# RDF Graph Digest Techniques and Potential Applications

Craig Sayers, Alan H. Karp
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2004-95
May 28, 2004*

RDF, graph, digest, signature, RDF applications, Resource Description Framework

Digests are short digital abbreviations computed from original content. This paper compares existing algorithms for computing the digest of a Resource Description Framework (RDF) graph and discusses potential applications.

Digest computation is complicated by blank node relabeling and statement reordering. We treat these separately: surveying four techniques for handling blank node identity and three algorithms for dealing with statement ordering.

It is possible to avoid the issue of blank nodes by limiting permissible graph operations, placing restrictions on allowable graph content, or modifying the RDF specification. Assuming none of those are reasonable, then practical solutions are still possible but they require adding additional information to the graph.

To deal with statement ordering, the simplest approach is to use a sort with the attending *O(Nlog(N))* runtime. An incremental algorithm is also possible. This is theoretically faster, offering *O(N)* runtime but may be slower in practice for applications which do not benefit from the incrementality.

Graph digests are applicable to digital signatures. Other applications include verifying integrity, detecting changes, generating content-based identifiers, and improving efficiency when accessing remote stores.

# RDF Graph Digest Techniques and Potential Applications.

Craig Sayers and Alan H. Karp
HP Labs, Palo Alto.

May 27, 2004

**Abstract**

Digests are short digital abbreviations computed from original content. This paper compares existing algorithms for computing the digest of a Resource Description Framework (RDF) graph and discusses potential applications.

Digest computation is complicated by blank node relabeling and statement reordering. We treat these separately: surveying four techniques for handling blank node identity and three algorithms for dealing with statement ordering.

It is possible to avoid the issue of blank nodes by limiting permissible graph operations, placing restrictions on allowable graph content, or modifying the RDF specification. Assuming none of those are reasonable, then practical solutions are still possible but they require adding additional information to the graph.

To deal with statement ordering, the simplest approach is to use a sort with the attending $O(Nlog(N))$ runtime. An incremental algorithm is also possible. This is theoretically faster, offering $O(N)$ runtime but may be slower in practice for applications which do not benefit from the incrementality.

Graph digests are applicable to digital signatures. Other applications include verifying integrity, detecting changes, generating content-based identifiers, and improving efficiency when accessing remote stores.

## 1 Introduction

When sending a message between machines it is sometimes desirable to verify that it has not been tampered with. Our task is to generate a digital signature which may be sent to the recipient to enable verification. For text-based messages this is a well-studied problem and there are well-known signature algorithms [14].

Since computing a digital signature is relatively expensive, it is common to compute a short "fingerprint" or "digest" from the message then sign that instead of the original message [1]. Provided it is difficult to find a different message that generates the same digest then a recipient can extract the digest from the signature and compare it to the computed digest of the received message to verify authenticity. (See Schneier [18] for a great introduction). Figure 1 shows an example of signing and sending a message.

Mechanisms are needed for applying similar algorithms efficiently for the case where the message contains information encoded in the Resource Description Framework (RDF). In particular algorithms are needed for computing the digest of an RDF graph.

Note that while we are using digital signatures to introduce the need for graph digests, they have much wider applicability (see Section 5).

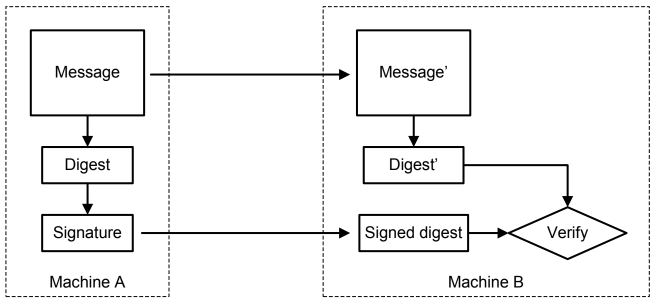

Figure 1: **When sending a message between machines it is sometimes desirable to guarantee that it has not been altered in transit. Since digitally signing an entire message is expensive, it is common to compute a short digest by hashing the message and then signing that digest. At the receiving end, the digest is extracted from the signature and compared with the computed digest of the received message to verify that the message is intact and has not been tampered with.**

## 1.1 RDF Graphs

RDF Graphs [11] encode information about resources. They are labeled directed graphs and may contain cycles. An example of a simple RDF graph containing just two nodes and arcs is shown in Figure 2. In this case the nodes are unlabeled or *blank nodes*.

