Computing the digest of an RDF graph

Craig Sayers, Alan H. Karp
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto

HPL-2003-235(R.1)
March 23, 2004*

RDF(Resource Description Framework), graph, digest, signature, cryptography, security

Efficient algorithms are needed for computing the digest of a Resource Description Framework (RDF) graph. These may be used to assign unique content-dependent identifiers and for use in digital signatures, allowing a recipient to verify that RDF was generated by a particular individual and/or has not been altered in transit. Ideally the digest algorithms should still permit verification of the RDF graph even when it has been transported via intermediaries who may serialize in different formats and alter the blank node identifiers.

In an advance over previous work, we show that the use of a set hash allows the digest of an RDF graph to be computed in O(N) time. Furthermore, our algorithm allows for incremental updates to the graph, so the time to recompute the digest to account for additional statements is proportional to the number of new statements being added.

The security of the algorithm depends on the number of statements and the choice of the combining and hashing algorithms used for the set hash.

1 Introduction

Imagine that you wish to transmit a document and later allow a recipient to verify that they received exactly what you sent.

If you are concerned only with detecting hardware failures then providing the recipient with a simple checksum might suffice. However, if you are concerned about detecting an adversary who may deliberately try to send a fake file, then you need something more sophisticated. That something is a "message digest".

A "digest" is a kind of digital fingerprint for a message or document [18, 11]. Formally, it has the properties that:

- It is typically much smaller than the original document.
- It is relatively easy to compute.
- It is "one-way". That is, given a digest it is infeasible to find the original document.
- It is "collision-resistant". That is, given a digest and a document it is very difficult to find a different document that has the same digest.

A common use of digests is for signing documents [16]. Computing a digital signature is usually expensive. So it is common to compute a message digest and then sign that instead of the original message [1]. The message digest has the property that it is computationally very difficult to find a different message that generates the same digest. Thus, a recipient can extract the digest from the signature and compare it to the computed digest of the received message to verify authenticity.

Digests have other interesting uses as well. For example, you can compare digests computed at different times to see if a document has changed and the digest can also serve as a convenient content-based identifier [7, 17].

In addition, if the digest may be computed incrementally then you can add information to a document and generate an updated digest even if you don't know all the content of the original document.

There are already a number of well-known digest algorithms that operate on serialized data. For example, MD5 and SHA-1 both take inputs of arbitrary length and generate digests with fixed lengths of 128 and 160 bits respectively [15, 8]. Since the computation must be repeatable for both sender and recipient, it is usual to construct a canonical serialization of the input data and then apply the digest function. This works well for text documents, or tree-like structures (such as XML) which have a well-defined ordering.

In the case of graphs, the ordering of their arcs or nodes may be undefined and so the creation of a canonical serialization is non-trivial. Indeed it has been shown that any solution for creating a canonical serialization would also provide a solution to the known-hard graph isomorphism problem [5].

In this paper we examine the particular case of a graph containing statements expressed in the Resource Description Framework (RDF) (see Section 1.1). By carefully avoiding the need for an intermediate canonical serialization we are able to efficiently compute a digest for such graphs.

1.1 An RDF Graph

The Resource Description Framework (RDF) [12, 10, 14] provides a means to make statements about resources. An RDF graph is a set of such statements. Figure 1 shows an example graph.

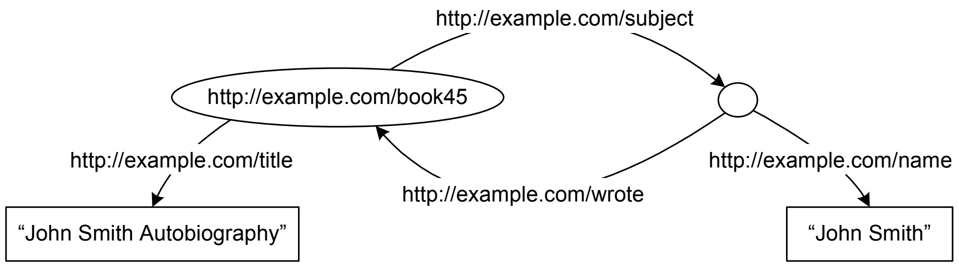

Figure 1: An example of an RDF graph containing statements about an autobiography and its author.

