Hashing of RDF Graphs and a Solution to the Blank Node Problem

Edzard Höfig[1] and Ina Schieferdecker[2]

[1] Beuth University of Applied Sciences, Luxemburger Str. 10, 13353 Berlin, Germany
edzard.hoefig@beuth-hochschule.de
[2] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
ina.schieferdecker@fokus.fraunhofer.de

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.

1 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.

1.1 Motivation

Attribution is one of the fundamental characteristics that OIS needs to support, as more complex operations, like provenance tracking, build upon such an attribution framework. To engineer an OIS, we needed to start somewhere, so we decided to quickly implement an attribution system. Technology-wise, we use the Resource Description Framework (RDF) [2] to hold our data sets. Thus, our first task was the implementation of a hash function for RDF graphs. This turned out to be a complex undertaking. The problem is that RDF does not have a single, concrete syntax. Although quite rigidly defined in terms of semantics, the specification explicitly states that "A concrete RDF syntax may offer many different ways to encode the same RDF graph or RDF dataset, for example through the use of namespace prefixes, relative IRIs, blank node identifiers, and different ordering of statements." [2, Section 1.8]. Unfortunately, to work properly, hash functions need a single, concrete syntax. Often, RDF data is transmitted not as a document — which would be bound to a concrete syntax — but comes from query interfaces, e.g., SPARQL endpoints [3]. What we really needed was a hash function that can work on an in-memory RDF graph. The hash function itself is not the issue, as there are several implementations available (we are relying on SHA-256 [4, Section 6.2] to calculate the final hash value). The problem is to deterministically construct a single character string that distinctly represents the RDF graph, and the main issue here is the identity of blank nodes contained in the graph.

1.2 Paper Structure

The current section introduces the reader to the general issue and explains our motivation. In Section 2, we are studying both the underlying problems that arise when trying to implement a hash function, as well as the related work in the scientific community. Section 3 contains the description of an algorithm that is able to create a hash value for in-memory RDF graphs with blank nodes and we conclude with a critical discussion of our contribution in Section 4.

2 Problems and Related Work

Initially, we looked at the literature and found a number of articles, dating back about ten years and discussing the issue in great detail. There were even standards that seemed directly applicable, for example XML Signature [5,6]. After studying the literature, we found that none of the articles fully explains a general solution to the problem. All of them need certain constraints, or make assumptions about the data, for example the use of a certain concrete syntax. For our purposes the situation was inadequate, because of our following requirements:

1. No modification of the RDF data is needed for the algorithm to work
2. No additional data needs to be available, apart from the RDF graph
3. The algorithm works in-memory and not on a concrete syntax

During studies of the subject, we found that three issues were of paramount importance, if we wanted to solve the problem. We will explain these first, before investigating the existing work.

Blank Node Identifiers: RDF graphs might contain blank nodes. Such nodes do not have an Internationalized Resource Identifier (IRI) [7] assigned. Once loaded by an RDF implementation, they are assigned local identifiers, which are not transferable to other implementations. When trying to calculate a hash value, this is a major issue as blank nodes cannot be deterministically addressed. One can think of blank nodes as anonymous and having an identity is a necessary pre-condition for calculating a hash value. Solving the blank node issue is an algorithmic challenge.

Order of Statements Calculation of hash values effectively serialises a RDF graph structure into a string. RDF does not imply a certain order of statements, e.g. the order of predicates attached to a single subject. For serialisation purposes we need a deterministic order, or otherwise we might end up with hash values differing between implementations. This issue can be solved by adhering to a sort order when serialising the graph.

Encoding RDF uses literals to store values in the graph. These literals need to follow a common encoding, otherwise different hash values might be calculated. The same goes for namespace prefixes or relative IRIs (both features of the XML syntax for RDF [8]) –they need to be encoded with fully qualified names when stored in memory, or, at the latest, when serialised.

The most influential paper on the subject of RDF graph hashing was written by Carroll in 2003 [9]. Building on earlier work [10], Carroll explains that the runtime of any algorithm for generic hashing of RDF data is equivalent to the graph isomorphism problem, which is not known to be $\mathcal{NP}$-complete nor known to be in $\mathcal{P}$. Carroll then refrains from finding a generic solution to the problem and details his algorithm, which runs in $\mathcal{O}(n\ log(n))$, but re-writes RDF graphs to a canonical format. The proposed algorithm works on the N-Triples format (a concrete document syntax). As far as solving the blank node identity problem, the article states: "Since the level of determinism is crucial to the workings of the canonicalization algorithm, we start by defining a deterministic blank node labelling algorithm. This suffers from the defect of not necessarily labeling all the blank nodes." [9, Section 4]. Carroll continued to work on the subject, for example by publishing applications based on digitally signing graphs, together with Bizer, Hayes, and Stickler [11], but did not seem to have designed a general algorithm for hashing RDF graphs.

