A Content Integrity Service For Long-Term Digital Archives

Stuart Haber, HP Labs, New Jersey U.S.A.
Pandurang Kamat, Rutgers University, New Jersey U.S.A.

Abstract

We present a content integrity service for long-lived digital documents, especially for objects stored in long-term digital archives. The goal of the service is to demonstrate that information in the archive is authentic and has not been unintentionally or maliciously altered, even after its bit representation in the archive has undergone one or more transformations. We describe our design for an efficient, secure service that achieves this, and our implementation of the first prototype of such a service that we built for HP's Digital Media Platform. Our solution relies on one-way hashing and digital time-stamping procedures.

Our service applies not only to transformations to archival content such as format changes, but also to the introduction of new cryptographic primitives, such as new one-way hash functions. This feature is absolutely necessary in the design of an integrity-preserving system that is meant to endure for decades.

Introduction

Information in a digital archive can include complex multipart documents. In a long-term archive these documents may be expected to undergo multiple transformations during their lifetime, including, for example, format changes, modifications to sub-parts and to accompanying metadata. Skeptical users of a digital archive may desire, or in some case may be legally required, to verify the integrity of records that they have retrieved from the archive.

All typical algorithmic techniques for verifying the integrity of a digital object begin with a representation of the object in question as a sequence of bits. When digital objects are transformed in any nontrivial way, their bit representations are changed as well, so that these algorithmic techniques no longer apply to the transformed object. In fact, it is the usual aim of a cryptographic technique for proving integrity that it ”fail”—more precisely, that it correctly succeed in proving lack of integrity—when even a single bit in the object's representation is changed.

In this work we describe an efficient and secure Content Integrity Service (CIS) that solves this problem, which we designed and implemented as a service on the Digital Media Platform (DMP) [1].

Background

The basic building blocks of our solution are cryptographic hash functions and time-stamping procedures. Throughout this article we refer to the objects of concern in a digital archive or repository simply as "documents".

Hash functions

A *cryptographic hash function* is a fast procedure H that compresses input bit-strings of arbitrary length to output bit-strings (called *hash values*) of a fixed length, in such a way that it is computationally infeasible to find two different inputs that produce the same hash value. (Such a pair of inputs is called a *collision* for H.) For any digital document x, its hash value $v = H(x)$ can be used as a proxy for x, as if it were a characteristic "fingerprint" for x, in procedures for guaranteeing the bit-by-bit integrity of x [2].

Time-stamping

A digital time-stamping scheme is a procedure that solves the following problem: given a digital document x at a specific time t, produce a *time-stamp certificate* c that anyone can later use (along with x itself) to verify that x existed at time t. Certificates that will pass the verification test should be difficult to forge [3]. There are two different families of time-stamping algorithms, those using digital signatures (hash-and-sign) and those based entirely on cryptographic hash functions (hash-linking).

In what is sometimes called a *hash-and-sign* time-stamping scheme, the time-stamp certificate for a document consists of a digital signature computed by a Time-Stamping Authority (TSA) for the document and the time of signing. In practice the TSA usually digitally signs the hash of the document rather than the document itself, and hence the name. This has two major drawbacks as a tool for long-term archives:

(1) It requires the assured existence of trustworthy archived key-validity history data, in order to check the validity of the TSA's public key. It is a problem for any TSA to manage such a key-validity database over extended periods of time, let alone integrating it with currently deployed commercial PKIs (public-key infrastructures). (See [4] for a proposed solution.)

(2) The trustworthiness of the certificate depends entirely on an assurance that the TSA's private signing key has never been compromised. This is an unacceptable premise for long-term archives. The combination of increasing computational resources with advances in cryptanalytic techniques can be expected to render current digital-signature algorithms ineffective and susceptible to attacks. More simply, the private key of a TSA may leak or be stolen. Either way, an adversary would have the ability to produce certificates for any document, with an arbitrary claimed time, past or future.

For the CIS, we chose a time-stamping technique called *hash-linking*. In this technique, the hash value of the document to be time-stamped is combined with other hash-values received during the same time period to create a *witness hash value*. This witness value is then stored by the TSA or published as a widely witnessed event. This kind of linking makes it computationally infeasible for an adversary to back-date a document, since that would entail computing hash collisions for the witness values (or their hash preimages). This technique relies only on the collision-resistance properties of hash functions, and does not have any secrets or keys that need to be securely protected over extended periods of time [5, 6, 7].