In this case, the graph encodes statements about an autobiography and its author. There are two literal nodes to represent the title of the book and the name of the author. The node representing the book itself is labeled with a Universal Resource Identifier (URI); while the node representing the author is a "blank node". Blank nodes like this are common in RDF graphs and may be used in cases where a universal identifier is either unnecessary or unknown. Notice also that the graph contains a cycle, since the subject of this autobiographical book is naturally also its author.

To transport the graph between machines we first need to serialize it. In the case of RDF there are a number of different serialization schemes, but they all have two properties in common:

- The ordering of statements is undefined and may vary.
- Any blank nodes may be assigned arbitrary labels and those need not be maintained.

For example, if the graph in Figure 1 were serialized in an abbreviated format it might look like this on one machine[1]:

```
book45 title ‘‘John Smith Autobiography’’ .
book45 subject _1 .
_1 wrote book45 .
_1 name ‘‘John Smith’’ .
```

If that were loaded into another machine and then again serialized the result might look like this:

```
_451 wrote book45 .
book45 title ‘‘John Smith Autobiography’’ .
_451 name ‘‘John Smith’’ .
book45 subject _451 .
```

Naturally this reordering of statements and relabeling of blank nodes complicates computation of the digest.

1.2 Background

If we were simply sending a graph directly from one agent to another then we can serialize the graph in a message and then compute a digest and signature for that message using conventional means. This has been used successfully in the past [2] for serialized RDF messages and works well for direct communication where it is sufficient to sign a particular message rather than the graph itself. The difficulty with this approach is that the digest

[1]For brevity we have used the N-triples syntax but have abbreviated the URIs by removing the “http://example.com/”

only applies to that particular message. Thus, to permit later verification of the digest you must have access to the original message.

Previous approaches to the problem of computing the digest for a graph (rather than a message containing a graph) have broken it into two parts, first creating a canonical representation and then computing the hash of that. As noted earlier, this is complicated by the order-independence of the RDF statements and the presence of blank nodes.

In [13] they take the hash of each statement, sort those and append them. Then they compute the digest of the result. This gives an algorithm that takes O(N log(N)) time and apparently does not consider blank nodes.

In [5] they make a canonical serialization using a sorting algorithm and then compute the digest of that. Their algorithm is more sophisticated since it does handle blank nodes but it still takes O(N log(N)) time due to the sort.

In our algorithm, we compute the digest without doing any intermediate canonical serialization. The key to this approach is recognizing the applicability of incremental cryptographic functions. These were introduced by Bellare *et. al.* [3, 4] and are similar to the Set Hash described much earlier by Zobrist [21]. The idea is to compute a digest not by working on an entire message at once, but instead break the message into pieces, pass each through a randomizing function and then combine the results.

2 Definitions

An RDF graph is a set of statements. Each statement describes an arc between two nodes of the graph and may be represented by a triple:

> *subject predicate object .*

where the arc goes from the *subject* node to the *object* node and is labeled by the *predicate*.

Let S be the set of statements in a graph. Each set contains N triples $t_1, t_2, \ldots t_N$ and each triple, t_i, consists of a subject s_i, predicate p_i and object o_i.

Our goal is to find a digest function, $\mathcal{D}$, which generates:

$$d = \mathcal{D}(S)$$

and is:

- Compact. d is typically much smaller than S.
- One-way. That is, given a digest it is infeasible to find the original graph. If an adversary knows d it is very difficult for them to determine S.
- Collision-resistant. That is, given a digest and a graph it is very difficult to find a different graph that has the same digest. If an adversary knows $\mathcal{D}$ and S it is very difficult to find a set of statements T such that:
  $$\mathcal{D}(S) \equiv \mathcal{D}(T) \ \textit{when} \ S \not\equiv T$$
- Incremental. Given $\mathcal{D}(S)$ and a new triple t_{N+1} it should be possible to compute $\mathcal{D}(S \cup t_{N+1})$ in constant time (assuming some upper bound on the size of resources and literals in the triple).

