149] Aidan Hogan, Jürgen Umbrich, Andreas Harth, Richard Cyganiak, Axel Polleres, and Stefan Decker. An empirical survey of linked data conformance. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14:14–44, 2012.

[150] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, volume 30, pages 720–731. VLDB Endowment, 2004.

[151] Adam Horowitz. The constitutionality of the children's internet protection act. *St. Thomas Law Review*, 13(1):425–444, 2000.

[152] Ian Horrocks. Ontologies and the Semantic Web. *Communications of the ACM (CACM)*, 51(22):58–67, 2008.

[153] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic Web architecture: Stack or two towers? In François Fages and Sylvain Soliman, editors, *Principles and practice of Semantic Web reasoning*, volume 3703 of *Lecture Notes in Computer Science*, pages 37–41. Springer, Berlin Heidelberg, 2005.

[154] Evelyn Johanna Sophia Hovenga. Importance of achieving semantic interoperability for national health information systems. *Texto & Contexto - Enfermagem*, 17(1):158 –167, 2008.

[155] Renato Iannella. ODRL version 2.1 XML encoding. Specification, W3C ODRL Community Group, 2015. `https://www.w3.org/community/odrl/xml/2.1/` (last accessed: 01/21/16).

[156] Renato Iannella, Susanne Guth, Daniel Pähler, and Andreas Kasten. ODRL version 2.1 core model. Specification, W3C ODRL Community Group, 2015. `https://www.w3.org/community/odrl/model/2.1/` (last accessed: 01/21/16).

[157] Renato Iannella, Michael Steidl, and Susanne Guth-Orlowski. ODRL version 2.1 common vocabulary. Specification, W3C ODRL Community Group, 2015. `https://www.w3.org/community/odrl/vocab/2.1/` (last accessed: 01/21/16).

[158] Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other non-random properties for step-reduced SHA-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 276–293. Springer, Berlin Heidelberg, 2009.

[159] International Organization for Standardization (ISO). Information technology – open systems interconnection – basic reference model: The basic model. ISO/IEC Standard 7498-1:1994, 1994.

[160] International Organization for Standardization (ISO). Information technology – database languages – SQL – part 1: Framework (SQL/framework). ISO/IEC Standard 9075-1:2011, 2011.

[161] International Organization for Standardization (ISO). Codes for the representation of names of countries and their subdivisions – part 1: Country codes. ISO Standard 3166-1:2013, 2013.

[162] International Press Telecommunications Council (IPTC). RightsML: Specification of a profile and vocabulary for the communication of usage rights in ODRL 2.0. IPTC Standards, 2013. `http://www.iptc.org/std/RightsML/1.1/RightsML_1.1EP2-spec_1.pdf` (last accessed: 01/21/16).

[163] Ashraf Mohammed Iqbal. An OWL-DL ontology for the HL7 reference information model. In Bessam Abdulrazak, Sylvain Giroux, Bruno Bouchard, Hélène Pigot, and Mounir Mokhtari, editors, *Toward Useful Services for Elderly and People with Disabilities*, volume 6719 of *Lecture Notes in Computer Science*, pages 168–175. Springer, Berlin Heidelberg, 2011.

[164] Amit Jain and Csilla Farkas. Secure resource description framework: an access control model. In *Proceedings of the eleventh ACM symposium on Access control models and technologies (SACMAT '06)*, pages 121–129, New York, NY, USA, 2006. ACM.

[165] Ravi Chandra Jammalamadaka and Sharad Mehrotra. Querying encrypted XML documents. In *10th International Database Engineering and Applications Symposium (IDEAS '06)*, pages 129–136. IEEE, 2006.

[166] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 63–74. IEEE, 2003.

[167] Murat Kantarcıoğlu and Chris Clifton. Security issues in querying encrypted data. In *Data and Applications Security XIX*, pages 325–337. Springer, 2005.

[168] Andreas Kasten and Ansgar Scherp. Iterative signing of RDF(S) graphs, Named Graphs, and OWL graphs: Formalization and application. Arbeitsberichte aus dem Fachbereich Informatik 03/2013, Universität Koblenz-Landau, 2013.

[169] Andreas Kasten and Ansgar Scherp. Towards a configurable framework for iterative signing of distributed graph data. In *Proceedings of the Workshop on Society, Privacy and the Semantic Web – Policy and Technology (PrivOn'13)*, volume 1121, Sydney, 2013. CEUR Workshop Proceedings.