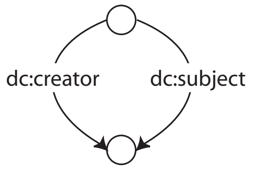

Figure 2: **A simple RDF graph consisting of just two blank (unlabeled) nodes and two arcs.**

To transmit an RDF graph between sites it must be serialized as a list of statements and doing so requires assigning labels to any blank nodes and choosing some ordering for the statements. Each RDF graph thus has many possible serializations. For example, the above graph could be serialized as:

```
_1 dc:subject _2 .
_1 dc:creator _2 .
```

but it could also be serialized as:

```
_2 dc:creator _1 .
_2 dc:subject _1 .
```

Both serializations are equally valid and equivalent (at least as far as any RDF processing engine is concerned).

## 1.2 Direct communication

The simplest case of securely transmitting a graph is to rely on a particular serialization of that graph. In this case, we use the same general message signing procedure shown in Figure 1. Generating a message by serializing the RDF graph, hashing that to form a short digest and then signing that digest. On a receiving machine, the digest can be extracted from the signature and compared with the hash of the received serialization to verify it has not been altered in transit. Then the graph may be reconstructed from that serialization. This is shown in Figure 3. This approach works well, and has been used in practice [2].

The downside is that we've only computed the digest for that particular message. If the RDF is loaded into an in-memory graph then we lose any ability to recompute the digest. Even if we could exactly duplicate the serialization algorithm, the arbitrary labels assigned to blank nodes and the lack of any defined statement ordering precludes recreating the original serialized message.

## 1.3 Graph communication

A more sophisticated solution would allow a graph to be transferred. This avoids the need to remember any particular serialization and also allows the graph to be stored and transported by intermediaries.

We'll begin by considering a "perfect" algorithm. That is, one which generates a digest for arbitrary RDF graphs, places no restrictions on content or operations, and does not modify the graph.

Assuming a perfect RDF canonical serialization algorithm exists, then we can use that to compute a graph digest. At the sending machine the canonical serialization can be hashed to give a digest; while at the receiving end the graph may be passed through the same algorithm and the results compared. Since the serialization is canonical, it should be the same regardless of any changes to the RDF graph caused by the communications channel.

An example of signing a graph using a perfect canonical serialization scheme is shown in Figure 4. This is a simple and appealing solution. Unfortunately, there is unlikely to ever be any fast perfect canonical serialization algorithm. Carroll [6] showed this by noting that canonical serializations can be used to solve graph isomorphism, and thus if there was a fast solution to canonical serialization we could use that to make a fast solution for graph isomorphism. But graph isomorphism is known to be a hard problem for which there is no fast algorithm.

The perfect canonical serialization is complicated by the presence of blank nodes and the lack of any defined statement ordering. Practical algorithms must deal efficiently with both of those.

## 1.4 Overview

In this paper we compare several alternative practical techniques for computing the digest of an RDF graph. In Section 3 we compare four different mechanisms for handling blank node identity. In Section 4 we look at three different algorithms which account for statement ordering. Finally, Section 5 describes some of the many applications for graph digests.

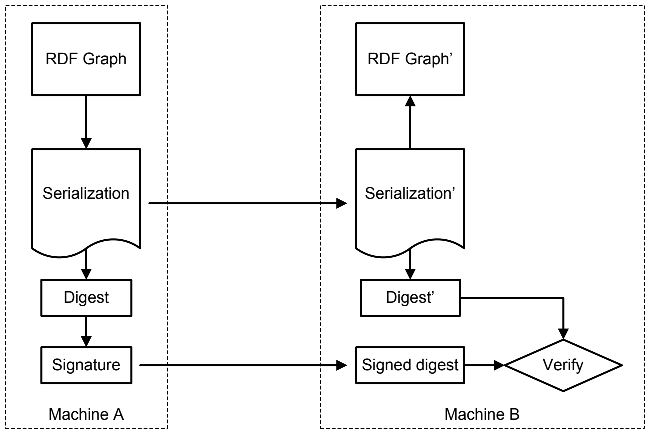

Figure 3: **Using direct communiction we serialize the RDF graph, hash it to form a short digest and then sign that digest. On a receiving machine, the digest can be extracted from the signature and compared with the hash of the received serialization to verify it has not been altered in transit. Then the graph may be reconstructed from that serialization.**