In addition, we desire the digest algorithm to be independent of statement ordering and require it to be invariant to changes in the blank node labels.

We also need the algorithm to perform gracefully in the presence of duplicate statements. Since the RDF graph is a set of statements, then by definition a properly constructed graph should not contain any duplicates. However, an error or deliberate adversary could cause us to receive a message which did contain duplicates. Accordingly, we require the graph digest to produce a different answer if there are any duplicate statements present and we require it to still maintain all the other required properties (one-way, collision-resistant, incremental) even in that case.

3 Incremental Cryptography

In [4] they suggest using a function defined as follows:

Given a message X, consisting of N ordered message blocks: $x_1 \ldots x_N$, they compute a digest of that message:

$$\mathcal{D}(X) = \bigodot_{i=1}^{N} h(<i> .x_i)$$

Where:

- $\odot$ is a combining operation which is both associative and commutative to support incremental operation (see Section 6.2 for a discussion of suitable functions).
- $h()$ is the randomizing function. This takes an input of arbitrary length and generates an n bit output (they recommend a keyed version derived from a known hashing algorithm [8]).
- $<i>$ is an index used to ensure the order-dependence of the message blocks.

In our case, with RDF statements, we can treat each individual statement as one block of the message and we specifically want the message digest to be independent of the statement order. Accordingly, we remove the indexing term $<i>$ to give:

$$\mathcal{D}(X) = \bigodot_{i=1}^{N} h(x_i)$$

Applying this to our RDF triples gives:

$$\mathcal{D}(S) = \bigodot_{i=1}^{N} h(\mathit{serialization}(t_i))$$

Where:

- *serialization*() may be done using a subset of any of the RDF serialization syntaxes. The only special requirements are that it must not allow any optional characters, must be repeatable, operate one statement at a time and must include an identifier for each blank node. We find it convenient to use a subset of the N-triples syntax [9] with no optional whitespace.

To further enhance the digest we can add additional information into the digest:

$$\mathcal{D}(X) = \left(\bigodot_{i=1}^{N} h(\mathit{serialization}(t_i)), Ex \right)$$

Where Ex is an extra piece of information, derived from X. While there are many choices for Ex, one simple example is to use N the number of statements in the RDF graph. This means that an adversary who wanted to beat the digest would need to find another RDF graph which had exactly the same number of statements and generated the same set hash. It should be noted that while this seems obviously more difficult, it does not necessarily make the algorithm more secure.

4 Message digest without blank nodes

Assume for now that we have an RDF graph containing no blank nodes.

Person A has such a graph comprising a set of statements, S to send to B. To compute $\mathcal{D}(S)$ Person A uses the algorithm:

1. Compute a set hash over all the triples in S:

$$H = \bigodot_{i=1}^{N} h(\mathit{serialization}(t_i))$$

2. Construct a digest:

$$\mathit{digest} = (H, N)$$

3. sign the digest and send to B.

To verify the received message is correct, B does:

1. Retrieve the graph, S' containing triples $t'_1 \dots t'_{N'}$ sent by A and compute the set hash:

$$H' = \bigodot_{i=1}^{N'} h(\mathit{serialization}(t'_i))$$

2. Retrieve the digest sent by A and extract the expected set hash H and extra information N.

3. The graph is verified if:

$$H \equiv H' and N \equiv N'$$

4.1 Incrementality

It should be clear that, given a digest, we can extract the H and N and then add an additional triple, t_{N+1} with:

$$H' = H \odot h(serialization(t_{N+1}))$$

$$digest' = (H', N+1)$$

In addition, if $\odot$ is invertible then we can also remove triples incrementally.

The incrementality also provides for some interesting applications. For example you can add or remove statements and recompute the digest for a graph without needing to know the whole graph [2]. The potential for separation of knowledge is also valuable. For example, if two services offered computation of incremental digests, you could send half the graph to each service and then combine the results.

