A Framework for Iterative Signing of Graph Data on the Web

Andreas Kasten[1], Ansgar Scherp[2], and Peter Schauß[1]

[1] University of Koblenz-Landau, Koblenz, Germany
{andreas.kasten,schauss}@uni-koblenz.de,
[2] Kiel University and Leibniz Information Centre for Economics, Kiel, Germany
asc@informatik.uni-kiel.de

Abstract. Existing algorithms for signing graph data typically do not cover the whole signing process. In addition, they lack distinctive features such as signing graph data at different levels of granularity, iterative signing of graph data, and signing multiple graphs. In this paper, we introduce a novel framework for signing arbitrary graph data provided, e g., as RDF(S), Named Graphs, or OWL. We conduct an extensive theoretical and empirical analysis of the runtime and space complexity of different framework configurations. The experiments are performed on synthetic and real-world graph data of different size and different number of blank nodes. We investigate security issues, present a trust model, and discuss practical considerations for using our signing framework.

Keywords: #eswc2014Kasten

1 Introduction

Trusted exchange of graph data on the Semantic Web requires to verify the authenticity and integrity of the graph data through digital signatures. It ensures that graph data is actually created by the party who claims to be its creator and modifications on the data are only carried out by authorized parties [29]. Existing algorithms cover only a specific part of the whole graph signing process. Tummarello et al. [32] is—to the best of our knowledge—the only solution addressing the whole signing process. However, it has severe limitations as its graph signing function can only be applied on simple graphs, so-called minimum self-contained graphs (MSGs). An MSG is the smallest subgraph of a complete RDF graph that contains a specific statement d and the statements of all blank nodes associated directly or indirectly with d. Thus, in the worst case a RDF graph consists of as many MSGs as the number of statements it contains. As each statement needs to be signed separately, the approach by Tummarello et al. results in a high signature overhead in terms of time required to sign the graph as well as statements needed to represent the MSGs' signatures.

There is no solution available today that supports signing graphs at different levels of granularity (e. g., single MSGs, ontology design patterns, and entire graphs). In addition, there is no support for signing multiple graphs or iteratively signing graphs. The latter is important as it allows to build chains of signatures for provenance tracking and building a network of trust on the Semantic Web.

We address these shortcomings by a generic approach for signing graphs such as RDF(S) graphs, Named Graphs, and OWL graphs. We introduce a framework that divides the process of signing and verifying graph data into different steps as depicted in Fig. 1. These steps follow the XML standard [2]. First, a *canonicalization function* is applied to normalize the data. Thus, given two graphs that differ only in the blank node identifiers, the canonicalization function ensures that their representation is the same. This is important for the subsequent *serialization function* that transforms the canonicalized data into a sequential representation before applying a *hash function* [27] to compute a cryptographic hash value on the serialized data. Finally, the *signature function* combines the graph's hash value with a signature key [27]. The results of these four functions constitute the graph signing step. Subsequently, the *assembly function* 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.

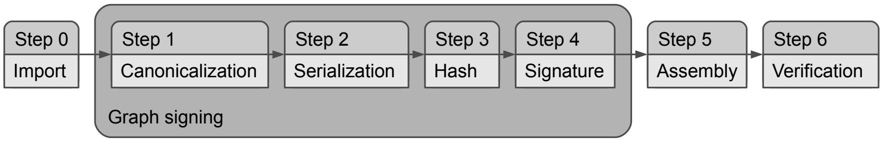

Fig. 1. The general process of signing and verifying graph data (cf. [2]).

Our approach can be considered a framework, as each of these steps can be implemented in different, independent software components [31]. Due to the formal specification of the framework's interfaces, the concrete algorithms used to implement the components can be arbitrarily combined. This enables a better comparison of different algorithms and allows to configure the framework such that it is optimized towards efficiency of the signing process or minimizing the signature overhead. In summary, the contributions and novel features of our graph signing framework are:

(i) Support for signing graphs on different levels of granularity such as minimum self-contained graphs (MSGs), set of MSGs, and entire graphs.
(ii) Support for signing multiple graphs at once.
(iii) Iterative signing of graph data for provenance tracking.

The need for our signing framework and its novel features is presented along a concrete use case in the subsequent section. Related work and implementations of concrete functions used in the graph signing process are presented in Sections 3 and 4, respectively. A formal definition of our framework is given in Section 5. Four example configurations are presented in Section 6 and their performance is empirically evaluated in Section 7. A threat model and security analysis of the example configurations is presented in Section 8. Finally, we discuss the key management and trust model in Section 9, before we conclude.

2 Scenario: Trust Network for Web Content

We consider building a trust network for Internet regulation in Germany. The information about what kind of content is to be regulated is provided as graph data by different authorities as shown in Fig. 2. In the trust network, an authority receives signed graph data from another authority, adds its own graph data, digitally signs the result again and publishes it on the web.