In one implementation of hash-linking, the witness hash values themselves are linked in a hash chain, and hash values within each time period are combined using a Merkle hash tree [8]. For example, Figure 1 illustrates this process for an interval during which the requests $y_1, \ldots, y_4$ were received. In this diagram, H_{12}

$$\begin{tikzcd}
W_{i-1} \arrow[r, bend right] & W_i & & & \\
& H_{14} \arrow[u] & & & \\
& H_{12} \arrow[u] & & H_{34} \arrow[ull] & \\
& y_1 \arrow[u] & y_2 \arrow[ul] & y_3 \arrow[u] & y_4 \arrow[ul]
\end{tikzcd}$$

Figure 1. *Hash-Linking using a Merkle hash tree*

is the hash of the concatenation of y_1 and y_2, H_{14} is the hash of the concatenation of H_{12} and H_{34}, and similarly for the other nodes, and W_i and W_{i-1} are the respective witness hash values for the current and the previous intervals. The time-stamp certificate for the third request (the one containing hash value y_3), for example, is $[t_i; (y_4, R), (H_{12}, L), (W_{i-1}, L)]$, where t_i is the time of the current interval. One validates the claim that this is a correct time-stamp certificate for a digital document by hashing the document, taking the resulting hash value (presumably y_3), combining it on the right with y_4 from the certificate, and hashing the concatenation; taking the resulting hash value (presumably H_{34}) and combining it on the left with H_{12}, and so on, finally obtaining a putative witness value. This value is then checked against the published witness value, W_i, that is associated with the time t_i.

Design of the Content Integrity Service

The Content Integriy Service is only one piece of the daunting engineering project of designing a large-scale long-lived digital archiving system [9].

The essence of our solution is to use a secure digital time-stamping system, first to time-stamp every document at ingestion into the archive, storing the resulting time-stamp certificate in the archive with the document; and second to produce an auditable record of every transformation to a document in the archive, in such a way as to verifiably link the time-stamp certificate for the transformed version of the document to its original form. Let us first look at why we need renewable integrity certificates and how to produce an auditable transformation history.

Renewing integrity certificates

In order to explain our solution, we first describe the process of "renewing" digital time-stamps [6]. The need for this is motivated by the fact that—as noted above for the particular case of keys for digital signatures—with advances in computational power and resources, as well as the discovery of entirely new cryptanalytic algorithms, particular instances of cryptographic primitives that were secure when they were first deployed may become insecure several years later. In the last couple of years, the cryptographic community has been surprised by powerful new attacks on the hash functions MD5 and SHA-1, among others [10, 11]. The question of how best to introduce a new and presumably more secure hash function into a system that now uses an older hash-function design that may soon be subject to devastating compromise is no longer the merely academic question it was when it was first raised (and incorrectly solved) by the authors of [5].

Suppose that an implementation of a particular time-stamping system is in place, and consider the pair (d, c_1), where c_1 is a valid time-stamp certificate (in this implementation) for the bit-string d. Now suppose that some time later an improved time-stamping system is implemented and deployed—by replacing the hash function used in the original system with a new hash function, or even perhaps after the invention of a completely new algorithm. Is there any way to use the new time-stamping system to buttress the guarantee of integrity supplied by the certificate, c_1, in the face of potential later attacks on the old system?

One could simply submit d as a request to the new time-stamping system; but this would lose the connection to the original time of certification. Another possibility is to submit c_1 as a request to the the new time-stamping system. But that would be vulnerable to the later existence of a devastating attack on the hash function used in the computation of c_1, as follows: if an adversary could find another document d' with the same hash value as d, then he could use this renewal system to back-date d' to the original time.

Suppose instead that the pair (d, c_1) is time-stamped by the new system, resulting in a new certificate c_2, and that some time after this is done (i.e. at a definite later date), the original method is compromised. The certificate c_2 provides evidence not only that the document contents d existed prior to the time of the new time-stamp, but that it existed at the time stated in the original certificate, c_1; prior to the compromise of the old implementation, the only way to create a valid time-stamp certificate was by legitimate means.

Auditable transformation history

Now suppose that we are interested in the long-term preservation of a particular digital document. For this description, suppose that we are only interested in enabling the authentication of the entire document (as opposed to making this possible as well for parts of the document). In its original form, let d denote the bit-string representation of the document in file format f, and let us suppose that d is time-stamped at time t, with resulting time-stamp certificate c. We will write $c = TS(d;t)$ to indicate that the certificate is for input consisting of the document d, and it was computed at time t.