5 Consideration for blank nodes

The use of blank node identifiers is optional in some serialization schemes. Even in cases where names are specified, they are only guaranteed to be unique within a particular graph. For example several different machines could all generate RDF graphs which used the blank node identifier "_:1". As a result, some implementations routinely rename incoming blank nodes, assigning each a new unique local identifier. This renaming does not change the meaning of the graph and it can assist in later processing (avoiding conflicts when merging graphs, for example). Unfortunately it also means that a statement which is serialized on one machine need not be identical

[2] You do however need to know enough to be able to safely avoid accidentally introducing duplicate statements

(on a character-by-character basis) to an equivalent statement on another machine even when both machines use identical serialization syntaxes.

We deal with this in two ways:

5.1 Rely on consistent interpretation

In rare cases, the sender may know that the serializations used on the sending machine and any intermediary machines maintain blank node identity and that that identity is preserved at the recipient for long enough to compute the hash. This is possible, for example, if the sender uses the N-triples syntax, the serialization is not altered in transit, and the recipient can extract and use the blank node identities from the received serialization for verifying the hash.

In this special case, one can use the presented algorithm without change.

5.2 Add additional statements

In the more general case, the graph from machine A may be serialized and deserialized multiple times before it reaches B and there is no guarantee that the arbitrary labels assigned to blank nodes on machine A will be maintained.

The solution is to add additional statements to the graph to capture the arbitrary labels assigned to the blank nodes. This is similar to the technique of Carroll [5] and they provide some nice formal semantics which are equally applicable here. The algorithms used are different however. While they use a sophisticated algorithm to add as few labels as possible, we take the opposite approach and simply add a label for every blank node.

Specifically, on machine A, compute the hash as above, and then for every blank node, _b, which is serialized with internal label L, add a new statement:

`_b hasLabel` L .

Then A sends to B the modified graph (including the blank node label statements) and the signature of the modified graph.

On machine B, extract the blank node label statements and use those to temporarily relabel the blank nodes so the internal labels on B match those used on A. Now verify the hash using the same algorithm as above.

If machine B forwards the graph on to another machine without modifying it then no special accommodation for blank nodes is necessary (the ultimate receiving machine can do the relabeling if it needs to recompute the digest). However, if B does choose to modify the graph and it wishes to recompute the digest incrementally then it must perform the modifications on the relabeled graph and maintain those blank node labels.

6 Choosing combining and randomizing functions

In applying the algorithm we have two main knobs we can turn to alter performance and security. These are the choices for the randomizing and combining functions.

6.1 The randomizing function

There are a number of existing hashing functions which are suitable choices for $h()$, the randomizing function that takes an input of arbitrary length and generates an output with n bits.

As always, there is a trade-off between performance and security. For practical use the existing SHA-1 algorithm appears a good choice [8, 18]. It generates an output with 160 bits and implementations are readily available.

More sophisticated randomizing functions, such as those which require a key or those which generate much longer outputs, may be constructed using the SHA-1 algorithm as a building block (see [4]).

We note that the graph digest algorithm does not depend on any particular randomizing function and as better hash functions are developed they may naturally be incorporated.

6.2 The combining function

The combining function, $\odot$, must obviously be associative and commutative, but in addition it also needs to meet some cryptographic requirements. In this section we consider three different choices.

6.2.1 XOR

Using XOR as the combining function is an efficient, but very poor choice. As noted above, we require the graph digest to cope gracefully with duplicate statements (even though the definition of the graph precludes duplicates, we cannot rely on an adversary complying with those rules). XOR does not behave well in the presence of duplicate statements. Thus it is not an appropriate choice.

Even if we could guarantee there were no duplicate statements, XOR would still be a poor choice since there are known efficient algorithms for finding collisions [4].

6.2.2 Multiplication

If the combining function, $\odot$ is multiplication mod some suitably-large prime P then the security is expected to be very good (see [4]) but the operations are relatively expensive.