Sayers and Karp, colleagues of Carroll, published two technical reports at Hewlett-Packard that explains RDF hashing and applications thereof [12,13]. They identify four different ways to tackle the blank node problem [13, Section 3 ff.], namely:

Limit operations on the graph The idea is to maintain blank node identifiers across implementations, which is not possible in an open world scenario.

Limit the graph itself Avoid the use of blank nodes. This is clearly not the way for us, as we strive for a general solution to the problem.

Modify the graph Work around the problem by adding information about blank node identity within the graph. Not possible for us, as we don't want to change the RDF graph.

Change the RDF specification Generally assign globally unique identifiers to blank nodes. This will most likely never happen.

Apart from changing the RDF specification, none of these methods actually solves the problem and they can be seen as workarounds.

Other authors also investigated RDF graph hashing, e.g. Giereth [14] for the purpose of encrypting fragments of a graph. Giereth works around the blank node issue by modifying the graph. A more current approach by Kuhn and Dumontier [15] proposes an encoding of hash values in URIs. The approach replaces blank nodes with arbitrarily generated identifiers and thus needs to modify the RDF data to work.

Although Carroll did already provide pointers in the right direction, the final idea for solving the issue can be attributed to Tummarello et al., who introduce the concept of a "Minimum Self-Contained Graph" (MSG) [16]. A MSG is a partitioning of a graph, so that each MSG contains at most one transitively connected sub-graph of blank nodes. We apply this idea to construct a blank node identity, as explained in the next section.

3 The Algorithm

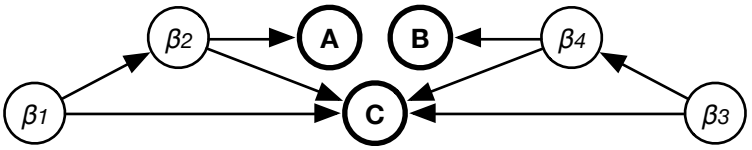

Fig. 1. An example structure with blank nodes

Our approach to the blank node labeling problem relies on constructing the identity of a blank node though its context. This is similar to the MSG principle of Tummarello et al. [16, Section 2] and a logical continuation of the thoughts of Carroll [9, Section 7.1]. For an example, see Fig. 1. The diagram shows three nodes with IRIs (**A**, **B**, **C**) and four blank nodes (β_1 ... β_4). If we go on to define the identity of a node as determined by its direct subjects, we can distinguish β_2 and β_4, but not yet the two other ones: there are both blank nodes pointing to another blank node and the **C** node. We can only discern every node by taking more of the context into account: not only direct neighbours, but neighbouring nodes one hop away. Consequentially, the identity of a blank node can only be constructed when following all of the transitive blank nodes, reachable from the original one. Using this scheme, we are able to establish an identity for blank

nodes. Having identity for both: blank nodes, and IRI nodes, we can generate a characteristic, implementation-independent string for both node types. As RDF only allows these two types of nodes as subjects, we are now able to create a list of so-called "subject strings". To solve the issue of implementation-dependent statement order, it is necessary to establish an overall ordering criterium over this list. We are using a simple lexicographical ordering based on the unicode value of each character in a string. As we are solely relying on subject nodes to calculate the hash value, we need to also encode the predicates and objects of the RDF graph into the subject strings, as well. This way of encoding graph structure into the overall data used to calculate the hash value is a major difference to other algorithms striving for the same goal. Usually, only flat triple lists are processed and the hash function calculates a value for each triple. To encode the graph structure we are using special delimiter symbols. Without these symbols, differing graph topology might lead to the same string representation and thus, the same hash value. For the final cryptographic calculation of the hash value, we are using SHA-256 on UTF-8 [17] encoded data. SHA-256 was chosen as the amount of characters in the overall data string seems to be moderate. It is recommended to use SHA-256 for less than 2^{64} bits of input [4, Section 1].