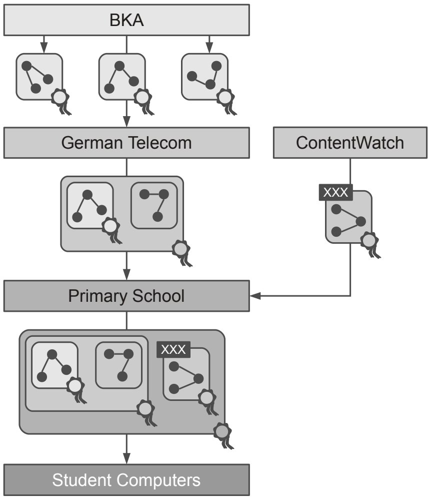

Fig. 2. Trust Network of Web Content

In the scenario, the German Federal Criminal Police Office (Bundeskriminalamt, BKA) provides a blacklist of web sites to be blocked. For example, until today the access to neo-Nazi material on the Internet is prohibited by German law (Criminal Code, §86 [8]). According to §86, the BKA digitally signs the blacklist graph along with its legal background and sends it to Internet service providers (ISPs) such as the German Telecom. By verifying its authenticity and integrity, the ISPs can trust the BKA's data. This data only describes what is to be regulated and not how it is regulated. Thus, ISPs like the German Telecom add concrete details such as proxy servers used for blocking illegal web sites. The technical details comprise graph data of different granularity such as ontology design patterns modeling the regulation meta-information as well as concrete IP addresses of the German Telecom's hardware (see graphs of different granularity (i)). The ISP adds its technical regulation details to the BKA's original blacklist graph. Subsequently, the German Telecom signs its own data together with the already signed BKA data to model its provenance relation (see iterative signing (iii) in the introduction). Customers of the Germany Telecom such as the primary school depicted in Fig. 2 are able to verify the authenticity and integrity of the regulating information. The school has to ensure that its students cannot access illegal content. The iterative signing of the regulation data allows the school to check which party is responsible for which parts of the data, i. e., it can track the provenance of the regulation data. Furthermore, the school has to ensure that adult content cannot be accessed by the students, too. To this end, it receives regulation information for adult content from private authorities such as ContentWatch. The company offers regulation data as Named Graphs to protect children from Internet pornography and the like. Thus, different regulation information from multiple sources is incorporated by the school and digitally signed, before it is deployed to the school's computers (see signing multiple graphs (ii)). This ensures that the students using these computers can access the Internet only after passing the predefined protection mechanisms.

3 Approaches for Signing and Verifying Graphs

First, we discuss different graph signing functions as depicted in Figure 1. Subsequently, we discuss existing assembly functions. 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.

Graph Signing Functions Tummarello et al. [32] present a graph signing function for fragments of RDF graphs. These fragments are so-called minimum self-contained graphs (MSGs) and are defined over RDF statements. An MSG of a statement d is the smallest subgraph of the entire RDF graph which contains this statement d and the statements of all blank nodes associated with it. Consequently, statements without blank nodes are an MSG on their own. The approach of Tummarello et al. is based on signing one MSG at a time. Thus, signing a full graph with multiple MSGs requires multiple signatures which creates a high overhead of signature statements for the whole graph. The signature of an MSG is stored as six statements, which are linked to the signed MSG via RDF Statement reification of one of its statements. This RDF reification is used for identifying the signed MSG. Due to the RDF Statement reification mechanism, if the original MSG contains blank nodes, the signature statements become part of the signed MSG as well. Signing this MSG again creates additional signature statements which also become part of the signed MSG. Thus, signature statements created in the ith signing iteration are referring to the signed MSG in the same way as the signature statements created in $i + 1$st iteration. Thus, it is impossible to distinguish between different signing iterations.

Signing a graph can also be accomplished by signing a document containing a serialization of the graph [26]. For example, a graph can be serialized using an XML-based format such as RDF/XML [4] or OWL/XML [17] and signed using the XML signature standard [2]. If the graph is serialized using a plain text-based format such as the statement-based serializations N-Triples [3] or N3 [6], also standard text document signing approaches may be used [27]. However, this means that the created signature can only be verified with the very single concrete encoding of the graph [26]. For example, if the serialized graph data changed the order of its statements (e. g., when being transferred to a triple store and retrieved back) it may not be possible anymore to verify the authenticity and integrity of the graph with the signature.

Assembly Function Tummarello et al. [32] present a simple assembly function which adds statements to the signed MSG containing the signature value and a URL to the signature verification key used for the signature's verification. Information about the graph signing function and its sub-functions is not provided. Once the URL to the 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 cannot check the true authenticity of the key as the issuer is only implicitly encoded in the key itself. Finally, the XML signature standard [2] defines a schema for describing details of the assembly function like the canonicalization function, hash function, and signature function used for computing the signature value.

4 Theoretical Analysis of Graph Signing Sub-functions

As outlined in the introduction, a graph signing function consists of four different sub-functions, namely the canonicalization function, serialization function, hash function for graphs, and signing function. We describe different implementations of the sub-functions and discuss their runtime complexity and space complexity. A formal definition of all sub-functions is provided in Section 5. Table 1 summarizes the complexity of different implementations of the four sub-functions. In the table, n refers to the number of statements to be signed and b corresponds to the number of blank nodes in the graph.