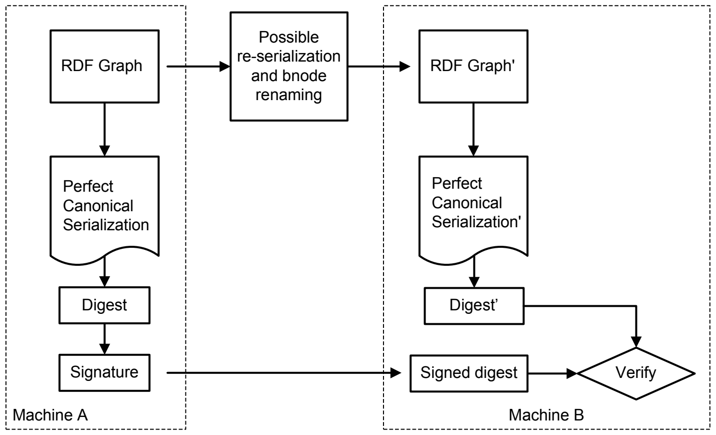
RDF Graph
Perfect Canonical Serialization
Digest
Signature
Machine A
Possible re-serialization and bnode renaming
RDF Graph'
Perfect Canonical Serialization'
Digest'
Signed digest
Verify
Machine B

Figure 4: **Assuming a canonical serialization algorithm exists then it can be used at both the sending and receiving ends to construct identical serializations. This conveniently allows the graph to be transferred through intermediaries who may reorder statements or relabel blank nodes. It also means that we only need to store the graph in order to be able to later verify the signature.**

## 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:

$$\mathcal{D}(S) \equiv \mathcal{D}(T) \ when \ S \not\equiv T$$

## 3 Blank node identity

The key to a practical graph digest is removing the influence of blank nodes. There are four ways to do this and we'll examine each one in turn.

- Limit the operations which may be performed on the graph.
- Limit the graphs for which we support computing digests.
- Modify the graph, adding additional information to aid handling blank nodes.
- Change the RDF specification.

## 3.1 Limit operations

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 digest. 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.

Placing restrictions on intermediaries is clearly not an ideal choice, but it is simple.

## 3.2 Limit graphs

Another solution is to simply avoid the use of blank nodes entirely. This won't always be an option, but if you are designing an ontology and data generator from scratch, and you know you'll need to later sign the generated graphs, then you can take care not to use any blank nodes. This limits what you can express, but for some applications that may be quite acceptable.

## 3.3 Modify the graph

Rather than limiting the operations, or limiting the graphs, we can change the problem. In describing use of the perfect canonical serialization, we assumed that the original graph to be signed must remain unchanged. If we relax that restriction, allowing the graph to be modified, then we can insert additional information into the graph itself and use that to later speed digest computation.

In particular, we can add additional statements to the graph to capture the arbitrary labels assigned to the blank nodes.

There are two alternative approaches to this. One proposed by Carroll [6] is to perform a sophisticated analysis of the graph and add a relatively small number of additional statements. Another approach is to simply add a statement to encode the label assigned to each blank node.

For example, if we had a graph containing the blank node "_1":

> _1 dc:creator "Pat" .
> _1 dc:type "Book" .

Then we could add an additional statement:

_1 myOnt:hasLabel "_1" .

After passing though intermediaries, the blank node labels may change, so a receiving machine may see the graph:

_42 dc:creator "Pat" .
_42 dc:type "Book" .
_42 myOnt:hasLabel "_1" .

By extracting the blank node label statements, it may use those to relabel the blank nodes, conveniently returning the graph to the same form used by the sender:

_1 dc:creator "Pat" .
_1 dc:type "Book" .
_1 myOnt:hasLabel "_1" .

Using this approach we can guarantee the blank node labels at the sending and receiving machines match without needing to restrict the content of the graph, or the serializing/deserializing that may be performed by intermediaries.

The additional statements are of course not without cost. The extra statements consume space and require time to process. Nevertheless, those costs are reasonable when compared to the time which would be required if we didn't add the additional statements.

## 3.4 Change the specification

A final approach, which we mention for sake of completeness, would be to change the specification. Specifically, if blank nodes were assigned globally-unique identifiers, and those identifiers were required to be maintained in all serializations then there would be no need to handle blank nodes specially.