3.1 Preconditions and Remarks

To calculate the digest of a single RDF graph g, the graph has to reside in memory first, as we are not concerned with any network or file-based representations of the graph. We require that the graph's content is accessible as a set of $< S, P, O >$ triples [3]. It is beneficial to have fast access to all subject nodes of the graph, and to all properties of a node and we are using matching patterns to express this type of access, e.g. $<?n_s, ?, ? >$ to denote any node n_s that appears in the role of a subject in the RDF triple data[4]. Literals and IRI identifiers need to be stored as unicode characters. There are no restrictions on the blank node identifiers, as we do not use them for calculation of the digest. For sake of clarity, we present our algorithm broken down in four separate sub-aglorithms: A function that calculates the hash value for a given graph (Algorithm 1), a procedure that calculates the string representation for a given subject node (Algorithm 2), another one for the string representation of the properties of a given subject node (Algorithm 3), and a last one for calculation of the string representation of an object node (Algorithm 4). These sub-algorithms call each other and thus, could be combined in a single operation. It should be noted, that the algorithm uses reentrance to establish the transitive relationship needed to assign identities to the blank nodes.

Our algorithm uses a number of different delimiter symbols with strictly defined, constant values, which we assigned greek letters to. Table 1 gives an overview of these symbols.

[3] Triples with a *S*ubject, *P*redicate, and *O*bject

[4] The question mark notation is inspired by the SPARQL query syntax [3]

Symbol	Symbol Name	Value (Unicode)	Value Name
α_s	Subject start symbol	{ (U+007B)	Left curly bracket
ω_s	Subject end symbol	} (U+007D)	Right curly bracket
α_p	Property start symbol	((U+0028)	Left parentheses
ω_p	Property end symbol	) (U+0029)	Right parentheses
α_o	Object start symbol	[(U+005B)	Left square bracket
ω_o	Object end symbol	] (U+005D)	Right square bracket
β	Blank node symbol	$*$ (U+002A)	Asterisk

Table 1. Guide to delimiters and symbols used in the algorithms

Use of these symbols is unproblematic in regard to their appearance as part of the RDF content. As we use two symbols to delimit a scope in the string, we can clearly distinguish between use as delimiter and use as content. As our goal is the creation of a string representation for each subject node, the algorithm makes heavy use of string concatenation and we are using the $\oplus$ symbol to denote this operation. In several places, strings are nested between start and stop delimiters, like this: $\alpha \oplus string \oplus \omega$.

Some final remarks regarding the implementation before delving into the specifics of the algorithm: We are using some variables ($visitedNodes$, g) as parameters for functions and procedures. Of course, these should be better put away as shared variables (e.g. attributes in an object). The variable $result$ is always local and needs to be empty at the start of each function. Furthermore, there are some functions that dependent on a concrete implementation and are quite trivial to use. We skip an in-depth discussion of those, e.g., $predicates(...)$.

3.2 Calculating the Hash Value

Algorithm 1 Calculating a hash value for a RDF graph g

1: **function** CALCULATEHASHVALUE(g)
2: **for all** $n_s \in g$ that match $<?n_s, ?, ? >$ **do** ▷ All subject nodes
3: $visitedNodes \leftarrow \varnothing$
4: $subjectStrings \leftarrow subjectStrings \cup encodeSubject(n_s, visitedNodes, g)$
5: **end for**
6: sort $subjectStrings$ in unicode order
7: **for all** $s \in subjectStrings$ **do**
8: $result \leftarrow result \oplus \alpha_s \oplus s \oplus \omega_s$
9: **end for**
10: **return** $hash(result)$ ▷ Using SHA-256 and UTF-8
11: **end function**

To calculate the hash value for a given RDF graph, Algorithm 1 is used. The function takes a single parameter: the RDF graph g to use for calculation of

its hash value. At first the algorithm iterates over all of the subject nodes n_s that exist in g (lines 2–5). A subject node is any node that appears in the role of a subject in a RDF triple contained in g. For each of the subject nodes we create a data structure, called *visitedNodes*, which is used to record if we already processed some blank node. This is necessary for termination of the construction of the blank node identities. *visitedNodes* needs to be empty before calculating the string representation for n_s by calling the procedure *encodeSubject*(...) in line 4, which is explained in the next section. After all subject nodes are encoded, the resulting list needs to be sorted. Any sorting order could be used and as we do not require specific semantics for this step, we are establishing an ordering simply by comparing the unicode numbering of letters. The sorting operation in line 6 is key to deterministically create a hash value, as the result would otherwise build upon the (implementation-dependent) order of nodes in g. All of the sorted subject-strings are then concatenated to form an overall result string, while each single subject string is enclosed with the subject delimiter symbols (line 8). Finally, the result string is subjected to a cryptographic hash function and returned (line 10).