A *canonicalization function* assures that the in principle arbitrary identifiers of a graph's blank nodes do not affect the graph's signature. Carroll [9] presents a canonicalization function for RDF graphs that replaces all blank node identifiers with uniform place holders, sorts all statements of the graph based on their N-Triples [3] representation, and renames the blank nodes according to the order of their statements. If this results in two blank nodes having the same identifier, additional statements 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 $\mathcal{O}(n)$ with n being the number of statements in the graph [9]. Fisteus et al. [12] perform a canonicalization of blank node identifiers based on the hash values of a graph's statements. First, all blank nodes are associated with the same identifier. Second, a statement'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 statements have the same hash value, new identifiers of the blank nodes are computed by combining the hash values of the statements in which they occur. This process is repeated until there are no collisions left. Colliding hash values are detected by sorting them. Using a sorting algorithm such as merge sort leads to a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$. Finally, Sayers and Karp [25] provide a canonicalization function for RDF graphs, which stores the identifier of each blank node in an additional statement. If the identifier is changed, the original one can be recreated using this statement. Since this does not require sorting the statements, 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 statements created so far. This list contains at most b entries with b being the total number of additional blank node statements. Thus, the space complexity of the function is $\mathcal{O}(b)$.

Table 1. Complexity of the functions used by the graph signing function σ_N. n is the number of statements and b is the number of blank nodes in the statements.

Function	Example	Runtime	Space
Canonicalization κ_N	Carroll [9], Fisteus et al. [12]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Sayers and Karp [25]	$\mathcal{O}(n)$	$\mathcal{O}(b)$
Serialization ν_N	N-Triples [3], N3 [6], TriG [7], and others	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Hash λ_N	Melnik [16], Carroll [9]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
	Fisteus et al. [12], Sayers and Karp [25]	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Signature ϵ	RSA [24], Elliptic Curve DSA [21]	$\mathcal{O}(1)$	$\mathcal{O}(1)$

A *serialization function* transforms an RDF graph into a sequential representation such as a set of bit strings. This representation is encoded in a specific format such as statement-based N-Triples [3] and N3 [6] or XML-based RDF/XML [4] and OWL/XML [17]. TriG [7] is a statement-based format built upon N3, which allows for expressing Named Graphs. When signing RDF graphs, statement-based formats are often preferred to XML-based notations due to their simpler structure. If a serialization function processes each statement 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 like [9, 25] also include a serialization function and provide a canonicalized, serialized graph as output.

Applying a *hash function* on a graph is often based on computing the hash values of its statements and combining them into a single value. Computing a statement's hash value can be done by hash functions such as SHA-2 [20]. Melnik [16] uses a simple hash function for RDF graphs. A statement'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 statements 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 [9] uses a graph-hashing function which sorts all statements, concatenates the result, and hashes the resulting bit string using a simple hash function such as SHA-2 [20]. 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 function are the same. Fisteus et al. [12] suggest a hash function for N3 [6] datasets. The statements' hash values are computed with the canonicalization function of the same authors described above. The hash value of a graph is computed by incrementally multiplying the hash values of its statements modulo a large prime. Since this operation is commutative, sorting the statements' 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 [25] compute a hash value of an RDF graph similar to the approach of Fisteus et al. First, the statements 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 $\mathcal{O}(1)$.

A *signature function* computes the actual graph signature by combining the graph's hash value with a secret key. Existing signature functions are Elliptic Curve DSA [21] and RSA [24]. Since the graph's hash value is independent from the number of statements, the signature is as well. Thus, the runtime complexity and the space complexity of all signature functions are $\mathcal{O}(1)$.

5 Formalization of Graph Signing Framework

Based on the related work and existing graph signing sub-functions, we formally define our graph signing framework. The formalization is required for the analysis of the complexity classes of the different combinations of the sub-functions in the graph signing process. However, the reader may also directly continue with the different configurations of our graph signing framework in Section 6. By design, our framework supports signing at different levels of granularity (requirement (i) in the introduction), iterative signing ((iii) in the introduction), and signing multiple graphs at once (requirement (ii)). These requirements are fulfilled by different functions of the framework as explained in more detail in this section.

