# Towards a Configurable Framework for Iterative Signing of Distributed Graph Data

Andreas Kasten[1] and Ansgar Scherp[2]

[1] University of Koblenz, 56070 Koblenz, Germany,
andreas.kasten@uni-koblenz.de,
[2] University of Mannheim, 68131 Mannheim, Germany,
ansgar@informatik.uni-mannheim.de

**Abstract.** When publishing graph data on the web such as vocabularies using RDF(S) or OWL, one has only limited means to verify its authenticity and integrity. Today's approaches require a high signature overhead and do not allow for an iterative signing of graph data. This paper presents a configurable framework for signing arbitrary graph data provided in RDF(S), Named Graphs, or OWL. Our framework supports signing graph data at different levels of granularity: minimum self-contained graphs (MSG), sets of MSGs, and entire graphs. It supports an iterative signing of graph data, e. g., when different parties provide different parts of a common graph, and allows for signing multiple graphs. Both can be done with a constant, low overhead for the signature graph, even when iteratively signing graph data.

## 1 Introduction

Exchanging trusted graph data on the Semantic Web is only possible to a limited extend today. On the contrary, the amount of graph data published and shared on the web has tremendously increased. In order to track provenance and building trust networks for knowledge-based systems, it becomes inherently necessary to be able to verify the authenticity and integrity of the graph data by signing it. Authenticity and integrity are basic security requirements which ensure that graph data is really created by the party who claims to be its creator and that any modifications on the data are only carried out by authorized parties. To the best of our knowledge, the only solution for signing graph data so far is the work by Tummarello et al. [1]. It provides a simple graph signing function for so-called minimum self-contained graphs (MSGs). An MSG is defined over statements. It is the smallest subgraph of the complete RDF graph that contains a statement and the statements of all blank nodes associated either directly or recursively with it. Statements without blank nodes are an MSGs on their own.

Tummarello et al. provide an important early step for signing graph data. However, it has significant shortcomings regarding the functionality provided and overhead required for representing the graph signature: First, the signing function can be applied on MSGs only. To this end, the signature is attached to the MSG by using the RDF Statement reification mechanism. This requires significant overhead for representing the signature statements. Second, it cannot be applied on, e. g., sets of statements like ontology design patterns or graphs as a whole. The approach does not support signing Named Graphs or signing multiple graphs at the same time. Finally, the approach by Tummarello et al. does not allow for an iterative signing of graph data as the signature statements become part of the MSG they sign. There is no explicit relationship between the signature and the signed statements. This makes it practically impossible to verify the integrity and authenticity of the graph data.

In this paper, we present a configurable framework for signing RDF(S) graphs, Named Graph, and OWL graphs. The general process of signing and verifying graph data is based on the XML signature standard [2] and depicted in Fig. 1. First, a *canonicalization function* normalizes the data to a unique representation. Second, a *serialization function* transforms the canonicalized data into a sequential representation (if not already provided in sequential form). Third, a *hash function* [3] computes a cryptographic hash value on the serialized data. Fourth, a *signature function* combines the data's hash value with a signature key [3]. The results of the first four functions are combined and together constitute the graph signing step. Fifth, an *assembly function* creates a signature graph containing all data for verifying the graph's integrity and authenticity, which is the last step.

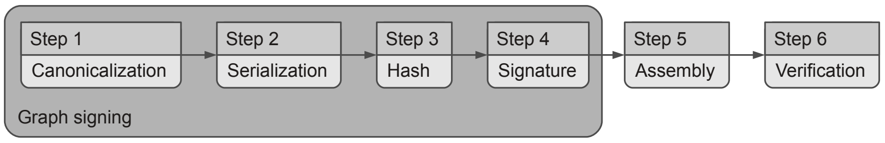

**Fig. 1.** The general process of signing and verifying graph data.

The framework as outlined in Fig. 1 can be configured, e. g., to optimize the signing process towards efficiency or minimizing the signature overhead. The resulting signature graph is assembled with the signed graph and can be published on the web. The contribution of this work is:

- A configurable framework for implementing different signing functions of graph data.
- The framework supports different levels of granularity of signing graph data. It can be used to sign a minimum self-contained graph (MSG), a set of MSGs, entire graphs, and multiple graphs at once.
- The signing process can be applied on graph data distributed over the web.
- The framework allows for an iterative signing of graph data.
- Signed graphs can contain assertional knowledge as well as terminological knowledge.
- The overhead for signing graphs is constant even when iteratively signing graph data.