6.2.3 Addition

Provided that a very high level of security is not required, then addition modulo a suitably-large number M offers a good compromise between speed and security. This is based on the "AdHash" algorithm of Bellare [4] and is similar to the "MSET_VADD_HASH" algorithm recently analyzed by Clarke *et. al.* [6]. Note that M must be at least as large as 2^n, where n is the number of bits in the randomizing function. (In practice, if the number of statements is limited we can dispense with the modulo arithmetic entirely.)

This gives us a digest where the probability of another set of statements happening by chance to have the same hash is acceptably small. This is quite sufficient for many practical purposes.

However, it should be noted that the use of addition and a 160-bit randomizing function is likely insufficient to guarantee security against a determined adversary. Recently Wagner [19] has shown an efficient algorithm for finding a set of numbers which sum to a particular value. That algorithm is most efficient when the adversary is allowed to choose the number of things to sum. In particular, for the special case where the number of statements is exactly $2^{\sqrt{n}-1}$ they have a solution which runs in $O(2^{2\sqrt{n}})$. Without additional steps, achieving adequate security against a determined adversary may require n to be at least 1200 [20].

6.3 Improved security

If a very high security were desired then we can do even better by having a shared secret between sender and recipient. In particular we could replace the public hash used for randomizing function $h()$ with a message authentication code (MAC) function [18, 4]. Such functions are like a hash, but require a secret key to obtain the hash value.

In the specific case of RDF graphs, we note that the task for an adversary is more complicated than simply finding a collision, since an adversary doesn't just have to generate RDF statements that have the same digest, they also have to generate RDF statements that are similar enough to a true set to fool a recipient into performing some operation advantageous to the attacker.

7 Time complexity

We first consider a theoretical lower-bound and then analyze the time for this algorithm.

7.1 A theoretical lower-bound

When computing the digest of an arbitrary RDF graph, the fastest possible algorithm requires $O(N)$ time. The proof is as follows:

Assume there were an algorithm which ran in a faster time than $O(N)$. That algorithm can not visit each triple, for doing so would require $O(N)$ time. Therefore there must be at least one triple that the algorithm does not visit.

But in that case an adversary could alter that triple to generate a different graph with the same digest. This violates the collision-resistance property. Hence there is no algorithm which runs in faster than $O(N)$ time.

A probabilistic algorithm is possible. For example one could probabilistically choose a subset of the information in the graph for use in the digest computation. However, for most practical implementations the probability that any particular piece of information is used must be high (otherwise it is likely that an attacker could change some content without affecting the digest). Even if the probability were low, it must still be constant with respect to the number of statements and hence the algorithm is still $O(N)$.

7.2 Analysis of our algorithm

When dealing with a single statement, and assuming an N-triples serialization syntax, then the time to generate the serialization and then compute its hash is dependent on the length of the serialized statement.

If we assume that there is an upper-bound on the length of a statement then both $\mathit{serialization}()$ and $h()$ take $O(1)$ time. The process of adding and processing the additional statements for blank node identity takes $O(N)$ time (since there can't possibly be more than twice as many blank nodes as statements). Provided the combining operation may combine two hashed blocks in $O(1)$ time, then the entire algorithm on both sender and recipient takes $O(N)$ time.

This offers a better theoretical performance than any existing RDF graph digest algorithm and matches the theoretical lower-bound. However, it should be noted that, particularly if a high security is desired, the constant in our $O(N)$ time can be large. Thus it may not always provide the best practical solution (especially if the number of statements in the graph is not large, or an application is unable to take advantage of the incremental updates).

We note that in some practical cases it may be possible to use additional knowledge about the graph content to further improve performance. For example, if you knew that only some subset of the information was important, you could choose to only compute the digest for that subset.

8 Conclusions

Our task is to take an RDF graph and compute a digest for that graph. The digest can be used as a content-based identifier for a particular instance of the graph. We can also sign the digest, and then later use the signature to verify both that the graph was authored by a particular person and that it has not been altered.

By computing the digest for a graph (rather than for a particular instance of a message containing that graph) we allow the digest to be recomputed later even after the graph has been passed through intermediaries who may reorder statements, relabel blank nodes, and serialize in different formats.