Definition of Graphs An RDF graph G is a finite set of RDF triples t. The set of all RDF triples is defined as $\mathbb{T} = (\mathbb{R} \cup \mathbb{B}) \times \mathbb{P} \times (\mathbb{R} \cup \mathbb{B} \cup \mathbb{L})$ with the pairwise disjoint sets of resources $\mathbb{R}$, blank nodes $\mathbb{B}$, predicates $\mathbb{P}$, and literals $\mathbb{L}$. Thus, it is $t = (s, p, o)$ with $s \in \mathbb{R} \cup \mathbb{B}$ being the subject of the triple, $p \in \mathbb{P}$ being the predicate, and $o \in \mathbb{R} \cup \mathbb{B} \cup \mathbb{L}$ being the object [1]. An OWL graph can be mapped to an RDF graph [22]. Thus, in the following we will only denote RDF graphs and include OWL graphs mapped to RDF graphs. The set of all possible RDF graphs is $\mathbb{G} = 2^{\mathbb{T}}$. A Named Graph extends the notion of RDF graphs and associates a unique name in form of a URI to a single RDF graph [10] or set of RDF graphs. This URI can be described by further statements, 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 $i = 1 \ldots l$. 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 defined as $\mathbb{G}_N = ((\mathbb{R} \cup \mathbb{B}) \times \mathbb{G} \times 2^{\mathbb{G}_N}) \cup \{(\varepsilon, \emptyset, G)\}$ with $G \in \mathbb{G}$.

Graph Signing Function Our graph signing function σ_N is built upon the functions described in the introduction. Input is a secret key k_s and a set of m Named Graphs NG_i with $NG_i = (a_i, A_i, \{C_{1_i}, \ldots, C_{l_i}\})$ and $i = 1, \ldots, m$. The resulting signature s is a bit string of length $d' \in \mathbb{N}$, i. e. $s \in \{0,1\}^{d'}$. The design of the graph signing function supports signing of multiple graphs at once (ii). Singing different levels of granularity (i) is achieved by interpreting all triples to be signed as graph $G \in \mathbb{G}$ and mapping G to Named Graph $C = (\varepsilon, \emptyset, G)$. C can then be signed with the graph signing function σ_N.

$$\sigma_N : \mathbb{K}_s \times 2^{\mathbb{G}_N} \to \{0,1\}^{d'}, \quad \sigma_N(k_s, \{NG_1, \ldots, NG_m\}) := s \tag{1}$$

$$\sigma_N(k_s, \{NG_1, \ldots, NG_m\}) := \epsilon(k_s, \lambda(\nu_N(\kappa_N(NG_1)) \cdot \ldots \cdot \nu_N(\kappa_N(NG_m))))$$

The different sub-functions of the graph signing function are defined below: The *canonicalization function* κ transforms a graph $G \in \mathbb{G}$ into its unique canonical form $\hat{G} \in \hat{\mathbb{G}}$ with $\hat{\mathbb{G}} \subset \mathbb{G}$ being the set of all canonical graphs. If two graphs $G_1, G_2 \in \mathbb{G}$ only differ in their blank node identifiers, it is $\kappa(G_1) = \kappa(G_2)$.

$$\kappa : \mathbb{G} \to \hat{\mathbb{G}}, \quad \kappa(G) := \hat{G} \tag{2}$$

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. 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.

$$\kappa_N : \mathbb{G}_N \to \hat{\mathbb{G}}_N, \quad \kappa_N(NG) := \hat{NG} \tag{3}$$

$$\kappa_N(NG) := \begin{cases} (\varepsilon, \emptyset, \hat{G}) & \text{if } NG = (\varepsilon, \emptyset, G),\ G \in \mathbb{G} \\ (a, \hat{A}, \{\hat{C}_1, \ldots, \hat{C}_l\}) & \text{if } NG = (a, A, \{C_1, \ldots, C_l\}) \end{cases}$$

The *serialization function* ν transforms a graph $G \in \mathbb{G}$ into a set of bit strings $\overline{G} \in 2^{\{0,1\}^*}$. A single bit string represents a statement in the graph G. The concrete characteristics of the bit strings in $\overline{G}$ depend on the used serialization format.

$$\nu : \mathbb{G} \to 2^{\{0,1\}^*}, \quad \nu(G) := \overline{G} \tag{4}$$

The serialization function ν can be extended to the function ν_N for Named Graphs $NG \in \mathbb{G}_N$. The result of ν_N is a set of o bit strings $\overline{NG} \in 2^{\{0,1\}^*}$ with $\overline{NG} = \{b_1, b_2, \ldots, b_o\}$. The function is recursively defined as follows:

$$\nu_N : \mathbb{G}_N \to 2^{\{0,1\}^*}, \quad \nu_N(NG) := \overline{NG} \tag{5}$$

$$\nu_N(NG) := \begin{cases} \overline{G} & \text{if } NG = (\varepsilon, \emptyset, G),\ G \in \mathbb{G} \\ \{a\} \cup \overline{A} \cup \overline{C}_1 \cup \ldots \cup \overline{C}_l & \text{if } NG = (a, A, \{C_1, \ldots, C_l\}) \end{cases}$$

The *hash function* λ computes a hash value h of arbitrary bit strings $b \in \{0,1\}^*$. The resulting hash value h has a fixed length $d \in \mathbb{N}$, i. e., $h \in \{0,1\}^d$.