The following scenario motivates the need for iteratively signing different types of graph data. The related work is presented in Section 3. Three different configurations are discussed in Section 4. Finally, we present an example implementation of our framework in Section 5.

## 2 Scenario: Trust Network for Content Regulation:

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

Due to Germany's history in the second World War, until today the access to neo-Nazi material on the Internet is prohibited by German law (Criminal Code, §86 [4]). The German Federal Criminal Police Office (Bundeskriminalamt, BKA) provides a set of formally defined ontologies making use of ontology design patterns [5]. The patterns represent knowledge such as wanted persons, recent crimes, and regulation information for Internet communication like it is required by §86. In addition, the BKA provides a blacklist of web sites to be blocked according to §86. It signs both the ontologies and the blacklist and publishes the ontologies on the web. Internet service providers (ISPs) such as the German Telecom receive the regulating information from the BKA. By verifying its authenticity and integrity, the ISPs can trust the BKA's regulation data. This data only describes what is to be regulated and not how it is regulated. Thus, ISPs like the German Telecom interpret the data received from the BKA and add concrete details such as the proxy servers and routers used for blocking the web sites. As shown in Fig. 2, the ISP compiles its technical regulation details as RDF graph which is based on the BKA's ontology pattern. It digitally signs the BKA's blacklist together with its own regulation graph and sends it to its customers. The customers 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 pupils cannot access illegal neo-Nazi content. The iterative signing of the regulation data allows the school to check which party is responsible for which parts of the data. Thus, it can track the provenance of the regulation's creation. In addition, the school has to ensure that adult content cannot be accessed by the pupils. To this

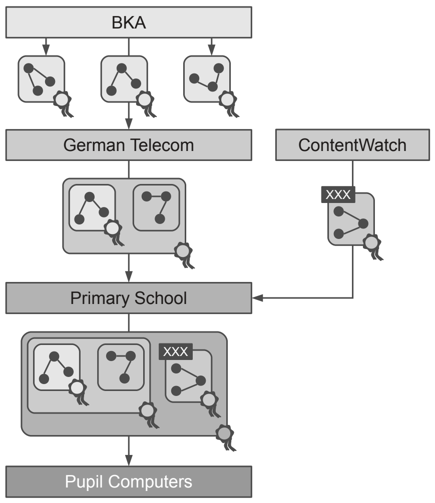

**Fig. 2.** Content Trust Network.

end, it receives regulation information for adult content from private authorities such as ContentWatch (`http://www.contentwatch.com`), which 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. Finally, the primary school digitally signs the incorporated regulation information before providing it to its client computers. This ensures that the pupils using these computers access the Internet only after passing the predefined regulation mechanisms.