[170] Andreas Kasten and Ansgar Scherp. Towards a framework for iteratively signing graph data. In *Proceedings of the seventh international conference on Knowledge capture (K-CAP'13)*, pages 141–142. ACM, 2013.

[171] Andreas Kasten and Ansgar Scherp. Towards search on encrypted graph data. In *Proceedings of the Workshop on Society, Privacy and the Semantic Web – Policy and Technology (PrivOn'13)*, volume 1121, Sydney, 2013. CEUR Workshop Proceedings.

[172] Andreas Kasten and Ansgar Scherp. Ontology-based information flow control of network-level Internet communication. *International Journal of Semantic Computing*, 9(1):1–45, 2015.

[173] Andreas Kasten, Ansgar Scherp, and Peter Schauß. A framework for iterative signing of graph data on the web. In Valentina Presutti, Claudia d'Amato, Fabien Gandon, Mathieu d'Aquin, Steffen Staab, and Anna Tordai, editors, *The Semantic Web: Trends and Challenges (ESWC'14)*, volume 8465 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014.

[174] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*, chapter 7.2: Primes, Factoring, and RSA, pages 261–274. In [176], 2008.

[175] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*, chapter 12.3: RSA Signatures, pages 426–429. In [176], 2008.

[176] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, Boca Raton, London, New York, 2008.

[177] Patrick Kierkegaard. Electronic health record: Wiring europe's healthcare. *Computer Law & Security Review*, 27(5):503–515, 2011.

[178] Michael Kifer and Harold Boley. RIF overview (second edition). W3C working group note, World Wide Web Consortium (W3C), 2013. `http://www.w3.org/TR/rif-overview/` (last accessed: 01/21/16).

[179] Kingdom of Saudi Arabia. Anti-Cyber Crime Law, 2007. `http://www.citc.gov.sa/English/RulesandSystems/CITCSyste/Documents/LA_004_%20E_%20Anti-Cyber%20Crime%20Law.pdf` (last accessed: 01/21/16).

[180] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *Advances in Cryptology (CRYPTO'10)*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Berlin Heidelberg, 2010.

[181] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3 of *The Art of Computer Programming*, chapter 6.2.3: Balanced Trees, pages 458–481. Addison-Wesley, 2nd edition, 1998.

[182] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*, chapter 8: Non-Functional Requirements, pages 187–213. Wiley, 2003.

[183] Tim Krauskopf, Jim Miller, Paul Resnick, and Win Treese. PICS label distribution label syntax and communication protocols. W3C Recommendation, World Wide Web Consortium (W3C), 1996. `http://www.w3.org/TR/REC-PICS-labels` (last accessed: 01/21/16).

[184] Tobias Kuhn and Michel Dumontier. Trusty URIs: Verifiable, immutable, and permanent digital artifacts for linked data. In Valentina Presutti, Claudia d'Amato, Fabien Gandon, Mathieu d'Aquin, Steffen Staab, and Anna Tordai, editors, *The Semantic Web: Trends and Challenges (ESWC'14)*, volume 8465 of *Lecture Notes in Computer Science*, pages 395–410. Springer International Publishing, Berlin Heidelberg, 2014.

[185] Jin Kyu Lee, Shambhu J. Upadhyaya, H. Raghav Rao, and Raj Sharman. Secure knowledge management and the Semantic Web. *Communications of the ACM (CACM)*, 48(12):48–54, 2005.

[186] Han Li, Jie Zhang, and Rathindra Sarathy. Understanding compliance with Internet use policy from the perspective of rational choice theory. *Decision Support Systems*, 48(4):635–645, 2010.

[187] Jun Li, Youdong Ding, Yunyu Shi, and Jianfei Zhang. Building a large annotation ontology for movie video retrieval. *Journal of Digital Content Technology and its Applications*, 4(5):74–81, 2010.

[188] Jun Li and Edward R. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In Sushil Jajodia and Duminda Wijesekera, editors, *Data and Applications Security XIX*, volume 3654 of *Lecture Notes in Computer Science*, pages 69–83. Springer, Berlin Heidelberg, 2005.

[189] Yaping Li and Minghua Chen. Privacy preserving joins. In *24th International Conference on Data Engineering (ICDE'08)*, pages 1352–1354. IEEE, 2008.