$$\lambda : \{0,1\}^* \to \{0,1\}^d, \quad \lambda(b) := h \tag{6}$$

The hash function λ_N computes a hash value h_N of a serialized Named Graph $\overline{NG} = \{b_1, b_2, \ldots, b_o\}$ and is built upon the function λ. The function λ_N computes a hash value of each bit string $b_i \in \overline{NG}$ with $b = 1, \ldots, o$ and combines the results into a new bit string $h_N \in \{0,1\}^d$ using a combining function ϱ. Example combining functions ϱ are discussed in [25]. The function ϱ is defined as follows:

$$\varrho : 2^{\{0,1\}^d} \to \{0,1\}^d, \quad \varrho(\{h_1, h_2, \ldots, h_o\}) := h_N \tag{7}$$

Using λ and ϱ, the hash function λ_N for Named Graphs is defined as follows:

$$\lambda_N : 2^{\{0,1\}^*} \to \{0,1\}^d, \quad \lambda_N(\overline{NG}) := h_N = \varrho(\{\lambda(b_1), \lambda(b_2), \ldots, \lambda(b_o)\}) \tag{8}$$

A *signature function* ϵ computes the signature value of a graph based on the graph'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 $\mathbb{K} = \mathbb{K}_p \times \mathbb{K}_s$ 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. Using $s \in \{0,1\}^{d'}$ as identifier for the resulting bit string, the signature function is defined as follows:

$$\epsilon : \mathbb{K}_s \times \{0,1\}^d \to \{0,1\}^{d'}, \quad \epsilon(k_s, b) := s \tag{9}$$

Assembly Function An assembly function ς_N creates the signature graph $S \in \mathbb{G}$ and includes it in a Named Graph NG_S. The content and structure of S depend on the implementation of the function ς_N. The graph provides information about how to verify a graph's signature. 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 [14]. The Named Graph NG_S contains the signature graph S as its annotation graph and the signed graphs NG_i as its content graphs. In order to support iterative signing of Named Graphs (iii), the result of the assembly function ς_N is also a Named Graph which can be signed again using the graph signing function σ_N.

$$\varsigma_N : \mathbb{K}_s \times 2^{\mathbb{G}_N} \rightarrow \mathbb{G}_N \tag{10}$$

$$\varsigma_N(k_s, \{NG_1, \ldots, NG_m\}) := (a_S, S, \{NG_1, \ldots, NG_m\})$$

Verification Function 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. It also requires a signature verification function δ as inverse of the signature function ϵ. The function δ 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. It holds $\delta(k_p, \epsilon(k_s, b)) = b$ with the secret key k_s.

$$\delta : \mathbb{K}_p \times \{0,1\}^{d'} \rightarrow \{0,1\}^d, \quad \delta(k_p, s) := b \tag{11}$$

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, a signature value s, and set of signed Named Graphs $\{NG_1, \ldots, NG_m\}$. All values can be taken from the signature graph S. The 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. The signature is valid iff both computed values are equal. It is $h' = \lambda_N(\nu_N(\kappa_N(NG_1)) \cup \ldots \cup \nu_N(\kappa_N(NG_m)))$.

$$\gamma_N : \mathbb{K}_p \times 2^{\mathbb{G}_N} \times \{0,1\}^* \rightarrow \{TRUE, FALSE\} \tag{12}$$

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

6 Four Configurations of the Graph Signing Framework

The runtime complexity and space complexity when signing a graph depends on the characteristics of the graph as well as on the graph signing function σ_N and its subfunctions. The signature overhead depends on the additional statements created by these functions and on the size of the signature graph created by the assembly function ς_N. Table 2 summarizes four possible configurations A, B, C, and D of the signing framework as well as their complexity and signature overhead for signing a single graph. The example configurations correspond to the related work described in Sections 3 and 4 and are referred to by the names of their authors. Also new configurations can be created by combining different algorithms from different authors. To ease comparability, each configuration uses N-Triples for serialization and RSA as signature function ϵ. Except for B, the configurations differ only in the canonicalization function κ_N and hash function λ_N. Configuration B implements the approach by Tummarello et al. and needs an additional preparation function to split a graph into MSGs. This is required as otherwise configuration B would not be able to sign entire graphs.

A) Carroll The canonicalization function and hash function of Carroll [9] both have a runtime complexity of $\mathcal{O}(n \log n)$ and a space complexity of $\mathcal{O}(n)$ (see detailed discussion in Section 4). A graph signing function built upon these functions shares the same complexity. The canonicalization function creates b_h additional statements with $b_h \leq b$ being the number of blank nodes sharing the same identifier (see Section 4). Thus, the canonicalized graph contains b_h more statements than the original graph. A signature graph as it is used in [14] consists of 19 statements and results in a signature overhead of $b_h + 19$ statements.

B) Tummarello et al. The approach by Tummarello et al. [32] is based on the canonicalization function and hash function of Carroll [9], i. e., on configuration A. However, Tummarello et al. only allows for signing individual MSGs. In order to sign a complete graph, it has to be split into r disjoint 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 [15] 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 statements in MSG i. Since all MSGs are disjoint, it is $\sum_{i=1}^{r} n_i = n$. Thus, the total runtime complexity is $\mathcal{O}(n \log n)$ and the space complexity is $\mathcal{O}(n)$. The signature of each MSG is stored using six additional statements. Signing a graph requires r different signatures. The overhead created by the assembly function of Tummarello et al. is six statements. Thus, the overhead for r MSGs is $6r$ statements. Combined with the b_h statements from Carroll's canonicalization function results in a total overhead of $b_h + 6r$ statements.

