IPHV8I2010X
International Journal Of Advanced Research and Innovation -Vol.8, Issue .II
ISSN Online: 2319 – 9253
Print: 2319 – 9245

HASHING AND SIGNING OF RDF GRAPHS FOR TRUSTED URLS

[#1]PEDDI KISHORE, Associate Professor & HOD,
[#2]VODAPELLY KARTHIK,
Department of CSE,
SREE CHAITANYA INSTITUTE OF TECHNOLOGICAL SCIENCES,KARIMNAGAR,T.S, INDIA.

Abstract: The ability to calculate hash values is fundamental for using cryptographic tools, such as digital signatures, with RDF data. Without hashing it is difficult to implement tamper-resistant attribution or provenance tracking, both important for establishing trust with open data. We propose a novel hash function for RDF graphs, which does not require altering the contents of the graph, does not need to record additional information, and does not depend on a concrete RDF syntax. We are also presenting a solution to the deterministic blank node labeling problem. The Semantic Web consists of many RDF graphs nameable by URIs. This paper extends the syntax and semantics of RDF to cover such Named Graphs. This enables RDF statements that describe graphs, which is beneficial in many Semantic Web application areas. As a case study, we explore the application area of Semantic Web publishing: Named Graphs allow publishers to communicate assertional intent, and to sign their graphs; information consumers can evaluate specific graphs using task-specific trust policies, and act on information from those Named Graphs that they accept. Graphs are trusted depending on: their content; information about the graph; and the task the user is performing. The extension of RDF to Named Graphs provides a formally defined framework to be a foundation for the Semantic Web trust layer.

I.INTRODUCTION

Two years ago we started to discuss requirements for Open Information Spaces (OIS): distributed systems that facilitate the sharing of data, while supporting certain trust-related properties, e.g. attribution, provenance, or non-repudiation [1, Section II]. While working on the topic, we quickly found out that it is necessary to not only record trust-relevant information, but also to make sure that the information cannot easily be tampered with. In closed systems, where access is strictly regulated and monitored, it is possible to record attribution information like “Alice created this data set” in a trustworthy manner. In open systems, where everyone is free to share and re-use any provided data sets, this is harder and usually requires some sort of cryptographic processing. One of most commonly used methods employed in this context are hash functions. They take a fixed version of a data set (the snapshot) and calculate a smaller, characteristic string for that data (the hash value). Cryptographic hash functions are constructed in a way that even very small changes to the original snapshot result in completely different hash values. Furthermore, it is a runtime expensive operation to construct a data set that yields a given hash value. Thus, by publishing the hash value for a snapshot x, it is possible to verify that some data set y is highly likely to be identical to x, simply because their hash values match. For example, to record the information “Alice created this data set”, Alice would create a hash value of the data set and digitally sign it using common cryptographic techniques. The signature could then be published as Meta data along with the data set, allowing verification of the attribution information by calculating the hash value of a local copy of the data set and comparing it to the one in the signature.

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 has tremendously increased in the last years. Thus, it becomes inherently necessary to be able to verify the authenticity and integrity of graph data published on the web in order track its provenance and building trust networks for knowledge-based systems using that data. 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. Signing graph data allows for verifying the provenance and trustworthiness of, e. g., assertional knowledge provided as RDF graphs or terminological knowledge published in form of vocabularies defined in RDFS or OWL. 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 MSG 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 the signing of Named Graphs and cannot be used to sign multiple graphs at the same time. Finally, the approach by Tummarello et al. does not allow for an iterative signing of graph data. The signature statements created for each signing step become part of the same MSG. 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 work, we present a formal framework for signing arbitrary graph data. The framework 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. As input graph data, one can use RDF(S), Named Graph, or OWL. Our 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.

II.TRUST NETWORK FOR CONTENT REGULATION

The ability to publish arbitrary content on the Internet also imposes the ethical and legal obligation to regulate access to content that is, e. g., inappropriate to minors or threatens public peace. In the scenario depicted in Fig. 1, 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. It adds its own graph data to the received one, digitally signs both, and publishes it again 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 [2]). The German Federal Criminal Police Office (Bundeskriminalamt, BKA) provides a set of formally defined ontologies (ii) making use of ontology design patterns [3]. 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 (iii) 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 (a) and integrity (b), 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. 1, 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 (iv) together with its own regulation graph (i) and sends it to its customers. The customers such as the primary school depicted in Fig. 1 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 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 (ii) to protect children from Internet pornography and the like. Thus, different regulation information (v) from multiple sources (vi) is incorporated by the school. Finally, the primary school digitally signs the incorporated regulation information (iv) before providing it to the client computers located in the school. This ensures that the pupils using these computers access the Internet only after passing the predefined regulation mechanisms.