## 3 Related Work

The related work is structured along the process of signing data as outlined in the introduction. An explicit discussion of the runtime complexity and space complexity of the presented functions is provided in our TR [6]. We summarize this section by explaining why the related work is not sufficient and describe the unique features of our approach. A *canonicalization function* assures that in principle arbitrary identifiers of a graph's blank nodes do not affect the graph's signature. Carroll [7] presents a canonicalization function that replaces all blank node identifiers with a uniform place holder, sorts all serialized statements of the graph, and renames the blank nodes according to the order of their statements. Additional statements are added for blank nodes sharing the same identifier. Fisteus et al. [8] provide a canonicalization function which requires a hash value of each statement based on the authors' hash function described below and sorts the statements according to their hash values. Sayers and Karp [9] provide a canonicalization function 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. The subsequent *serialization function* transforms a graph into a sequential representation such as a bit string . Functions for serializing (RDF) graphs are well known, e. g., N-Triples [10] and TriG [11]. Applying a *hash function* on a graph is often based on computing and combining the hash values of the serialized statements. Melnik [12] computes the hash value of a statement by concatenating the hash value of its subject, predicate, and object and hashing the result. The hash values of all statements in an RDF graph are sorted, concatenated, and hashed again. Fisteus et al. [8] suggest a hash function which associates all blank nodes with the same identifier, computes the statements' hash values like with Melnik's approach [12], and combines these values to form the hash value of the entire graph. Carroll [7] uses a hash function which sorts all serialized statements, concatenates the result into a bit string, and hashes this bit string using a simple hash function such as SHA-2 [13]. Finally, Sayers and Karp [9] compute a graph's hash value by incrementally multiplying the hash values of its statements modulo a prime number. *Signature functions* compute the actual signature of a graph by combining the hash values with a signature key. Possible signature functions are DSA [14] and RSA [15]. Tummarello et al. [1] present a graph signing function for minimum self-contained graphs (MSGs). An MSG of a statement is the smallest subgraph of the entire RDF graph containing this statement and the statements of all associated blank nodes. The graph signing function of Tummarello et al. is based on Carroll's canonicalization function and hash function [7]. The resulting signature is stored as a set of six statements, which are linked to the signed MSG via RDF Statement reification of one of the MSG's statements. The graph signing function signs one MSG at a time. Signing multiple MSGs requires multiple signatures. Individually signing MSGs with only one statement creates a high overhead of six signature statements. The approach by Tummarello et al. does not allow for iterative signing of graph data. The signature statements created for each signing step become part of the signed MSG. Signing this MSG again also signs the included signature statements. This makes it impossible to relate a set of signature statements to the corresponding signed graph data. Thus, verifying the signature becomes practically impossible. A graph can also be signed by signing a particular serialization of it [16]. For example, a graph serialized as RDF/XML [17] or OWL/XML [18] can be signed using the XML signature standard [2]. However, such a signature can only be verified as long as the specific serialization is still available. Finally, *assembly functions* create a detailed description of how a graph's signature can be verified. This description may be added to the signed graph data or be stored at a separate location. Tummarello et al. [1] present a simple assembly function which adds the signature value and a URL to the signature key to a signed MSG. Information about the graph signing function and its subfunctions is not provided. If the signature key is not available anymore at the URL, the signature can no longer be verified. In order to describe the parameters of a signing function, the XML signature standard [2] may be used.

In contrast to the related work on graph signing and the individual functions that contribute to graph signing, our approach allows for signing graphs at different levels of granularity like a single MSG, a set of MSGs, an entire graphs, and even multiple graphs at the same time. It supports for signing both terminological knowledge and assertional knowledge that can be distributed over different sources on the web. Finally, our graph signing approach only requires a low signature overhead, which is constant also for iteratively signed graph data.

## 4 Example Configurations of the Framework

We present three example configurations of the signing framework. To ease comparability, each configuration uses N-Triples for serialization and RSA as signature function. The configurations differ only in the canonicalization function and hash function. In the following, $n$ refers to the number of statements to be signed and $b$ corresponds to the number of blank nodes in the graph. Please note that different configurations are also possible.

*A) Tummarello et al.* [1] use the canonicalization function and hash function of Carroll [7]. Due to their complexity, the runtime complexity of the graph signing function is $O(n \log n)$ and its space complexity is $O(n)$. Carroll's canonicalization function handles blank node identifiers by sorting all of a graph's statements. Additional statements are created for blank nodes sharing the same identifier. With $b_h \leq b$ being the number of such statements, the canonicalized graph contains $b_h$ more statements than the original graph. The approach by Tummarello et al. only allows for signing a single MSG at a time. The signature is stored using six additional statements. Signing a graph with $r$ MSGs requires $r$ different signatures. The overhead created by the assembly functions is then $6r$ statements. Thus, the total overhead is $b_h + 6r$ statements.

*B) Minimum Signature Overhead* Using the canonicalization function and hash function of Fisteus et al. [8] leads to a signing process with a minimum signature overhead. Both functions have a runtime complexity of $O(n \log n)$ and a space complexity of $O(n)$. Thus, the runtime complexity of the signing function $\sigma_N$ is $O(n \log n)$ and the space complexity is $O(n)$. Since the functions of Fisteus et al. do not create any additional statements, the signature overhead is solely determined by the signature graph $S$. Using a signature graph as in the example depicted in Fig. 4 results in a signature overhead of 19 statements. When $m$ graphs are signed at the same time, the $m$ graphs are arranged using RDF bag. The resulting signature graph is of $19 + 2m$ statements.