This is not a large leap. The N-triples syntax [10] and the most recent RDF/XML syntax [3] already provide the means to encode blank node labels. The only required changes are to assign the identifiers a globally-unique name (such as a UUID [12]) and require any reader to preserve the identifiers.

Having global labels would also have the desirable property that any two graphs could be merged without needing to first perform blank node relabeling.

### 3.5 Comparison

By limiting the operations, the graphs, or modifying the specification, then it is possible to avoid the issue of blank nodes entirely without any performance penalty. If that is not possible, then it is necessary to add additional information to the graph.

If we use one additional statement per blank node, then 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). Note that in practice, if the blank nodes are assigned incremental numeric labels then a lookup table can be employed and each lookup can be performed in constant time, however, if the blank nodes are assigned arbitrary text labels then some form of hash map will be needed and the possibility for hash collisions means that it takes only approximately constant time for each lookup.

## 4 Graph digest algorithms

Having dealt with blank nodes, the remaining complexity in computing a graph digest is the lack of any statement ordering. To solve that there are three approaches and again we'll look at each in turn.

### 4.1 Hash-Sort

The approach used by Melnik and Dunham [13] is to compute a hash for each statement, sort the hashes, concatenate them, and then hash the result:

$$\begin{aligned}
d_i^s &= hash(s_i) \\
d_i^p &= hash(p_i) \\
d_i^o &= \begin{cases} hash(o_i) << 8 & \text{if } o_i \text{ is a literal} \\ hash(o_i) & \text{otherwise} \end{cases} \\
d_i &= hash(concat(d_i^s, d_i^p, d_i^o)) \\
digest &= hash(concat(sort(d_1, d_2 \ldots d_N)))
\end{aligned}$$

where $hash()$ is a hashing function (this takes an input of arbitrary length and generates an $n$ bit output) and $<<$ is a bitwise left-shift.

## 4.2 Sort-Hash

The approach used by Carroll [6] is to serialize each statement, sort them, concatenate them together, and then compute a hash of the result:

$$\begin{aligned} u_i &= \mathit{serialize}(s_i, p_i, o_i) \\ \mathit{digest} &= \mathit{hash}(\mathit{concat}(\mathit{sort}(u_1, u_2 ... u_N))) \end{aligned}$$

where *serialize*() 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.

## 4.3 Incremental digest

In both the preceding approaches statement ordering was handled via a sort. In [16] we showed the sort could be avoided by using a set hash [4, 5, 20, 8]. We first modify the graph to permit blank node relabeling and then compute a set hash, treating each statement as one member of the set:

$$\mathit{digest} = \bigodot_{i=1}^{N} \mathit{hash}(\mathit{serialize}(t_i))$$

where $\odot$ is a combining operation which is both associative and commutative to support incremental operation. One convenient but relatively insecure function is addition modulo $2^n$, where $n$ is the number of bits generated by the *hash* function.

## 4.4 Comparison

### 4.4.1 Security

When considering security of the digest, we consider the task for an adversary attempting to find a different graph which has the same digest.

In the case of both hash-sort and sort-hash, the security is almost entirely dependent on the choice of the hashing function. Existing functions, such as SHA-1 [18, 9] appear to be a good choice.

In the case of the incremental digest, the security depends on both the hashing function and the combining function. If the hashing function is again SHA-1 and the combining function is addition modulo $2^{160}$ then security is acceptable for some applications but may be inadequate against a determined attacker [19]. Better security is possible, but only by using more time-consuming combining functions or a much longer randomizing function.

#### 4.4.2 Time complexity

All of the algorithms need some accommodation for blank nodes, and all can use the same technique.

If we assume some upper bound on the length of URIs and literals then both $serialize()$ and $hash()$ take constant time per statement.

For both hash-sort and sort-hash, the runtime is dominated by the need for the sort and is hence $O(Nlog(N))$.

For the incremental digest, assuming the combining operation may combine two hashed blocks in $O(1)$ time, then the algorithm takes $O(N)$ time. This is better in theory, however in practice it may be more expensive, especially if a high level of security is required.

For incremental computation, the incremental digest is significantly faster, even with very high security, since it can add a new statement in constant time, while the other algorithms all require recomputing a hash over all the statements (in the best case) and also redoing the sort (in the worst case).