3.3 Encoding the Subject Nodes

In RDF, subject nodes can be of two types: they can either be blank nodes or IRIs [2, Section 3.1]. The procedure *encodeSubject*(...), shown in Algorithm 2 and used by Algorithm 1, needs to take care of this. The procedure has three arguments: n_s – the subject node to encode as a string, *visitedNodes* – our data structure for tracking already visited nodes, and g – the RDF graph.

Algorithm 2 Encoding a subject node n_s

```
1:  procedure ENCODESUBJECT(n_s, visitedNodes, g)
2:      if n_s is a blank node then
3:          if n_s ∈ visitedNodes then
4:              return ∅                                   ▷ This path terminates
5:          else
6:              visitedNodes ← visitedNodes ∪ n_s   ▷ Record that we visited this node
7:              result ← β
8:          end if
9:      else
10:         result ← IRI of n_s                                ▷ n_s has to be a IRI
11:     end if
12:     result ← result ⊕ encodeProperties(n_s, visitedNodes, g)
13:     return result
14: end procedure
```

The discrimination of types comes first: lines 2–8 process blank nodes and lines 9–11 take care of IRIs. For the blank nodes, we have to distinguish between the case where we already met a blank subject node (line 3–4), and the case where

we didn't (line 5–7). In the case that the subject node was already encountered, the graph traversal ends and we return an empty string. If the blank subject node is hitherto unknown, then *result* is set to the blank node symbol β (see Table 1) and the node is recorded in *visitedNodes* as being processed. If the subject is not a blank node, but an IRI, we simply set *result* to be the IRI itself. Only encoding the subject node itself is not sufficient for our purposes, as we need to establish an identity based on the context of the current subject node. In line 12 this process is triggered by calling the $encodeProperties(...)$ procedure (see Algorithm 3) and concatenating the returned string with the existing *result*. The *result* itself is returned in line 13.

3.4 Encoding the Properties of a Subject Node

Algorithm 3 is responsible for encoding all of the properties of a given subject node n_s into a single string representation. We understand properties as the predicates (p) and objects (o) that fullfil $< n_s, ?p, ?o >$, where n_s is a given subject. Apart from n_s, the procedure $encodeProperties(...)$ needs $visitedNodes$ as a second, and g as a third argument.

Algorithm 3 Encode properties for a subject node

```
1: procedure encodeProperties(n_s, visitedNodes, g)
2:     p ← predicates(n_s, g)        ▷ Retrieve all predicate IRIs that have n_s as subject
3:     sort p in unicode order
4:     for all iri ∈ p do
5:         result ← result ⊕ α_p ⊕ iri
6:         for all n_o ∈ g that match < n_s, iri, ?n_o > do   ▷ All objects for n_s and iri
7:             objectStrings ← objectStrings ∪ encodeObject(n_o, visitedNodes, g)
8:         end for
9:         sort objectStrings in unicode order
10:        for all o ∈ objectStrings do
11:            result ← result ⊕ α_o ⊕ o ⊕ ω_o
12:        end for
13:        result ← result ⊕ ω_p
14:    end for
15:    return result
16: end procedure
```

The algorithmic structure reflects the complexity of graph composition using RDF predicates: a subject node can be associated with multiple predicates, and the predicates are allowed to be similar, if associated with different objects. We use a two stage process to encode properties: First, all unique predicate IRIs of the given subject node n_s are retrieved and ordered (lines 2–3). We are postulating a function $predicates(...)$ that returns all predicate IRIs for a given subject node n_s by searching all triples for matches to $< n_s, ?p_x, ? >$, extracting the IRI of the identified predicate p_x, and eliminating double entries. In a second step,

we encode each predicate IRI and the set of objects associated with it (line 4–14). The property encoding starts in line 5, where the *result* is concatenated with the property start symbol α_p and the predicate IRI. Subsequently, we retrieve all object nodes (nodes that appear in triples with n_s as subject and *iri* as predicate), encode each object node using the procedure *encodeObject*(...), and store their respective string representations in a *objectStrings* list (line 6–8). The *encodeObject*(...) procedure is detailed in Algorithm 4. After collecting all the encoded object strings, the resulting list has to be sorted (line 9). In line 10–12 each object string is appended to *result*, while taking care to enclose the string in delimiter symbols. Encoding of a single property (one pass of the loop started in line 4) ends with appending the property stop symbol ω_p to *result*. Once the procedure has encoded all properties it returns with the complete *result* string in line 15.