Now suppose that at some later time t', it is decided to make a format change to format f', using a particular conversion or migration procedure. Let d' denote the bit-string representation of the resulting document. Simply computing a new time-stamp certificate for d' would lose the connection between d', the new representation or rendition of the document, and its original version. The aim rather is to memorialize—and enable later verification of—this format conversion, while preserving the assurance of integrity all the way back to that of the original form of the document. We can do this by adapting the procedure for renewing time-stamps described above. Let i denote a standard format for describing an invocation of the migration procedure used to convert from format f to format f', perhaps including file-names and other useful meta-data for input and output files d and d', respectively. Then, immediately after performing the conversion, a new time-stamp request for $[d, d', i, c]$ is submitted, and the resulting time-stamp certificate $c' = TS(d, d', i, c; t')$ is stored with the document in the archive. The new certificate *c'* can be used to verify the integrity of the latest form of the original document.

Assuming the integrity of i as a description of the transformation, the only way to compromise the security of our solution is to compute collisions for the hash functions that we use.

Document Transformations

The description of our algorithm in the previous section was couched in terms of a simple transformation to a single entire document. But variations on the same technique can be applied to complex transformations to one or more pieces of a multi-part document, including format conversions, annotations, additions of metadata, and later modifications of the document. Naturally, transformations can follow one another, and each one can be certified by a CIS certificate. Transformations include:

- *Business workflow:* A document may undergo several transformations one after the other as part of a business workflow. Each step of the workflow instance can be regarded as a single transformation that CIS can certify.
- *Document redaction:* Sensitive parts of a document may be redacted (i.e. removed or blacked out) before it is made public, for reasons of security, privacy, or protection of trade secrets. By applying CIS, the integrity of the redacted version can be linked to the integrity of the original.
- *Integrity metadata:* The process of renewing a document's time-stamp certificate as described above can be regarded as a special case of CIS, as applied to a particular modification of an item of metadata corresponding to the document (namely, its time-stamp certificate). But similar logic applies to other sorts of accompanying metadata related to integrity, including digital signatures, public-key certificates, and key-validity information such as CRLs (certificate revocation lists) and the like.

A particularly interesting case is provided by the extraction of subdocuments from a complex multi-part document that can be represented hierarchically in a graph. (For a concrete example, the US Congressional Record is a sequence of "volumes", each consisting of "daily issues" that in turn contain "sections" containing "items of business"; a typical item of business is a Senator's speech, which might be broken up into paragraphs.) By an algorithmic technique analagous to the Merkle-tree method illustrated in Figure 1 above (and described in greater detail below), it is possible to combine hash values for individual subdocuments so as to compute a single summary hash value that depends on the entire document. Furthermore, the hash value for any subdocument can be linked to the summary hash value by a succinct list of hashing instructions; this list can be used to extend a CIS certificate for the entire document (represented by its summary hash value) to form a CIS certificate for the subdocument.

Returning to the example of the Congressional Record, such an extracted subdocument could be a single paragraph of a Senator's speech, or it could be an entire daily issue. In any case, the act of compiling a collection of subdocuments extracted from the archive, each accompanied by its extended integrity certificate, is yet another well-defined transformation to which the CIS applies.

If this compilation is chosen in response to a query to a suitably designed search engine for the archive, such a CIS certificate can even be computed on the fly to accompany the response to the query.

Note that it is not necessary for the archivist to choose, at the moment a document is ingested into the archive, exactly how it will turn out to be convenient later to break up the document into subdocuments. This choice can be made later, at which point the computation of hash values for subdocuments and their grouping into an appropriate hierarchical structure is simply another well-defined transformation.

In all of these cases, CIS can be used to verifiably prove that the new document was produced by applying the particular transformation(s) to the original document(s). CIS does not put any restriction on the way transformations are represented in the system, so long as the representation is consistent.

The CIS can also be useful outside the context of a large digital archive. To give one example, it is common practice in discovery requests in litigation in the US court system that large batches of Microsoft Office files and email messages are converted to tiff image files before being turned over to opposing counsel. CIS could be used to buttress the integrity of these files, especially (but not exclusively) in cases where the original files were signed or time-stamped in their original format.