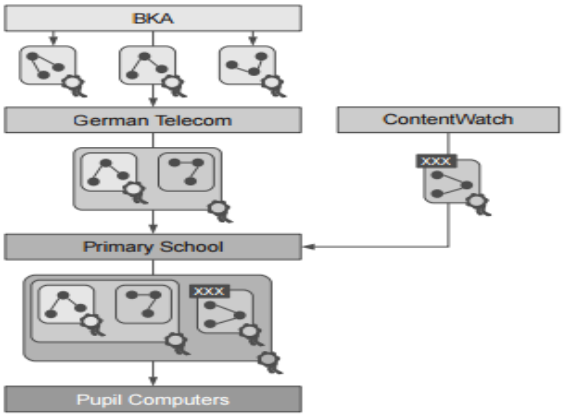

Fig. 1. Trust network for content regulation.

III.RELATED WORK

The related work is structured along the process of signing data: First, the data is normalized using a canonicalization function. This function rearranges the data's structure to a unique representation. Second, the canonicalized data is transformed into a sequential representation by applying a serialization function. If the canonicalized data is already in a sequential form, this step can be omitted. Third, a cryptographic hash value is computed on the serialized data. A hash function transforms a sequential representation of arbitrary length to one of fixed

length [4]. Fourth, the actual signature value is computed using a signature function. The signature is created by combining the hash value of the serialized data with a signature key [4]. The signature key is the secret key of an asymmetric key pair such as an RSA pair [5]. The combined results of all aforementioned functions actually make up the graph signing function. Fifth, the assembly function creates a signature graph which contains all data for verifying the graph's integrity and authenticity, which is the last step.

3.1 Canonicalization Functions for Graphs A canonicalization function assures that the in principle arbitrary identifiers of a graph's blank nodes do not affect the graph's signature. Some canonicalization functions also ensure a unique ordering of the graph's statements. A formal definition of canonicalization functions is given in Section 5.2. Carroll [6] presents a canonicalization function for RDF graphs that replaces all blank node identifier with a uniform place holder, sorts all statements of the graph based on their NTriples [7] 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 Unix's sort algorithm that has a runtime complexity of O(n log n) and a space complexity of O(n) with n as the number of statements in the graph [6]. Fisteus et al. [8] provide a canonicalization function for datasets serialized in N3 [9]. The function requires a hash value for each statement based on a hash function of the same authors described in Section 3.3. The hash function assures that the blank nodes do not affect the statements' hash values. The canonicalization function sorts the statements according to their hash values. Due to the sorting process, the runtime complexity of the canonicalization function is O(n log n) and its space complexity is O(n). Finally, Sayers and Karp [10]. 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 O(n). In order to detect already handled blank nodes, the function maintains a list of additional statements created so far. This list contains at most b entries with b as the total number of additional statements. Thus, the space complexity of the function is O(b).

3.2 Serialization Functions for Graphs

A serialization function transforms an RDF graph into a sequential representation such as a bit string or a set of bit strings. This representation is encoded in a specific format such as statement-based N-Triples [7] and N3 [9] or XML-based RDF/XML [11] and OWL/XML [12]. TriG [13] is a statement-based format built upon N3, which allows for expressing Named Graphs. HDT [14] is a binary format for encoding RDF graphs in a compact form, which requires less storage space than ASCII-based formats. When signing RDF graphs, statement-based formats are often preferred to XML-based notations due to their simpler structure. Section 5.3 gives a formalization of serialization functions. If a serialization function does not utilize the full expressiveness of its serialization format like sorting the statements, it can be implemented with a runtime complexity of O(n) and a space complexity of O(1).