3.5 Encoding the Object Nodes

Processing the object nodes itself is trivial when compared to the property encoding. Objects in RDF triples can be three things: an IRI, a literal, or a blank node [2, Section 3.1]. The *encodeObject*(...) procedure needs to return an appropriate string representation for each of these three cases. It takes three arguments: an object node n_o, *visitedNodes*, and the RDF graph g.

Algorithm 4 Encode an object node

```
1: procedure ENCODEOBJECT(n_o, visitedNodes, g)
2:     if n_o is a blank node then
3:         return encodeSubject(n_o, visitedNodes, g)        ▷ Re-enter Algorithm 2
4:     else if n_o is a literal then
5:         return literal representation of n_o              ▷ Consider language and type
6:     else
7:         return IRI of n_o                                 ▷ n_o has to be a IRI
8:     end if
9: end procedure
```

The three aforementioned cases are treated as follows. If n_o is a blank node, we continue with re-entering Algorithm 2 (see line 3). The re-entrance allows us to construct a path through neighbouring blank nodes. Together with having potentially many object nodes associated with a single subject, this yields a connected graph, similar to the MSGs of Tummarello et al. If n_o is a literal, it is returned in a format according to [2, Section 3.3] in line 5, including any language and type information. If n_o is neither a blank node, nor a literal, it has to be an IRI and we return it verbatim. After all objects, properties, and subjects have been encoded, all sub-algorithms have returned and Algorithm 1 terminates.

3.6 An Example

Consider the RDF graph shown in Figure 1 at the beginning of Section 3. When applying the algorithm to the given graph, we end up with the following string before calculating the SHA-256 hash (see Algorithm 1, line 10):

$$\{*(-[*(-[\mathbf{A}][\mathbf{C}])][\mathbf{C}])\}\ \{*(-[*(-[\mathbf{B}][\mathbf{C}])][\mathbf{C}])\}\ \{*(-[\mathbf{A}][\mathbf{C}])\}\ \{*(-[\mathbf{B}][\mathbf{C}])\}$$

Please note that this string is not valid RDF, as scheme and path information are missing from the employed IRIs[5] used by **A**, **B**, and **C**. Also the symbol "−" is used to indicate an arbitrary IRI employed for all predicates.
Thanks to the delimiters, it is quite easy to understand the string's structure. The four subject node strings for β_1, β_3, β_2, and β_4 (in this order) are encapsulated between curly brackets each. **A**, **B**, and **C** only appear as object nodes and thus do not trigger the creation of additional subject strings. Instead, they are encoded as part of the blank node subject strings. Let's take a look at the first subject string for node β_1: $\{*(-[*(-[\mathbf{A}][\mathbf{C}])][\mathbf{C}])\}$. Apart from the curly brackets, the string starts with the blank node symbol "$*$", followed by the properties of that node, delimited in parentheses. The node has only a single predicate, used with two different objects: $[*(-[\mathbf{A}][\mathbf{C}])]$ and $[\mathbf{C}]$. If we would have additional predicate types, there would also be further parentheses blocks. Objects are delimited by square brackets and due to the re-entrant nature of the algorithm object strings follow the same syntax as just discussed.

4 Conclusion and Future Work

The presented algorithm calculates a hash value for RDF graphs including blank nodes. It is not necessary to alter the RDF data or to record additional information. It is not dependent on any concrete syntax. It solves the blank node labeling problem by encoding the complete context of a blank node, including the RDF graph structure, into the data used to calculate the hash value. The algorithm has a runtime complexity of $\mathcal{O}(n^n)$, which is consistent with current research on algorithms for solving the graph isomorphism problem [9]. The worst-case scenario is a fully meshed RDF graph of blank nodes, which does not seem to make any sense whatsoever — usually, we would expect the amount of blank nodes in a graph to be far smaller, thus the real execution speed to be less catastrophic. We concentrated on solving the primary problem, not on runtime optimisations and we are certain, that there is room for improvement in the given algorithm. There are some obvious starting points for doing this. For example, there are redundancies in the string representations for transitive blank node paths (the string representations for β_2 and β_4 appear twice in the example given in Section 3.6). One could cache already computed subject-strings, pulling them from the cache when needed. Also, the interplay between the SHA-256 digest computation and the subject string calculations has not been researched in sufficient detail.