*C) Minimum Runtime Complexity* Using the blank node labeling approach and incremental hash function of Sayers and Karp [9] leads to a minimum runtime complexity. In order to detect already handled blank nodes, the blank node labeling algorithm 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 $O(n)$ of the graph signing function. The signing overhead consists of $b$ statements added by the blank node labeling algorithm and 19 statements created by the assembly function for the signature graph $S$.

## 5 Implementation and Examples of Signed Graph Data

Our graph signing approach is designed as component-based software framework [19]. Our framework allows for implementing and providing various algorithms for the different steps of the signing process. It is implemented in Java and can be executed as command-line tool. The tool takes as input a graph or multiple graphs to sign and the user's private key and generates as output the signed graph. As output format, we use an extension of the TriG syntax [11]. This extension supports nesting of Named Graphs and thus reflects the framework's feature of iterative signing graph data. The signature statements are stored together with the content graphs, i. e., the signed graph data. Please note that this is just one possible implementation and that different output formats can also be used. It is also possible to store the content graphs separately from the signature statements. A formalization of the framework is given in our TR [6].

The subsequent examples are structured along the scenario given in Section 2. Fig. 3 shows the graph created in the scenario of Section 2. The graph has different parts signed by different parties. Each part is created by applying the graph signing function and the assembly function. In the following, we demonstrate the signing process for each party. All examples are based on configuration B of our framework (see Section 4). The first four examples (1) to (4a) use Named Graphs to associate the signature graph with the signed content graphs. In this case, the relation between the signature statements and the content graphs is implicitly given by embedding them into the same Named Graph. The last example (4b) uses a different output format which stores the signature statements separately from the content graphs. In this example, the relation between the signature statements and the content graphs is explicitly modeled as statements.

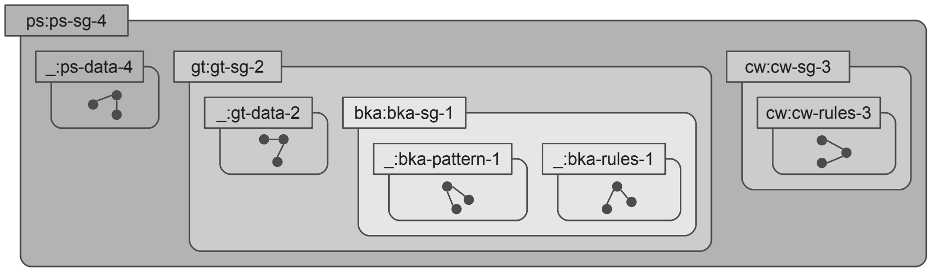

**Fig. 3.** Examples of iteratively signed graphs.

**Example 1: Signing an OWL Graph** In the first step of the scenario, the BKA creates an ontology design pattern for describing web sites to be blocked according to §86 of the German Criminal Code. Using this pattern, the BKA compiles a list of such web sites and encodes it as an OWL graph. It then signs the list along with the used regulation ontology design pattern. Listing 1 depicts a fragment of the resulting graph. The graph contains the regulation ontology design pattern, the list of blocked web sites, and a signature graph. The design pattern contains T-box knowledge of the BKA and is modeled as a separate graph `_:bka-pattern-1` shown in lines 20 to 32. The list of blocked web sites contains A-box knowledge. It is modeled as the graph `_:bka-rules-1` and shown in lines 33 to 39. Signing both `_:bka-pattern-1` and `_:bka-rules-1` results in the Named Graph `bka:bka-sg-1` and a signature graph. `bka:bka-sg-1` contains the graphs `_:bka-pattern-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 8 to 40 and the signature statements are shown in lines 9 to 19. `bka:bka-sg-1` and its two content graphs `_:bka-pattern-1` and `_:bka-rules-1` are also shown in Fig. 3 as part of the graph `ps:ps-sg-4`.