[190] Ping Lin and K. Selçuk Candan. Hiding traversal of tree structured data from untrusted data stores. In *Proceedings of the 2nd International Workshop on Security In Information Systems (WOSIS'04)*, pages 314–323, 2004.

[191] Ping Lin and K. Selçuk Candan. Secure and privacy preserving outsourcing of tree structured data. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, volume 3178 of *Lecture Notes in Computer Science*, pages 1–17. Springer, Berlin Heidelberg, 2004.

[192] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[193] Clifford Lynch. Canonicalization: A fundamental tool to facilitate preservation and management of digital information. *D-Lib Magazine*, 5(9), 1999. `http://www.dlib.org/dlib/september99/09lynch.html` (last accessed: 01/21/16).

[194] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[195] Brian McBride. RDF schema 1.1. W3C recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/rdf-schema/` (last accessed: 01/21/16).

[196] Declan McCullagh. Google excluding controversial sites. Online news article, 2002. `http://news.cnet.com/2100-1023-963132.html` (last accessed: 01/21/16).

[197] Kevin S. McCurley. The discrete logarithm problem. In *Proceedings of Symposia in Applied Mathematics*, volume 42, pages 49–74. American Mathematical Society, 1990.

[198] John F. McGuire. When speech is heard around the world: Internet content regluation in the United States and Germany. *New York University Law Review*, 74:750–792, 1999.

[199] Mo McRoberts and Víctor Rodríguez-Doncel. ODRL version 2.1 ontology. Specification, W3C ODRL Community Group, 2015. `http://www.w3.org/ns/odrl/2/ODRL21` (last accessed: 01/21/16).

[200] Sergey Melnik. RDF API draft, 2001. `http://infolab.stanford.edu/~melnik/rdf/` (last accessed: 01/21/16).

[201] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: New attacks on reduced SHA-256. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology (EUROCRYPT'13)*, volume 7881 of *Lecture Notes in Computer Science*, pages 262–278. Springer, Berlin Heidelberg, 2013.

[202] Paul Mockapetris. Domain names – implementation and specification. Request for Comments 1035, Network Working Group, 1987. `http://tools.ietf.org/html/rfc1035` (last accessed: 01/21/16).

[203] Knud Möller, Michael Hausenblas, Richard Cyganiak, and Siegfried Handschuh. Learning from linked open data usage: Patterns & metrics. In *Web Science Conference 2010*, 2010.

[204] Dominik Mosen. Erweiterung eines Nameservers zur policy-basierten Internetregulierung. Bachelorarbeit, Universität Koblenz-Landau, 2012.

[205] Tim Moses. eXtensible Access Control Markup Language (XACML) version 2.0. OASIS standard, OASIS, 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf` (last accessed: 01/21/16).

[206] Tim Moses. Privacy policy profile of XACML v2.0. OASIS standard, OASIS, 2005. http://docs.oasis-open.org/xacml/2.0/PRIVACY-PROFILE/access_control-xacml-2.0-privacy_profile-spec-os.pdf (last accessed: 01/21/16).

[207] Mark Mosley, Michael H. Brackett, Susan Earley, and Deborah Henderson. *The DAMA Guide to the Data Management Body of Knowledge (DAMA-DMBOK Guide)*, chapter 1: Introduction, pages 1–16. In [209], 2010.

[208] Mark Mosley, Michael H. Brackett, Susan Earley, and Deborah Henderson. *The DAMA Guide to the Data Management Body of Knowledge (DAMA-DMBOK Guide)*, chapter 7: Data Security Management, pages 151–169. In [209], 2010.

[209] Mark Mosley, Michael H. Brackett, Susan Earley, and Deborah Henderson. *The DAMA Guide to the Data Management Body of Knowledge (DAMA-DMBOK Guide)*. DAMA International, 2010.

[210] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 web ontology language XML serialization. W3C recommendation, World Wide Web Consortium (W3C), 2009. http://www.w3.org/TR/owl2-xml-serialization/ (last accessed: 01/21/16).

[211] Jon Mountjoy and Avinash Chugh. *WebLogic: The Definitive Guide*, chapter 3.3: HTTP Access Logs, pages 72–74. O'Reilly Media, 2004.

[212] Steven J. Murdoch and Ross Anderson. Tools and technology of Internet filtering. In Ronald Deibert, John Palfrey, Rafal Rohozinski, and Jonathan Zittrain, editors, *Access Denied: The Practice and Policy of Global Internet Filtering*, chapter 3, pages 57–72. The MIT Press, 2008.