## 5 Applications for Graph Digests

While we have focused on digital signatures, the general notion of a graph digest has much wider applicability. Think of the digest as an abbreviation for an instance of a graph.

### 5.1 Graph identity

In the present specification there is no concept of graph identity and hence no way to make statements about a graph. If graph identity were to be supported in the future, and it was possible to mark graphs as being closed (see [7] for a proposed serialization which would support this), then the digest of a closed graph could serve as its identifier. This would conveniently allow graphs to be compared simply by comparing their identifiers. Such comparisons are obviously merely for equality not isomorphism but they are nevertheless still useful.

## 5.2 Verification

One property of the digest is that you can verify data integrity. For example, a small device sending a stream of RDF information to a remote store can compute a running digest for later use in verifying that all the data arrived intact and has not been tampered with.

## 5.3 Secrecy

Since the digest is one-way, and serves as an abbreviation for the entire graph, you can use it in places where secrecy is desired. For example, if purchasing a book, the bookstore agent and your agent can share a graph, then the digest of that graph can be used in subsequent three-way conversations with a credit card company. The credit card company then doesn't know which book you purchased but it has enough information that you could later prove if the book you received did not match the one you paid for.

The incremental digest provides additional opportunites for secrecy since it permits a graph to be updated and the digest recomputed without requiring access to all the original statements in the graph. For example, if a graph stores transactions you can add a new transaction and update the graph digest without needing to know details of all the preceding transactions.

## 5.4 Detecting changes

There are applications where it is desirable to know if a graph has changed. To see why, consider interacting with a graph on a remote database. A common operating pattern is:

1. Query the graph, returning query result
2. Perform some computation based on the query result
3. Update the graph

The obvious difficulty is that the graph may change between the time you query and the time your update to the graph arrives back at the server. You could avoid that by locking, but that's expensive.

If the remote server continuously maintains a digest of the graph, then you can instead do:

1. Query the graph and its digest, returning query result and digest, $D$.

2. Perform some computation based on the query result
3. Send atomic command "if the digest is still $D$, update the graph, otherwise return an error."

In that way, you can guarantee that your operation is performed on a graph which has exactly the state you anticipated and there was no need to lock the graph.

Since we're only interested in detecting changes, it is possible to largely avoid the issue of blank nodes. So long as the blank nodes are assigned some label, and that label is available for use in the digest computation, then there is no need for any relabeling or the addition of extra statements. This provides a conservative indication of change, since any modification (even the harmless renaming of blank nodes) will cause the digest to change.

### 5.5 Finer granularity

Digests may also be applied to subsets of a graph. One particular example is to compute a digest for each unique subject node in a graph [15, 17].

## 6 Conclusions

The simplest approach to signing RDF is to serialize the RDF in a message and sign that particular message.

The task of signing a graph, rather than a particular message containing the graph, requires computing a graph digest. Since a fast perfect canonical serialization algorithm does not exist, we have examined techniques for developing practical algorithms. These use two phases: first removing variations in blank node identity and then using either a sort or an incremental digest to handle variations in statement ordering.

The described graph digest algorithms have wider applicability than just for digital signatures. They may be used to generate content-based identifiers for RDF graphs, to aid in security, to verify integrity, and to improve efficiency when remotely accessing graph stores.

## 7 Acknowledgements

Thanks to Kave Eshghi for a number of great discussions about content-based identifiers and to Kevin Wilkinson for suggesting a number of improvements to an early draft.

## 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] D. Becket. RDF/XML syntax specification (revised), W3C recommendation. http://w3.org/TR/1999/REC-rdf-syntax-grammar-20040210/, February 2004.

[4] 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.

[5] 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.

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

[7] J. Carroll and P. Stickler. RDF triples in XML. Technical Report HPL-2003-268, Hewlett Packard, 2003.

[8] 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).

[9] 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.

[10] 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.

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

[12] P. J. Leach and R. Salz. UUIDs and GUIDs. http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt, February 1998.

[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] 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.

[15] 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.

[16] C. Sayers and A. Karp. Computing the digest of an RDF graph. Technical Report HPL-2003-235-R1, Hewlett Packard Laboratories, Palo Alto, California, November 2003.

[17] C. Sayers and K. Wilkinson. A pragmatic approach to storing and distributing RDF in context using snippets. Technical Report HPL-2003-231, Hewlett Packard Laboratories, Palo Alto, California, November 2003.

[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] 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.