The complete signature graph created by the assembly function is depicted in Fig. 4. The signature is defined in a vocabulary following the XML signature standard [2]. The vocabulary is available from our homepage, referenced in the

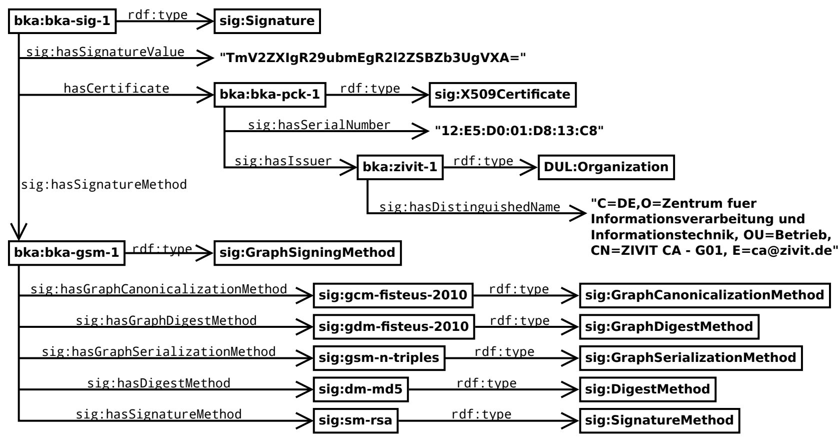

**Fig. 4.** Example signature graph following the XML signature standard [2].

conclusion. It can be combined with the W3C PROV vocabulary [20] in order provide additional information covering the signature's creator and creation date. The signature graph stores the computed signature `bka:bka-sig-1`, its signature value, and all parameters of the graph signing function required for verifying this value. In the signature graph, the function is identified as `bka:bka-gsm-1` and linked to all its subfunctions. This includes the graph canonicalization function `sig:gcm-fisteus-2010`, the graph serialization function `sig:gsm-n-triples`, the hash function (also called digest function) `sig:dm-md5`, the graph hashing function `sig:gdm-fisteus-2010`, and the signature function `sig:sm-rsa`. In order to verify the signature, the signature graph contains a reference to the BKA's public key certificate. The certificate contains the corresponding public key of the secret key, which was used as the signature key. The certificate is represented as `bka:bka-pck-1` and corresponds to an X.509 certificate [21] issued by the organization `bka:zivit-1`.

```
@prefix bka: <http://icp.it-risk.iwvi.uni-koblenz.de/policies/bka-graph#> .
@prefix DUL: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#> .
@prefix flow: <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/flow_control.owl#> .
@prefix proxy: <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/proxy_flow_control.owl#> .
@prefix sig: <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/signature.owl#> .
@prefix tec: <http://icp.it-risk.iwvi.uni-koblenz.de/ontologies/technical_regulation.owl#> .

bka:bka-sg-1 {
  bka:bka-sig-1 a sig:Signature ;
    sig:hasGraphSigningMethod bka:bka-gsm-1 ;
    sig:hasSignatureValue "TmV2ZXIgR29ubmEgR2l2ZSBZb3UgVXA=" ;
    sig:hasVerificationCertificate bka:bka-pck-1 .
  bka:bka-gsm-1 a sig:GraphSigningMethod ;
    sig:hasDigestMethod sig:dm-md5 ;
    sig:hasGraphCanonicalizationMethod sig:gcm-fisteus-2010 ;
    sig:hasGraphDigestMethod sig:gdm-fisteus-2010 ;
    sig:hasGraphSerializationMethod sig:gsm-n-triples ;
    sig:hasSignatureMethod sig:sm-rsa .
  ...
  _:bka-pattern-1 {
    proxy:URLBlockingRuleMethod a owl:Class ;
```

```
      rdfs:subClassOf flow:DenyingFlowControlRuleMethod , [
        a owl:Restriction ; owl:onProperty DUL:isSatisfiedBy ;
        owl:allValuesFrom proxy:URLBlockingRuleSituation
      ] , [
        a owl:Restriction ; owl:onProperty DUL:defines ;
        owl:someValuesFrom [ a owl:Class ; owl:intersectionOf (
          tec:EnforcingSystem [ a owl:Restriction ; owl:onProperty DUL:classifies ;
            owl:someValuesFrom tec:ProxyServer ]
        ) ]
      ] .
  }
  _:bka-rules-1 {
    bka:wst-1 a tec:WebSite ; DUL:hasQuality bka:uq-1 ; DUL:hasSetting bka:ri-1 .
    bka:uq-1 a tec:URLQuality ; DUL:hasRegion bka:ur-1 .
    bka:ur-1 a tec:URLRegion ;
      tec:hasURL "http://www.stormfront.org/" ; DUL:hasSetting bka:ri-1 .
    ...
  }
}
```