[213] National Institute of Standards and Technology (NIST). Advanced encryption standard. Federal Information Processing Standards Publication 197, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (last accessed: 01/21/16).

[214] National Institute of Standards and Technology (NIST). Digital signature standard (DSS). Federal Information Processing Standards Publication 186-3, 2009. http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf (last accessed: 01/21/16).

[215] National Institute of Standards and Technology (NIST). Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Special Publication 800-131A, 2011. http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf (last accessed: 01/21/16).

[216] National Institute of Standards and Technology (NIST). Recommendation for cryptographic key generation. Special Publication 800-133, 2012. http://dx.doi.org/10.6028/NIST.SP.800-133 (last accessed: 01/21/16).

[217] National Institute of Standards and Technology (NIST). Recommendation for key management pt. 1. Special Publication 800-57, 2012. `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf` (last accessed: 01/21/16).

[218] National Institute of Standards and Technology (NIST). Secure hash standard. Federal Information Processing Standards Publication 180-4, 2012. `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf` (last accessed: 01/21/16).

[219] National Institute of Standards and Technology (NIST). Digital signature standard (DSS). Federal Information Processing Standards Publication 186-4, 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf` (last accessed: 01/21/16).

[220] Eric K. Neumann and Dennis Quan. BioDash: a Semantic Web dashboard for drug development. *Pacific Symposium on Biocomputing*, pages 176–187, 2006.

[221] Jonas Öberg, Stuart Myles, and Lu Ai. ODRL version 2.1 JSON encoding. Specification, W3C ODRL Community Group, 2015. `https://www.w3.org/community/odrl/json/2.1/` (last accessed: 01/21/16).

[222] Daniel Oberle. *Semantic Management of Middleware*, chapter 3: Ontologies, pages 33–53. Semantic Web and Beyond: Computing for Human Experience. Springer US, 2006.

[223] Frank Oemig and Bernd Blobel. An ontology architecture for HL7 v3: Pitfalls and outcomes. In Olaf Dössel and Wolfgang C. Schlegel, editors, *World Congress on Medical Physics and Biomedical Engineering, September 7 - 12, 2009, Munich, Germany*, volume 25/12 of *IFMBE Proceedings*, pages 408–410. Springer, 2009.

[224] Hans Oh, Carlos Rizo, Murray Enkin, and Alejandro Jadad. What is eHealth: A systematic review of published definitions. *World Hospitals and Health Services*, 41(1):32–40, 2005.

[225] Open Networking Foundation. SDN architecture. Technical Report SDN ARCH 1.0 06062014, Open Networking Foundation, 2014. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf` (last accessed: 01/21/16).

[226] OpenNet Initiative. Internet filtering in Saudi Arabia. Research report, OpenNet Initiative, 2009. `http://opennet.net/research/profiles/saudi-arabia` (last accessed: 01/21/16).

[227] OpenNet Initiative. *Access Controlled: The Shaping of Power, Rights, and Rule in Cyberspace*, pages 369–387. In Deibert et al. [89], 2010.

[228] Bhavna Orgun and Joseph Vu. HL7 ontology and mobile agents for interoperability in heterogeneous medical information systems. *Computers in Biology and Medicine*, 36(7–8):817–836, 2006. Special Issue on Medical Ontologies.

[229] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT '02)*, pages 57–64, New York, NY, USA, 2002. ACM.

[230] Peter F. Patel-Schneider and Boris Motik. OWL 2 web ontology language mapping to RDF graphs. W3C recommendation, World Wide Web Consortium (W3C), 2012. `http://www.w3.org/TR/owl2-mapping-to-rdf/` (last accessed: 01/21/16).

[231] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[232] Radia Perlman. An overview of PKI trust models. *Network, IEEE*, 13(6):38–43, 1999.

[233] Francois Picalausa and Stijn Vansummeren. What are real SPARQL queries like? In *Proceedings of the International Workshop on Semantic Web Information Management (SWIM'11)*, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[234] Hoi Ting Poon and Ali Miri. Computation and search over encrypted XML documents. In *IEEE International Congress on Big Data (BigData Congress'15)*, pages 631–634, 2015.

[235] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 85–100, New York, NY, USA, 2011. ACM.