Table 2. Configurations A–D of a signing function σ_N with runtime complexity, space complexity, and signature overhead. n is the number of statements, b is the number of blank nodes, and b_h is the number of blank nodes which require special treatment.

Configuration	**Complexity of σ_N**		**Signature overhead**
	runtime	**space**	**of σ_N and ς_N**
A) Carroll [9]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$b_h + 19$ statements, $b_h \leq b$
B) Tummarello et al. [32]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$b_h + 6r$ statements, $b_h \leq b$, $r \leq n$
C) Fisteus et al. [12]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$0 + 19$ statements
D) Sayers and Karp [25]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$b + 19$ statements

C) Fisteus et al. The approach by Fisteus et al. [12] results in a configuration with minimum signature overhead. The canonicalization function has a runtime complexity of $\mathcal{O}(n \log n)$ and the hash function has a runtime complexity of $\mathcal{O}(n)$. Since these functions have a space complexity of $\mathcal{O}(n)$ and $\mathcal{O}(1)$, respectively, the runtime complexity of the signing function σ_N is $\mathcal{O}(n \log n)$ and the space complexity is $\mathcal{O}(n)$. As the functions of Fisteus et al. do not create any additional statements, the signature overhead is independent of the signed graph and only depends on the signature graph S.

Using a signature graph S consisting of 19 statements results in a signature overhead of 19 statements.

D) Sayers and Karp The approach by Sayers and Karp [25] leads to a minimum runtime complexity of $\mathcal{O}(n)$. In order to detect already handled blank nodes, the canonicalization function maintains a list of additional statements created so far. This list contains at most b entries with b being the total number of additional statements. Assuming that each statement of a graph can contain no, one, or two blank nodes and that a blank node is part of at least one statement, the graph can contain at most twice as many blank nodes as statements, i. e., $b \leq 2n$. This results in a space complexity of $\mathcal{O}(n)$ of the graph signing function. The signing overhead is b statements added by the blank node labeling algorithm and 19 statements created by the assembly function.

7 Empirical Evaluation

We evaluate the four example configurations of our graph signing framework and their sub-functions and compare the experimental findings with our theoretical analysis in Section 6. In the experiments, we measure the runtime and the space required for singing a whole graph as well as the number of additional statements created by the graph signing function and the assembly function. As data sets, we use synthetically created RDF graphs and real graph data ranging from 10,000 to 250,000 statements. In order to measure the influence of blank nodes in the graph on the graph signing function and the assembly function, we generate different percentages of blank nodes for the graph with 250,000 statements.

The results of our evaluation confirm our theoretical analysis concerning the runtime and required memory of the algorithms as well as the signature overhead. As described in Section 4, some canonicalization functions and hash functions sort the statements in a graph. Our evaluation shows that the sorting operation performed by a hash function profits from sorting operations that are performed by a preceding canonicalization function. This results in less runtime of the hash function. The overall runtime of configurations A, C, and D is mainly influenced by the runtime of the configurations' canonicalization functions and hash functions. On the other hand, the main part of the overall runtime of configuration B is the signature function. This is due the fact that configuration B signs each MSG in the graph separately whereas all other configurations compute only one signature for the whole graph.

In our evaluation, we used RSA with a key length of 2048 bit as signature function. This corresponds to a cryptographic security of 112 bit [19]. Using a key length of 3072 bit, which corresponds to a cryptographic security of 128 bit, takes about three times longer than using a key length of 2048 bit. This does hardly affect configurations A, C, and D as they only compute a single signature. However, it highly increases the overall runtime of configuration B that needs to sign each MSG separately. As alternative, one could use Elliptic Curve DSA [21] with a key length of 256 bits. It has the same security as RSA with 3072 bit keys but is about 76 times faster (measured using a single CPU with 2.53GHz and 4G RAM).

As practical implications from the results of the empirical investigation, we can suggest that one should use the approach by Sayers and Karp (configuration D) to sign graph data that contains few blank nodes. The approach by Fisteus et al. (configuration C) might be used for graphs with many blank nodes. If indeed the approach by Tummarello et al. (configuration B) shall be used, e. g., when no iterative signing is needed, we suggest applying the faster Elliptic Curve DSA as signature function.

8 Threat Model and Security Analysis

Essential security requirements for signing graph data are to ensure the integrity and authenticity of the data. Authenticity means that the party who claims to have signed the data is really the signature's creator. This requirement is achieved with trust models which are further described in Section 9. Integrity means that the signed data was not modified after the signature was created. Achieving this security requirement depends on the used cryptographic functions that are applied on the RDF graph and its statements. Thus, we can derive the following *threat model* which covers possible actions of an attacker:

- Removing existing statements from the signed graph.
- Inserting additional statements into the signed graph.
- Replacing existing statements of the signed graph with different statements. This also covers modifying statements in the graph.