**Listing 1.** Example of a signed RDF graph.

**Example 2: Signing a Named Graph** In the scenario, ContentWatch compiles a blacklist of web sites providing adult content and encodes it as Named Graph. Signing a Named Graph is similar to signing an RDF/OWL graph. Listing 2 depicts the signed Named Graph created by ContentWatch. The blacklist is identified as `cw:cw-rules-3` (lines 7 to 12). Signing it results in several signature statements (lines 2 to 6). The statements cover the used graph signing function `cw:cw-gsm-3` (line 3), the created signature value (line 4), and ContentWatch's public key certificate `cw:cw-pck-3` (line 5). The signature statements and the Named Graph `cw:cw-rules-3` are part of the newly created Named Graph `cw:cw-sg-3` (lines 1 to 13), which contains the signature statements as its annotation graph and the graph `cw:cw-rules-3` as its content graph.

```
 1 cw:cw-sg-3 {
 2   cw:cw-sig-3 a sig:Signature ;
 3     sig:hasGraphSigningMethod cw:cw-gsm-3 ;
 4     sig:hasSignatureValue "SXQncyBibHVlIGxpZ2h0" ;
 5     sig:hasVerificationCertificate cw:cw-pck-3 .
 6   ...
 7   cw:cw-rules-3 {
 8     cw:wst-3 a tec:WebSite ; DUL:hasQuality cw:uq-3 ; DUL:hasSetting cw:ri-3 .
 9     cw:uq-3 a tec:URLQuality ; DUL:hasRegion cw:ur-3 .
10     cw:ur-3 a tec:URLRegion ; tec:hasURL "http://www.youporn.com/" ; DUL:hasSetting cw:ri-3 .
11     ...
12   }
13 }
```

**Listing 2.** Example of a signed Named Graph.

**Example 3: Iteratively Signing of Graphs** The German Telecom receives the BKA's Named Graph `bka:bka-sg-1`. This graph contains general regulation information but does not describe how the regulations shall be implemented by the ISP. Thus, the German Telecom adds its own RDF graph `_:gt-data-2` with detailed regulation information including a proxy server and its IP address. Subsequently, it signs the graph `_:gt-data-2` together with the received Named Graph `bka:bka-sg-1`. The resulting Named Graph `gt:gt-sg-2` is depicted in Listing 3. It contains the created signature statements (lines 2 to 6), the graph `_:gt-data-2` created by the German Telecom (lines 7 to 13), and the BKA's

Named Graph `bka:bka-sg-1` (lines 14 to 22). The signature statements cover the used graph signing function `gt:gt-gsm-2` (line 3), the resulting signature value (line 4), and the ISP's public key certificate `gt:gt-pck-2` (line 5). The Named Graph `gt:gt-sg-2` contains the signature statements as its annotation graph and the two graphs `_:gt-data-2` and `bka:bka-sg-1` as its content graphs.

```
 1 gt:gt-sg-2 {
 2   gt:gt-sig-2 a sig:Signature ;
 3     sig:hasGraphSigningMethod gt:gt-gsm-2 ;
 4     sig:hasSignatureValue "YXJlIGJlbG9uZyB0byB1cw==" ;
 5     sig:hasVerificationCertificate gt:gt-pck-2 .
 6   ...
 7   _:gt-data-2 {
 8     bka:pr-1 DUL:hasQuality gt:naq-2 .
 9     gt:naq-2 a tec:NetworkAddressQuality ; DUL:hasRegion gt:ipr-2 .
10     gt:ipr-2 a tec:IPv4AddressRegion ; DUL:hasSetting bka:pi-1, bka:ri-1 ;
11       tec:hasIPAddress "141.26.83.115" ; tec:hasSubnetMask "255.255.0.0" .
12     ...
13   }
14   bka:bka-sg-1 {
15     bka:bka-gsm-1 a sig:Signature ;
16       sig:hasGraphSigningMethod bka:bka-gsm-1 ;
17       sig:hasSignatureValue "TmV2ZXIgR29ubmEgR2l2ZSBZb3UgVXA=" ;
18       sig:hasVerificationCertificate bka:bka-pck-1 .
19     ...
20     _:bka-pattern-1 { ... }
21     _:bka-rules-1 { ... }
22   }
23 }
```