3.3 Hash Functions for Graphs

Computing the hash value of 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 MD5 [15] or SHA- 2 [16]. Section 5.4 provides both a formalization of such hash functions and a formal definition of hash functions for graphs. Melnik [17] 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. Due to the use of a sorting algorithm, the function's runtime complexity is O(n log n) and its space complexity is O(n). Fisteus et al. [8] suggest a hash function for N3 datasets. First, all blank nodes are associated with the same identifier. Secondly, the statements' hash values are computed like with Melnik's approach [17]. 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. Finally, the hash value of a graph is computed by combining the hash values of its statements. Colliding hash values are detected by sorting them, which leads to a runtime complexity of O(n log n) and a space complexity of O(n). In the worst case, the runtime complexity is O(n 2) due to multiple re-hashing processes. Carroll [6] uses a graph-hashing function which serializes all statements, sorts the serialized representations, concatenates the result into a bit string, and hashes this bit string using a simple hash function such as SHA-2 [16]. As the function uses Unix's sort algorithm, it has a runtime complexity of O(n log n) and a space complexity of O(n). Finally, Sayers and Karp [10] compute a hash value of an RDF graph by incrementally multiplying the hash values of the graph's statements modulo a prime number. Since this operation is commutative and associative, sorting the statements' hash values is not required. Thus, the runtime complexity of the hash function is O(n). Due to the multiplication, the space complexity is O(1).

3.4 Signature Functions

A signature function computes the actual signature by combining the graph's hash value with a secret key. A

formalization for signature functions is given in Section 5.5. Possible signature functions are DSA [18], ElGamal [19], and RSA [5]. 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 O(1).

3.5 Graph Signing Functions

A graph signing function creates a signature for a graph by combining all aforementioned functions. A formal definition of graph signing functions is given in Section 5.6. Tummarello et al. [1] present a graph signing function for fragments of RDF graphs. These fragments are minimum self-contained graphs (MSGs) and are defined over statements. An MSG of a statement is the smallest subgraph of the entire RDF graph which contains this statement and the statements of all blank nodes associated with it. Statements 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 [6]. The resulting signature is stored as a set of six statements, which are added to the signed MSG. These signature statements are linked to the MSG via RDF Statement reification of one of the MSG's statements. The approach of Tummarello et al. is based on signing 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. Furthermore, 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. Signing a full graph can also be accomplished by signing a document containing a serialization of the graph [20]. For example, a graph can be serialized using an XML-based format such as RDF/XML [11] or OWL/XML [12]. This results in an XML document which can be signed using the XML signature standard [21]. If the graph is serialized using a plain text-based format such as the statementbased serializations N-Triples [7] or N3 [9], also standard text document signing approaches may be used [22]. However, signing the graphs on the granularity levels of single MSGs or sets of MSGs is not possible by these approaches. Iterative signing of the documents is also not supported. Most significant drawback, however, is that the signature is inextricably linked with the concrete document containing the graph [20]. This means that the created signature can only be verified with the very specific serialization of the graph contained in the document. For example, if the serialized graph data is transferred from the signed document into a triple store and then retrieved back, it is not possible anymore to verify the authenticity and integrity of the graph data with the document's signature. Finally, another option for ensuring the authenticity and integrity of the graph data would be to use secure communication channels like an SSL connection [23]. Indeed, when transmitting the graph data over a secure channel like SSL the recipient of the data can verify for the graph's authenticity and integrity. However, once the communication channel is closed after all data is transmitted, there is no chance to verify the graph data again without retrieving it once more from the provider. Verifying the authenticity and integrity of the graph data again might be necessary when the data is stored on not fully-trusted services like cloud computing or when it is transmitted further to other parties (which like to check the authenticity and integrity themselves).

3.6 Assembly Function

An assembly function 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 5.7 provides a formal definition of assembly functions. Tummarello et al. [1] present a simple assembly function which adds additional statements to a signed MSG. These statements contain the signature value and a URL to the signature key used to compute the value. Information about the graph signing function and its subfunctions is not provided. Once the URL to the signature key is broken, i. e., the signature key is not available anymore at this URL, the signature can no longer be verified. Even if a copy of the signature key is still available at a different location, the verifier finally cannot check the true authenticity of the signature key as the issuer is only implicitly encoded in the key itself. In order to describe a signing function, the XML signature standard [21] provides an XML schema containing all details of an XML signature. These details comprise the names of the used canonicalization function, the hash function, and the signature function used for computing the signature value. Furthermore, the XML schema also allows for describing a distinguished name of the signature key issuer, the key's serial number, and further information.

IV. GRAPH SIGNING FORMALIZATION

This section first defines RDF graphs as they are the basic data structure to be signed. This definition is then extended to Named Graphs. Subsequently, all functions of the signing process introduced in Section 3 are formally defined. These are the canonicalization function κN , the serialization function νN , the hash function λN , and the signature function . Using these functions, the graph signing function σN for Named Graphs is defined, followed by a definition of the signature graph S and the assembly

function ςN . The section is concluded with a description of the signature verification procedure. Fig. 2 depicts the process of signing graph data. The graph signing function is basically a combination of the functions used in the first four steps.