A *comprehensive security analysis* of a graph signing function σ_N must cover all possible algorithms used for its sub-functions. However, only those functions have to be analyzed which perform cryptographic operations such as the basic hash function λ, hash function λ_N for graphs, and signature function ϵ. Functions that do not perform any cryptographic operations such as sorting functions or serialization functions ν_N do not influence the security of the graph signing function. The basic hash function λ and the signature function ϵ are used in any example configuration of our graph signing framework. Thus, we conduct a security analysis of these functions first, before we discuss the particular security aspects of the four concrete configurations.

The cryptographic strength of the basic hash function λ determines the difficulty of modifying the signed graph data without being noticed by the verification mechanism. The more collision-resistant the chosen hash function is, the less likely are unauthorized modifications on the graph data such as removing statements, adding new statements, or replacing statements with other statements. A cryptographic strength of 112 bit can be achieved using SHA-2 [20] with an output length of 224 bits, whereas 128 bit security requires an output length of at least 256 bit [19]. The signature function ϵ determines the difficulty for an attacker masquerading as another party. A cryptographically strong signature function prohibits such attacks. 112 bit security can be achieved using RSA [24] with a key length of 2048 bits or Elliptic Curve DSA [21] with a key length of 224 bit. In order to achieve 128 bit security, an RSA key must be at least 3072 bits long and an Elliptic Curve DSA key must have at least 256 bits [19].

Configuration A uses the canonicalization function of Carroll [9], which does not perform any cryptographic operations. However, Carroll's graph hash function sorts all serialized statements, concatenates them, and computes a hash value of the result using a basic hash function λ. The resulting hash value is directly signed with the signature function ϵ. Thus, the security of configuration A solely depends on λ as well as ϵ. If an attacker removes a statement from the signed graph, its hash value changes and results in an invalid signature. Similarly, adding new statements to the graph or replacing statements with other statements changes the graph's hash value and thus invalidates the signature as well. *Configuration B* differs from configuration A only by using an additional split function. However, this split function does not require any cryptographic operations. Thus, the security analysis of configuration B is basically the same.

Regarding *configuration C*, both the canonicalization function and the hash function for graphs of Fisteus et al. [12] are computed based on the hash values of the statements in the graph using a hash function λ_S. The function λ_S uses a basic hash function λ for computing the hash value of a statement's subject, predicate, and object and combines the three results with a prime p. The size of p determines the bit length of the resulting hash value and thus the security of the hash function. The hash value of an entire graph is computed by applying the function λ_S on all statements in the graph. The results of single hash operations are combined using *MuHASH* [5], which is further discussed below.

Finally, *configuration D* uses the hash function for graphs by Sayers and Karp, which computes the hash values of each statement and combines the results using a combining function like *AdHASH* or *MuHASH* [5]. AdHASH adds all hash values to be combined modulo a large number m. MuHASH multiplies the hash values modulo a large prime p. In order to ensure 80 bit security, m must be chosen such that $m \gg 2^{1600}$ [33]. However, this would reduce the performance of the combining function. On the other hand, the security of MuHASH is based on the discrete logarithm problem which is proven to be hard to solve [5]. The size of p generally depends on the application in which the combining function is used. For signing graph data, one can choose a prime p with a length of at least 1024 bits as recommended in the literature [30].

9 Key Management and Trust Model

Digitally signing data is a mechanism for achieving integrity and authenticity of the data. Implementing these security requirements not only depends on the algorithms used for creating the signature but also on the organizational management of the used key material [28] and the trust model that is applied. This section briefly explains the necessity of key management and trust models as two main aspects in safely using digital signatures for graphs. Please note that our signing framework is independent from any particular key management system and trust model and can be used in any particular implementation.

Key management covers different organizational mechanisms for protecting a key pair from being compromised and misused by unauthorized parties. It ensures that a private signature key is only known to and used by its actual owner and that a public signature verification key can be related to the owner of the key pair. In order to achieve this, key management covers different tasks which have to be met when digitally signing graph data. These tasks cover secure creation and storing of keys as well as destroying old keys and revoking compromised keys [28]. Creating a key pair and storing the private key in a secure environment such as a dedicated cryptographic hardware module ensures that only authorized parties can access 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 that they are further used. Finally, the particular implementation of a key management depends also on the application in which signed graph data is used, e. g., professional environments have in general higher security requirements than private uses. Detailed guidelines for key management in professional environments are given, e.g., in [18, 19].