**Listing 3.** Example of iteratively signed graphs.

**Example 4a: Signing Multiple, Distributed Graphs** The last party in the scenario of Section 2 is the primary school. It retrieves the graph `gt:gt-sg-2` from the German Telecom and the graph `cw:cw-sg-3` from ContentWatch. In order to enrich the generic information encoded in `cw:cw-sg-3` with specific regulation details, the school adds its own regulation data as RDF graph `_:ps-data-4`. This includes a proxy server run by the school. The school signs the graph `_:ps-data-4` together with the two graphs `cw:cw-sg-3` and `cw:cw-sg-3`. This results in the Named Graph `ps:ps-sg-4` shown in Listing 4. It contains the graph `_:ps-data-4` (lines 7 to 13), the German Telecom's graph `gt:gt-sg-2` (lines 14 to 30), and ContentWatch's graph `cw:cw-sg-3` (lines 31 to 38). The school's signature graph contains the used graph signing function `ps:ps-gsm-4` (line 3), created signature value (line 4), and the certificate `ps:ps-pck-4` (line 5).

```
 1 ps:ps-sg-4 {
 2   ps:ps-sig-4 a sig:Signature ;
 3     sig:hasGraphSigningMethod ps:ps-gsm-4 ;
 4     sig:hasSignatureValue "QWxsIHlvdXIgYmFzZSBhcmU=" ;
 5     sig:hasVerificationCertificate ps:ps-pck-4 .
 6   ...
 7   _:ps-data-4 {
 8     cw:pr-3 DUL:hasQuality ps:naq-4 .
 9     ps:naq-4 a tec:NetworkAddressQuality ; DUL:hasRegion ps:ipr-4 .
10     ps:ipr-4 a tec:IPv4AddressRegion ; DUL:hasSetting cw:pi-3, cw:ri-3 ;
11       tec:hasIPAddress "141.26.83.116" ; tec:hasSubnetMask "255.255.0.0" .
12     ...
13   }
14   gt:gt-sg-2 {
15     gt:gt-sig-2 a sig:Signature ;
16       sig:hasGraphSigningMethod gt:gt-gsm-2 ;
```

```
      sig:hasSignatureValue "YXJlIGJlbG9uZyB0byB1cw==" ;
      sig:hasVerificationCertificate gt:gt-pck-2 .
    ...
    _:gt-data-2 { ... }
    bka:bka-sg-1 {
      bka:bka-sig-1 a sig:Signature ;
        sig:hasGraphSigningMethod bka:bka-gsm-1 ;
        sig:hasSignatureValue "TmV2ZXIgR29ubmEgR2l2ZSBZb3UgVXA=" ;
        sig:hasVerificationCertificate bka:bka-pck-1 .
      ...
      _:bka-pattern-1 { ... }
      _:bka-rules-1 { ... }
    }
  }
  cw:cw-sg-3 {
    cw:cw-sig-3 a sig:Signature ;
      sig:hasGraphSigningMethod cw:cw-gsm-3 ;
      sig:hasSignatureValue "SXQncyBibHVlIGxpZ2h0" ;
      sig:hasVerificationCertificate cw:cw-pck-3 .
    ...
    cw:cw-rules-3 { ... }
  }
}
```

**Listing 4.** Example of multiple signed graphs.

**Example 4b: Signing Multiple, Distributed Graphs** Listing 5 shows the same example as given in Listing 4 but is based on a different assembly function. Instead of embedding the signed graphs directly into the newly created Named Graph `ps:ps-sg-4`, `ps:ps-sg-4` only contains the signature statements and refers to the signed graphs by their URIs. The signature statements are the same as in Listing 4 and shown in lines 5 and 6. The signed graphs are modeled as list (lines 7 to 11) and contains the graphs `_:ps-data-4`, `gt:gt-sg-2`, and `cw:cw-sg-3`.

```
 1 @prefix lst: <http://ontologydesignpatterns.org/cp/owl/list.owl#> .
 2 @prefix bag: <http://ontologydesignpatterns.org/cp/owl/bag.owl#> .

 4 ps:ps-sg-4 {
 5   ps:ps-sig-4 a sig:Signature .
 6   ...
 7   ps:ps-sg-4 sig:hasSignature ps:ps-sig-4 ; sig:hasContentGraphs _:ps-cgs-1 .
 8   _:ps-cgs-1 a lst:List ; lst:hasFirstItem _:cg-ps-data-4 .
 9   _:cg-ps-data-4 a lst:ListItem ; bag:itemContent _:ps-data-4 ; lst:nextItem _:cg-gt-sg-2 .
10   _:cg-gt-sg-2 a lst:ListItem ; bag:itemContent gt:gt-sg-2 ; lst:nextItem _:cg-cw-sg-3 .
11   _:cg-cw-sg-3 a lst:ListItem ; bag:itemContent cw:cw-sg-3 .
12 }
```