We have shown that in the general case there is no graph digest algorithm which can run faster than $O(N)$.

By drawing on the work of others in incremental cryptography and applying it to RDF graphs, we have created an algorithm which runs in $O(N)$ time and permits incremental operation.

The security of the algorithm depends on the number of statements, the choice of the combining and hashing algorithms and the modulo of arithmetic used. For the case where the hashing function is SHA-1 and combining is performed using addition modulo 2^{160} then the resulting digests are sufficient for content-based identifiers but may be insufficient to guarantee security against a determined adversary.

9 Acknowledgements

Many thanks to Dave Reynolds and Jeremy Carroll for their comments on a very early writeup and to Kave Eshghi for advocating content-based identifiers and suggesting improvements to an earlier draft. Thanks also to Wenbo Mao and David Wagner for their helpful comments on an earlier version.

References

[1] M. Atreya. Digital signatures and digital envelopes, RSA Security white paper. http://www.rsasecurity.com/products/bsafe/whitepapers/Article5-SignEnv.pdf, November 2003.

[2] D. Banks, S. Cayzer, I. Dickinson, and D. Reynolds. The ePerson snippet manager: a semantic web application. Technical Report HPL-2002-328, Hewlett Packard Labs, Bristol, England, November 2002.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: the case of hashing and signing. In Y. Desmedt, editor, *Advances in Cryptography - Crypto '94, Lecture Notes in Computer Science, Vol. 839*. Springer Verlag, New York, 1994.

[4] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementally and at reduced cost. In W. Fumy, editor, *Advances in Cryptography - EuroCrypto '97, Lecture Notes in Computer Science, Vol. 1233*. Springer Verlag, New York, 1997.

[5] J. Carroll. Signing RDF graphs. In *Lecture Notes in Computer Science, volume 2870*. Springer-Verlag, September 2003.

[6] D. Clarke, S. Devadas, B. Gassend, M. van Dijk, and E. Suh. Incremental multiset hashes and their application to integrity checking. In *ASIACRYPT 2003 Conference*, November 2003. (to appear).

[7] K. Eshghi. Intrinsic references in distributed systems. In *IEEE Workshop on Resource Sharing in Massively Distributed Systems, Vienna, Austria*, July 2002.

[8] Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia. *FIPS 180-1: Secure hash standard*, April 1995.

[9] J. Grant and D. Beckett. Resource description framework (RDF) test cases, W3C working draft. http://w3.org/TR/2002/WD-rdf-testcases-20020429, April 2002.

[10] G. Klyne and J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts, November 2002.

[11] R. S. A. Laboratories. RSA Laboratories' Frequently Asked Questions about today's cryptography, version 4.1. http://www.rsasecurity.com/rsalabs/faq/, 2000.

[12] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification, W3C recommendation. http://w3.org/TR/1999/RED-rdf-syntax-19990222, February 1999.

[13] S. Melnik. RDF API draft: Cryptographic digests of RDF models and statements. http://www-db.stanford.edu/~melnik/rdf/api.html#digest, January 2001.

[14] E. Miller et al. Resource description framework (RDF)/W3C semantic web activity. http://w3.org/RDF, April 2002.

[15] R. Rivest. Request for comments (RFC) 1321: The MD5 message digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[16] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.

[17] C. Sayers and K. Eshgi. The case for generating URIs by hashing RDF content. Technical Report HPL-2002-216, Hewlett Packard Laboratories, Palo Alto, California, August 2002.

[18] B. Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., 1996.

[19] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Crypto 2002, Lecture Notes in Computer Science, Vol. 2442*. Springer Verlag, New York, 2002.

[20] D. Wagner. Personal Communication, 2004.

[21] A. L. Zobrist. A hashing method with applications for game playing. Technical Report 88, Computer Science Dept. University of Wisconsin Madison, 1970. reprinted in International Computer Chess Association Journal, Volume 13, Number 2, pp. 69–73, 1990.