A *trust model* defines under which conditions a management of a public key is considered trustworthy. Public key certificates establish such trust by providing some information about the key owner together with the public key. A public key certificate is signed by a trusted party known as the *certificate authority* (CA) [28]. By signing a public key certificate, a CA states that the public key in the certificate is really owned by the party described in the certificate. Thus, the trustworthiness of a public key and its owner depends on the trustworthiness of the CA, which signed the corresponding certificate. Two widely used models for managing public key certificates are PGP [34] and X.509 [11]. X.509 follows a strict hierarchical model with a few trusted root CAs, which are often pre-configured as trust-worthy in most operating systems. An example application for X.509 certificates is SSL [13]. On the other hand, PGP has no hierarchy and allows participants to be both end users and CAs at the same time. 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. For an overview of other trust models and their characteristics, please refer to [23].

10 Conclusion

Our framework allows for signing RDF(S) graphs, OWL graphs, and Named Graphs. As described in Section 5, the framework provides supports for signing graph data at different levels of granularity (i), signing multiple graphs at once (ii), as well as iteratively signing graph data (iii). We have discussed four different example configurations of our framework and conducted a detailed theoretical analysis as well as empirical evaluation on graph data of different size and characteristics.

References

1. M. Arenas, C. Gutierrez, and J. Pérez. Foundations of RDF databases. In *Reasoning Web*, pages 158–204. Springer, 2009.
2. M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML signature syntax and processing. W3C, 2008. http://www.w3.org/TR/xmldsig-core/.
3. D. Beckett. N-Triples. W3C, 2001. http://www.w3.org/2001/sw/RDFCore/ntriples/.
4. D. Beckett. RDF/XML syntax specification. W3C, 2004. http://www.w3.org/TR/rdf-syntax-grammar/.
5. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *EUROCRYPT'97*, pages 163–192. Springer, 1997.

6. T. Berners-Lee and D. Connolly. Notation3 (N3). W3C, 2011. `http://www.w3.org/TeamSubmission/n3/`.
7. C. Bizer and R. Cyganiak. TriG: RDF Dataset Language. W3C, 2013. `http://www.w3.org/TR/trig/`.
8. Bundesrepublik Deutschland. §86 StGB, 1975. `http://www.gesetze-im-internet.de/stgb/__86.html`.
9. J. J. Carroll. Signing RDF graphs. In *ISWC 2003*, pages 369–384. Springer, 2003.
10. J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *WWW*, pages 613–622. ACM, 2005.
11. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and T. Polk. Internet X.509 public key infrastructure. RFC 5280, IETF, 05 2008.
12. J. A. Fisteus, N. F. García, L. S. Fernández, and C. D. Kloos. Hashing and canonicalizing Notation 3 graphs. *JCSS*, 76(7):663–685, 2010.
13. A. O. Freier, P. Karlton, and P. C. Kocher. The secure sockets layer (SSL) protocol version 3.0. RFC 6101, IETF, 2011.
14. A. Kasten and A. Scherp. Towards a configurable framework for iterative signing of distributed graph data. In *PrivOn*, 2013.
15. D. E. Knuth. *Sorting and searching*, volume 3 of *Art of Computer Programming*. Addison-Wesley, 2 edition, 1998.
16. S. Melnik. RDF API draft, 2001. `http://infolab.stanford.edu/~melnik/rdf/`.
17. B. Motik, B. Parsia, and P. F. Patel-Schneider. OWL 2 web ontology language XML serialization. W3C, 2009. `http://www.w3.org/TR/owl2-xml-serialization/`.
18. NIST. Recommendation for cryptographic key generation. SP 800-133, 2012. `http://dx.doi.org/10.6028/NIST.SP.800-133`.
19. NIST. Recommendation for key management pt. 1. SP 800-57, 2012. `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.
20. NIST. Secure hash standard. FIPS PUB 180-4, 03 2012. `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`.
21. NIST. Digital signature standard (DSS). FIPS PUB 186-4, 06 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.
22. P. F. Patel-Schneider and B. Motik. OWL 2 web ontology language mapping to RDF graphs. W3C, 2012. `http://www.w3.org/TR/owl2-mapping-to-rdf/`.
23. R. Perlman. An overview of pki trust models. *Network, IEEE*, 13(6):38–43, 1999.
24. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 21(2):120–126, 1978.
25. C. Sayers and A. H. Karp. Computing the digest of an RDF graph. Technical report, HP Laboratories, 2004.
26. C. Sayers and A. H. Karp. RDF graph digest techniques and potential applications. Technical report, HP Laboratories, 2004.
27. B. Schneier. *Applied Cryptography*, chapter Protocol Building Blocks. Wiley, 1996.
28. B. Schneier. *Applied Cryptography*, chapter Key Management. Wiley, 1996.
29. B. Schneier. *Secrets and Lies*, chapter Security Needs. Wiley, 2004.
30. P. T. Stanton, B. McKeown, R. Burns, and G. Ateniese. FastAD: An authenticated directory for billions of objects. *ACM SIGOPS*, 44(1):45–49, 2010.
31. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
32. G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Signing individual fragments of an RDF graph. In *WWW*, pages 1020–1021. ACM, 2005.

33. D. Wagner. A generalized birthday problem. In *CRYPTO 2002*. Springer, 2002.
34. P. R. Zimmermann. *The official PGP user's guide*. MIT press, 1995.