[5] For example **A** instead of http://a/

It might be possible to reduce processing and storage overhead by combining these two operations, calling the digest operation on each subject string and combining the resulting values in order of the final sorted list. The sensibility of such optimizations should largely depend on the susceptibility of the digest algorithm implementation to the length of the given input strings. This, in turn, depends on the usage of blank nodes in the input RDF data. More blank nodes in the input data and more references between blank nodes means longer subject strings. Consequentially, to come to a more substantial assessment of the presented algorithm, we will need to study its performance on a number of real (and larger) data sets.

While we trust the general approach for solving the blank node labeling problem through an assignment of identity based on the surrounding context of the node, we did not proof that the algorithm works correctly. To assure that it works properly, we did test it: on the one hand in regard to its ability to process all possible RDF constructs using tests from W3C's RDF test cases recommendation [18], on the other hand in regard to the correctness of the blank node labeling approach using manually constructed graphs. These graphs range from simple, non cyclic ones with a single blank node to all possible permutations of a fully meshed graph of blank nodes with variations on the attachment of IRI nodes and predicate types.

Acknowledgments. We would like to thank Thomas Pilger and Abdul Saboor for their collection of material documenting the current state of the art and for tireless implementation work.

References

1. Höfig, E. Supporting Trust via Open Information Spaces. Proc IEEE 36th Annual Computer Software and Applications Conference, pp. 87–88, (2012)
2. World Wide Web Consortium.: RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, `http://www.w3.org/TR/rdf11-concepts/` (2014)
3. World Wide Web Consortium.: SPARQL 1.1 Query Language, W3C Recommendation, `http://www.w3.org/TR/sparql11-query/` (2013)
4. National Institute of Standards and Technology.: Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180–4 (2012)
5. World Wide Web Consortium.: XML Signature Syntax and Processing (Second Edition), W3C Recommendation, `http://www.w3.org/TR/xmldsig-core/` (2008)
6. Cloran, R., Irwin B.: XML Digital Signature and RDF. Proc. Information Security South Africa (2005)
7. Duerst, M., Suignard, M.: Internationalized Resource Identifiers (IRIs). Internet Engineering Task Force – Request for Comments 3987 (2005)
8. World Wide Web Consortium.: RDF 1.1 XML Syntax, W3C Recommendation, `http://www.w3.org/TR/rdf-syntax-grammar/` (2014)
9. Carroll, J. J.: Signing RDF Graphs. Proc. of the Second International Semantic Web Conference, LNCS 2870, pp. 369–384 (2003)
10. Carroll, J. J.: Matching RDF Graphs. Proc. of the First International Semantic Web Conference, LNCS 2342, pp. 5–15 (2002)

11. Carroll, J. J., Bizer, C., Hayes, P., Stickler, P.: Named Graphs, Provenance and Trust. Proc. International World Wide Web Conference, pp. 613–622 (2005)
12. Sayers, C., Karp. A. H.: Computing the Digest of an RDF Graph. Hewlett-Packard Labs Technical Report HPL-2003-235 (2003)
13. Sayers, C., Karp. A. H.: RDF Graph Digest Techniques and Potential Applications. Hewlett-Packard Labs Technical Report HPL-2004-95 (2004)
14. Giereth, M.: On Partial Encryption of RDF-Graphs. Proc. of the Fourth International Semantic Web Conference, LNCS 3729, pp. 308–322 (2005)
15. Kuhn, T., Dumontier, M.: Trusty URIs: Verifiable, Immutable, and Permanent Digital Artifacts for Linked Data. Proc. Eleventh European Semantic Web Conference, LNCS 8465, pp. 395–410 (2014)
16. Tummarello, G., Morbidoni, C., Puliti, P., Piazza, F.: Signing Individual Fragments of an RDF Graph. International World Wide Web Conference, pp. 1020–1021 (2005)
17. Yergeau, F.: UTF-8, a Transformation Format of ISO 10646. Internet Engineering Task Force – Request for Comments 3629 (2003)
18. World Wide Web Consortium.: RDF Test Cases, W3C Recommendation, `http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/` (2004)