[236] Jon Postel. Internet control message protocol. Request for Comments 792, Network Working Group, 1981. `http://tools.ietf.org/html/rfc792` (last accessed: 01/21/16).

[237] Valentina Presutti and Aldo Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *Proceedings of the 27th International Conference on Conceptual Modeling*. Springer, 2008.

[238] Jan Pries-Heje, Richard Baskerville, and John R. Venable. Strategies for design science research evaluation. *Proceedings of the European Conference on Information Systems (ECIS'08)*, pages 1–12, 2008.

[239] Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Towards fine grained RDF access control. In *19th ACM Symposium on Access Control Models and Technologies (SACMAT '14)*, pages 165–176, New York, NY, USA, 2014. ACM.

[240] Yves Raimond, Samer A. Abdallah, Mark B. Sandler, and Frederick Giasson. The music ontology. In *8th International Conference on Music Information Retrieval (ISMIR'07)*, pages 417–422, 2007.

[241] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter 13.3: External Merge Sort, pages 424–430. In [244], 3rd edition, 2003.

[242] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter 14.4: The Join Operation, pages 452–468. In [244], 3rd edition, 2003.

[243] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter 4.2.4: Joins, pages 107–109. In [244], 3rd edition, 2003.

[244] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2003.

[245] Blake Ramsdell and Sean Turner. Secure/multipurpose Internet mail extensions (S/MIME) version 3.2 message specification. Request for Comments 5751, Internet Engineering Task Force (IETF), 2010. `http://tools.ietf.org/html/rfc5751` (last accessed: 01/21/16).

[246] Alan L. Rector, R. Qamar, and T. Marley. Binding ontologies & coding systems to electronic health records and messages. In *Biomedical Ontology in Action (KR-MED 2006)*, 2006.

[247] Pavan Reddivari, Tim Finin, and Anupam Joshi. Policy-based access control for an RDF store. In *Proceedings of the Policy Management for the Web workshop (PM4W'05)*, volume 120, pages 78–83, 2005.

[248] Paul Resnick and James Miller. PICS: Internet access controls without censorship. *Communications of the ACM (CACM)*, 39(10):87–93, 1996.

[249] Paul J. Resnick, Derek L. Hansen, and Caroline R. Richardson. Calculating error rates for filtering software. *Commununications of the ACM*, 47(9):67–71, 2004.

[250] Ronald Rivest. The MD5 message-digest algorithm. Request for Comments 1321, Network Working Group, 1992. `https://tools.ietf.org/html/rfc1321` (last accessed: 01/21/16).

[251] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commununications of the ACM*, 21(2):120–126, 1978.

[252] Jonathan Robie and Michael Dyck. XML path language (XPath) 3.1. W3C Recommendation, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/xpath-31/` (last accessed: 01/21/16).

[253] Víctor Rodríguez-Doncel, Mari Carmen Suárez-Figueroa, Asunción Gómez-Perez, and María Poveda-Villalón. License linked data resources pattern. In *Proceedings of the 4th International Workshop on Ontology Patterns (WOP'13)*, volume 1188. CEUR Workshop Proceedings, 2013.

[254] Víctor Rodríguez-Doncel, Mari Carmen Suárez-Figueroa, Asunción Gómez-Perez, and María Poveda-Villalón. Licensing patterns for linked data. In *Proceedings of the 4th International Workshop on Ontology Patterns (WOP'13)*, volume 1188. CEUR Workshop Proceedings, 2013.

[255] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, Berlin Heidelberg, 2004.

[256] Daniel L. Rubin, Pattanasak Mongkolwat, Vladimir Kleper, Kaustubh Supekar, and David S. Channin. Medical imaging on the Semantic Web: Annotation and image markup. In *AAAI Spring Symposium: Semantic Scientific Knowledge Integration*, pages 93–98, 2008.

[257] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer, Berlin Heidelberg, 2001.

[258] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks against up to 24-step SHA-2. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology (INDOCRYPT'08)*, volume 5365 of *Lecture Notes in Computer Science*, pages 91–103. Springer, Berlin Heidelberg, 2008.

[259] Tahmineh Sanamrad, Lucas Braun, Donald Kossmann, and Ramarathnam Venkatesan. Randomly partitioned encryption for cloud databases. In Vijay Atluri and Günther Pernul, editors, *Data and Applications Security and Privacy XXVIII*, volume 8566 of *Lecture Notes in Computer Science*, pages 307–323. Springer, Berlin Heidelberg, 2014.