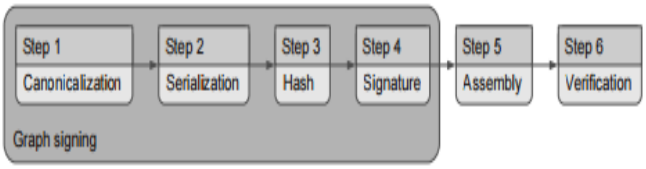

Fig. 2. The process of signing and verifying graph data. The graph signing function corresponds to the first four steps. In the fifth step, the signature graph is created. Finally, the sixth step is applied to verify the authenticity and integrity of the graph data.

V. IMPLEMENTATION AND USE OF THE SIGNING FRAMEWORK

This section describes how the previously defined graph signing function Σn and assembly function ςN are applied. The description is structured along the scenario given in Section 2. The implementation of our signing framework is conducted in Java and uses as output format an extension of the TriG syntax for named graphs [13]. In Section 7.1, we present the application of the mathematical functions de- fined in Section 5. Subsequently, we present extensive examples based on TriG for signing graph data. The examples are presented along the functional requirements defined in Section 4: The signing of OWL graphs is shown in Section 7.2. The section also demonstrates signing graph data at different levels of granularity
(REQ-1) and signing A-box and T-box statements (REQ-3). In Section 7.3, the signing of Named Graphs is shown (REQ-2) and Section 7.4 describes iterative signing of graph data (REQ-4). The signing of multiple and distributed
graphs (REQ-5) is demonstrated in Section 7.5. Overall, the examples show that our approach fulfills all functional requirements stated in Section 4. The support for the two non-functional security requirements on the authenticity of the graph data (REQ-A) and the integrity of the graph data (REQ-B) is argued in Section 7.6.

VI.CONCLUSION

In this paper, we have presented a formally defined framework for iteratively signing different types of graph data such as RDF(S) graphs, OWL graphs, and Named Graphs. The framework allows for signing multiple and distributed graphs and supports signing A-box and T-box knowledge. It also allows for signing different kinds of granularity such as single triples, ontology design patterns, and whole graphs. We have discussed three different possible configurations of the signing process and shown its practical applicability based on an extension of TriG [13]. The complete examples as well as the ontology used for modeling the signature graphs are available online. They can be found at our homepage: http: //icp.it-risk.iwvi.uni-koblenz.de/wiki/Signing_Graphs.

REFERENCES

1. Tummarello, G., Morbidoni, C., Puliti, P., Piazza, F.: Signing individual fragments of an RDF graph. In: WWW, ACM (2005) 1020–1021
2. Bundesrepublik Deutschland: §86 StGB (1975) http://www. gesetze-im-internet.de/stgb/__86.html.
3. Gangemi, A., Presutti, V.: Ontology design patterns. In: Handbook on Ontologies. Springer (2009) 221–243
4. Schneier, B.: Protocol Building Blocks. [22] 21–46
5. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. CACM 21 (1978) 120–126
6. Carroll, J.J.: Signing RDF graphs. In: ISWC 2003, Springer (2003) 369–384
7. Beckett, D.: N-Triples. W3C (2001) http://www.w3.org/2001/sw/RDFCore/ ntriples/.
8. Fisteus, J.A., Garc´ıa, N.F., Fern´andez, L.S., Kloos, C.D.: Hashing and canonicalizing Notation 3 graphs. JCSS 76 (2010) 663–685
9. Berners-Lee, T., Connolly, D.: Notation3 (N3). W3C (2011) http://www.w3.org/ TeamSubmission/n3/.
10. Sayers, C., Karp, A.H.: Computing the digest of an RDF graph. Technical report, HP Laboratories (2004)
11. Beckett, D.: RDF/XML syntax specification. W3C (2004) http://www.w3.org/ TR/rdf-syntax-grammar/.
12. 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/.
13. Bizer, C., Cyganiak, R.: TriG: RDF Dataset Language. Technical report, W3C (2013) http://www.w3.org/TR/trig/.
14. Fern´andez, J.D., Mart´ınez-Prieto, M.A., Guti´errez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web 19 (2013) 22–41
15. Rivest, R.: The MD5 message-digest algorithm. RFC 1321, IETF (1992)
16. NIST: Secure hash standard. FIPS PUB 180-4 (2012) http://csrc.nist.gov/ publications/fips/fips180-4/fips-180-4.pdf.