**Listing 5.** Multiple signed graphs with content graphs referred to by their URI.

## 6 Conclusion

In this paper, we presented a first version of our generic framework for iterative signing of distributed RDF(S) graphs, OWL graphs, and Named Graphs. It supports signing A-box and T-box knowledge at different granularity such as single MSGs, ontology design patterns, and whole graphs. We have discussed three different configurations of our framework and its implementation and application based on TriG [11]. The complete examples as well as the signature ontology are available from: http://icp.it-risk.iwvi.uni-koblenz.de/wiki/Signing_Graphs.

*Acknowledgement:* We thank Frederik Armknecht and Matthias Krause from the University of Mannheim, Theoretical Computer Science and IT Security Group for their very valuable feedback on the security aspects of graph signing framework. We also thank Peter Schauß for implementing our framework.

## References

1. Tummarello, G., Morbidoni, C., Puliti, P., Piazza, F.: Signing individual fragments of an RDF graph. In: WWW, ACM (2005) 1020–1021
2. Bartel, M., Boyer, J., Fox, B., LaMacchia, B., Simon, E.: XML signature syntax and processing. W3C (2008) http://www.w3.org/TR/xmldsig-core/.
3. Schneier, B.: Protocol Building Blocks. In: Applied Cryptography. Wiley (1996)
4. Bundesrepublik Deutschland: §86 StGB (1975) http://www.gesetze-im-internet.de/stgb/__86.html.
5. Gangemi, A., Presutti, V.: Ontology design patterns. In: Handbook on Ontologies. Springer (2009) 221–243
6. Kasten, A., Scherp, A.: Iterative signing of RDF(S) graphs, Named Graphs, and OWL graphs: Formalization and application. Technical report, University of Koblenz-Landau (2013) http://www.uni-koblenz.de/~fb4reports/2013/2013_03_Arbeitsberichte.pdf.
7. Carroll, J.J.: Signing RDF graphs. In: ISWC 2003, Springer (2003) 369–384
8. Fisteus, J.A., García, N.F., Fernández, L.S., Kloos, C.D.: Hashing and canonicalizing Notation 3 graphs. JCSS **76** (2010) 663–685
9. Sayers, C., Karp, A.H.: Computing the digest of an RDF graph. Technical report, HP Laboratories (2004)
10. Beckett, D.: N-Triples. W3C (2001) http://www.w3.org/2001/sw/RDFCore/ntriples/.
11. Bizer, C., Cyganiak, R.: TriG: RDF Dataset Language. W3C (2013) http://www.w3.org/TR/trig/.
12. Melnik, S.: RDF API draft (2001) http://infolab.stanford.edu/~melnik/rdf/.
13. NIST: Secure hash standard. FIPS PUB 180-4 (2012) http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.
14. NIST: Digital signature standard (DSS). FIPS PUB 186-3 (2009) http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf.
15. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. CACM **21** (1978) 120–126
16. Sayers, C., Karp, A.H.: RDF graph digest techniques and potential applications. Technical report, HP Laboratories (2004)
17. Beckett, D.: RDF/XML syntax specification. W3C (2004) http://www.w3.org/TR/rdf-syntax-grammar/.
18. Motik, B., Parsia, B., Patel-Schneider, P.F.: OWL 2 web ontology language XML serialization. W3C (2009) http://www.w3.org/TR/owl2-xml-serialization/.
19. Szyperski, C.: Component software: beyond object-oriented programming. Pearson Education (2002)
20. Groth, P., Moreau, L.: An overview of the prov family of documents. W3C (2013) http://www.w3.org/TR/prov-overview/.
21. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, T.: Internet X.509 public key infrastructure. RFC 5280, IETF (2008)