[260] Ravi Sandhu and Jaehong Park. Usage control: A vision for next generation access control. *Computer Network Security*, 2776:17–31, 2003.

[261] Craig Sayers and Alan H. Karp. Computing the digest of an RDF graph. Technical Report HPL-2003-235, HP Laboratories, 2004.

[262] Craig Sayers and Alan H. Karp. RDF graph digest techniques and potential applications. Technical Report HPL-2004-95, HP Laboratories, 2004.

[263] Peter Schauß. Digitales Signieren von RDF-Graphen. Bachelorarbeit, Universität Koblenz-Landau, 2013.

[264] Stefan Schellenberg. label format for age classification in telemedia / websites – age-de.xml. Technical paper, Online Management Kontor, 2010. `http://www.online-management-kontor.de/images/downloads/age-de-xml-label_definition_v3.0g_english1.pdf` (last accessed: 01/21/16).

[265] Ansgar Scherp, Daniel Eißing, and Carsten Saathoff. A method for integrating multimedia metadata standards and metadata formats with the multimedia metadata ontology. *International Journal of Semantic Computing*, 6(1):25–49, 2012.

[266] Ansgar Scherp, Carsten Saathoff, Thomas Franz, and Steffen Staab. Designing core ontologies. *Applied Ontology*, 6(3):177–221, 2011.

[267] Bruce Schneier. *Applied Cryptography*, chapter 2.6: Digital Signatures, pages 34–41. In [272], 1996.

[268] Bruce Schneier. *Applied Cryptography*, chapter 2: Protocol Building Blocks, pages 21–46. In [272], 1996.

[269] Bruce Schneier. *Applied Cryptography*, chapter 8: Key Management, pages 169–187. In [272], 1996.

[270] Bruce Schneier. *Applied Cryptography*, chapter 1.1: Terminology, pages 1–9. In [272], 1996.

[271] Bruce Schneier. *Applied Cryptography*, chapter 19.3: RSA, pages 466–474. In [272], 1996.

[272] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[273] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[274] Michael Schrefl, Katharina Grun, and Jürgen Dorn. SemCrypt – ensuring privacy of electronic documents through semantic-based encrypted query processing. In *21st International Conference on Data Engineering Workshops (ICDEW'05)*, pages 1191–1200. IEEE, 2005.

[275] Guus Schreiber and Yves Raimond. RDF 1.1 primer. W3C working group note, World Wide Web Consortium (W3C), 2014. `http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/` (last accessed: 01/21/16).

[276] Henning Schulzrinne, Hannes Tschofenig, John B. Morris, Jr., Jorge R. Cuellar, James Polk, and Jonathan Rosenberg. Common Policy: A document format for expressing privacy preferences. Request for Comments 4745, Network Working Group, 2007. `http://tools.ietf.org/html/rfc4745` (last accessed: 01/21/16).

[277] Mark M. Seeger. Three years hacker paragraph. *Datenschutz und Datensicherheit*, 34(7):476–478, 2010.

[278] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[279] A. Sheth, S. Agrawal, J. Lathem, N. Oldham, H. Wingate, P. Yadav, and K. Gallagher. Active semantic electronic medical records. In Jorge Cardoso, Martin Hepp, and Miltiadis D. Lytras, editors, *The Semantic Web*, volume 6 of *Semantic Web And Beyond Computing for Human Experience*, pages 123–140. Springer US, 2008.

[280] Keng Siau, Fiona Fui-Hoon Nah, and Limei Teng. Acceptable Internet use policy. *Communications of the ACM (CACM)*, 45(1):75–79, 2002.

[281] Radu Sion. Towards secure data outsourcing. In Michael Gertz and Sushil Jajodia, editors, *Handbook of Database Security*, pages 137–161. Springer US, 2008.

[282] Ian Sommerville. *Software Engineering*, chapter Software Requirements, pages 97–120. Addison Wesley, 6th edition, 2001.

[283] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. Springer, 2009.

[284] Ständige Konferenz der Kultusminister der Länder in der Bundesrepublik Deutschland. Neue Medien und Schule. Beschluss der Kultusministerkonferenz vom 22.02.2001, Sekretariat der ständigen Konferenz der Kultusminister der Länder in der Bundesrepublik Deutschland, 2001.

[285] Paul T. Stanton, Benjamin McKeown, Randal Burns, and Giuseppe Ateniese. Fast-AD: An authenticated directory for billions of objects. *ACM SIGOPS*, 44(1):45–49, 2010.

[286] John Strassner, José Neuman de Souza, Sven van der Meer, Steven Davy, Keara Barrett, Dave Raymer, and Srini Samudrala. The design of a new policy model to support ontology-driven reasoning for autonomic networking. *Journal of Network and Systems Management*, 17(1–2):5–32, 2009.

[287] Laurent Sustek. Hardware security module. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*, pages 254–256. Springer US, 2005.

[288] Andrew Tanenbaum and Maarten van Steen. *Distributed systems*. Pearson Prentice Hall, 2007.

[289] Paul C. Tang, Joan S. Ash, David W. Bates, J. Marc Overhage, and Daniel Z. Sands. Personal health records: definitions, benefits, and strategies for overcoming barriers to adoption. *Journal of the American Medical Informatics Association*, 13(2):121–126, 2006.

[290] Telecommunication Standardization Sector of ITU. Information technology – open systems interconnection – the directory: Public-key and attribute certificate frameworks. ITU-T Recommendation X.509, International Telecommunication Union (ITU), 2000.

[291] Steven J. Templeton and Karl E. Levitt. Detecting spoofed packets. In *Proceedings on the DARPA Information Survivability Conference and Exposition*, volume 1, pages 164–175, 2003.

[292] The W3C SPARQL Working Group. SPARQL 1.1 overview. W3C recommendation, World Wide Web Consortium (W3C), 2013. `http://www.w3.org/TR/sparql11-overview/` (last accessed: 01/21/16).

[293] Yulia A. Timofeeva. Hate speech online: Restricted or protected? Comparison of regulations in the United States and Germany. *Journal of Transnational Law & Policy*, 12:253–286, 2002.

[294] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proceedings of 2nd International Semantic Web Conference (ISWC'03)*, volume 2870 of *Lecture Notes in Computer Science*, pages 419–437. Springer, Berlin Heidelberg, 2003.

[295] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. Signing individual fragments of an RDF graph. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW'05)*, pages 1020–1021, New York, NY, USA, 2005. ACM.

[296] Ozan Ünay and Taflan İ. Gündem. A survey on querying encrypted XML documents for databases as a service. *SIGMOD Record*, 37(1):12–20, 2008.

[297] United States Congress. Health insurance portability and accountability act of 1996 (HIPAA). Public Law 104-191, 110 Statutes at Large 1936, 1996. `http://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm` (last accessed: 01/21/16).

[298] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 93–96, 2003.

[299] Vijay Vaishnavi and William Kuechler. Design science research in information systems, 2004. `http://desrist.org/design-research-in-information-systems/` (last accessed: 01/21/16).

[300] L. Vizenor, B. Smith, and W. Ceusters. Foundation for the electronic health record: An ontological analysis of the HL7's reference information model, 2004.

[301] W3C OWL Working Group. OWL 2 Web Ontology Language document overview. W3C recommendation, World Wide Web Consortium (W3C), 2009. `http://www.w3.org/TR/owl2-overview/` (last accessed: 01/21/16).

[302] W3C OWL Working Group. OWL 2 Web Ontology Language document overview (second edition). W3C recommendation, World Wide Web Consortium (W3C), 2012. `http://www.w3.org/TR/owl2-overview/` (last accessed: 01/21/16).

[303] Jan Walker, Eric Pan, Douglas Johnston, Julia Adler-Milstein, David W. Bates, and Blackford Middleton. The value of health care information exchange and interoperability. *Health Affairs*, 24:W5.10–W5.18, 2005.

[304] Hui Wang and Laks V. S. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, pages 127–138. VLDB Endowment, 2006.

[305] Peng Wang and Chinya V. Ravishankar. Secure and efficient range queries on outsourced databases using Rp-trees. In *IEEE 29th International Conference on Data Engineering (ICDE'13)*, pages 314–325. IEEE, 2013.

[306] Xin Wang. MPEG-21 rights expression language: Enabling interoperable digital rights management. *IEEE Multimedia*, 11(4):84–87, 2004.

[307] Xin Wang, Thomas De Martini, Barney Wragg, M. Paramasivam, and Chris Barlas. The MPEG-21 rights expression language and rights data dictionary. *IEEE Transactions on Multimedia*, 7(3):408–417, 2005.

[308] Xin Wang, Guillermo Lao, Thomas DeMartini, Hari Reddy, Mai Nguyen, and Edgar Valenzuela. XrML – eXtensible rights Markup Language. In *Proceedings of the 2002 ACM workshop on XML security (XMLSEC '02)*, pages 71–79, New York, NY, USA, 2002. ACM.

[309] Zheng-Fei Wang, Jing Dai, Wei Wang, and Bai-Le Shi. Fast query over encrypted character data in database. In Jun Zhang, Ji-Huan He, and Yuxi Fu, editors, *Computational and Information Science*, volume 3314 of *Lecture Notes in Computer Science*, pages 1027–1033. Springer, Berlin Heidelberg, 2005.

[310] Brent R. Waters, Dirk Balfanz, Glenn Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *11th Annual Network and Distributed Security Symposium (NDSS '04)*, 2004.

[311] Rainer Weißenfels. Policy-basierte Internetregulierung durch Router. Diplomarbeit, Universität Koblenz-Landau, 2014.

[312] Khin Than Win. A review of security of electronic health records. *Health Information Management Journal*, 34(1):13–18, 2005.

[313] World Health Organization (WHO). Clinical procedures safety. Best practice safety protocol, WHO, 2012.

[314] Pengtao Xie and Eric P. Xing. CryptGraph: Privacy preserving graph analytics on encrypted graph. *CoRR*, abs/1409.5021, 2014.

[315] Yanjiang Yang, Feng Bao, Xuhua Ding, and Robert H. Deng. Multiuser private queries over encrypted databases. *International Journal of Applied Cryptography*, 1(4):309–319, 2009.

[316] Zhiqiang Yang, Sheng Zhong, and Rebecca N. Wright. Privacy-preserving queries on encrypted data. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer Security (ESORICS'06)*, volume 4189 of *Lecture Notes in Computer Science*, pages 479–495. Springer, Berlin Heidelberg, 2006.

[317] Kurt D. Zeilenga. Lightweight directory access protocol. Request for Comments 4514, Internet Engineering Task Force (IETF), 2006. `https://tools.ietf.org/html/rfc4514` (last accessed: 01/21/16).

[318] Rui Zhang and Ling Liu. Security models and requirements for healthcare application clouds. In *3rd International Conference on Cloud Computing (CLOUD'10)*, pages 268–275. IEEE, 2010.

[319] Yilu Zhou, Edna Reid, Jialun Qin, Hsinchun Chen, and Guanpi Lai. US domestic extremist groups on the web: Link and content analysis. *IEEE Intelligent Systems*, 20(5):44–51, 2005.

[320] Philip R. Zimmermann. *The official PGP user's guide*. The MIT Press, 1995.

[321] Jonathan Zittrain and Benjamin Edelman. Empirical analysis of Internet filtering in China. Research Publication 2003-02, The Berkman Center for Internet & Society, 2003.

Curriculum Vitae

Personal Details

Name	Andreas Kasten
Date of Birth	12/05/1983
Place of Birth	Marburg
Nationality	German

Contact Information

Phone	+49 261 390 549 06
E-Mail	andreas-kasten@web.de
Street	Johannesstraße 24
City	56070 Koblenz

Education

08/90–07/94	Grundschule Dreihausen, Dreihausen
08/94–06/00	Gesamtschule Ebsdorfergrund, Heskem
08/00–06/03	Gymnasium Philippinum, Marburg
06/2003	Higher Education Entrance Qualification

Higher Education

10/03–03/09	Academic studies in Computational Visualistics Unversity of Koblenz-Landau, Koblenz
05/09–11/16	Ph. D. studies in Computer Science Unversity of Koblenz-Landau, Koblenz

Professional Occupation

05/06–08/06	Student assistant at the Institute of Software Engineering Unversity of Koblenz-Landau, Koblenz
09/06–01/07	Student assistant at the Institute of Computer Science Unversity of Koblenz-Landau, Koblenz
11/07–02/08	Student assistant at the Department of Aesthetics and Art History Unversity of Koblenz-Landau, Koblenz
05/09–09/15	Research assistant at the Institute of Information Systems Research Unversity of Koblenz-Landau, Koblenz
since 10/2015	Software engineer at the Hauptabteilung Betriebsorganisation Debeka-Gruppe, Koblenz