Prototype Implementation

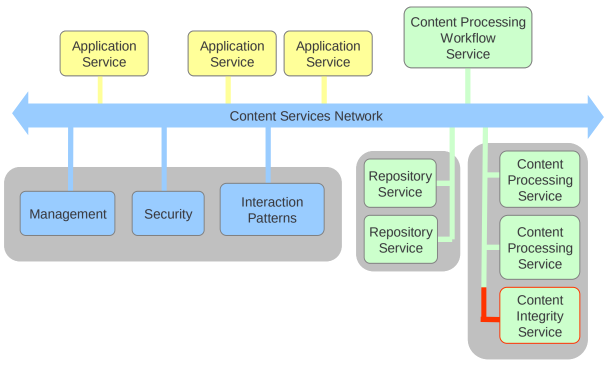

Figure 2. *Content Integrity Service on HP Digital Media Platform*

We have implemented a prototype of the Content Integrity Service within HP's Digital Media Platform (DMP). This is a modular, service-oriented architecture that was designed as a platform for building and maintaining an enterprise-scale suite of content storage and processing operations [1]. DMP presents an XML-based interface for service interaction, and the CIS is implemented as a service within this framework, as shown in Figure 2.

As we implemented the CIS in DMP, it handles complex documents stored in the DMP Repository, and consists of the following set of operations:

- *Certify* operates on documents alone, producing for any document a time-stamp certificate and storing it with the document in the Repository.

- *Validate* takes a document and its integrity certificate (in the format in which they would be stored in the Repository, and checks the validity of the certificate for the given document.
- *Transform-certify* operates on a document and a transformation, expressed as described below, and computes the corresponding CIS certificate.
- *Transform-validate* is used to check the validity of a transformed document and its CIS certificate.

CIS was built as a proof of concept, but we did not integrate it into a full document management system.

In the DMP architecture, transformations on content are expressed as *workflow instances* that can be serialized in XML. Our *Transform-certify* and *Transform-validate* operations use this XML representation of the workflow instance as their standard format for describing a transformation. In our notation above, this is the invocation *i* of the transformation.

In principle, an instance of the CIS can use any time-stamping service that is available, and even with a digital-signature system (preferably one with a well-engineered PKI). Our prototype was built so that it could make calls to *certify* and *validate* functions provided by any service. We chose to use the time-stamping service provided by Surety, LLC, whose hash-linking technique is the preferred method for long-lived documents [12].

In 2004, when we built our prototype, Surety's service used MD5 and SHA-1, evaluated in parallel, as its hash function. Since then, in light of recent attacks on both of these functions, Surety's service uses SHA-256 and RIPEMD-160 (also evaluated in parallel), and offers the renewal capability desribed above for records that were originally time-stamped with the older version of the service.

Graph data model for documents

DMP stores all content in a repository that models its data as a directed graph. According to this DMP Repository Abstraction, a document is a *graph* whose points are *nodes*, *resources*, and *literals*, joined by labelled directed edges called *properties*. All documents are stored as graphs. Nodes are typically used to representative hierarchical structure within a complex multi-part document. Components of the document are represented as resources, which are leaves in the graph. Literals are strings that may be used to represent metadata. Properties belong to a node, linking it to other nodes, resources or literals. Nodes and resources are named using Uniform Resource Identifiers (URIs). Figure 3 shows a simple graph consisting of a single node N, joined by property p to resource R and by property q to literal L.

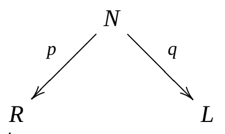
$$\begin{tikzcd} & N \arrow[dl, "p"'] \arrow[dr, "q"] & \\ R & & L \end{tikzcd}$$

Figure 3. A simple graph

The DMP Repository Abstraction is especially convenient for handling the integrity metadata used by the CIS. When a time-stamp or CIS certificate is computed for a document d, the service constructs an *auth-node* for d, linked via its `auth` property to d, via its `hash` property to d's hash value, and via its `cert` property to the certificate. Figure 4 shows the result of applying the CIS to the document represented by node N in Figure 3.

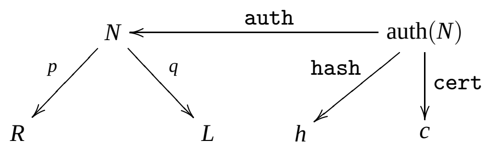
$$\begin{tikzcd} & N \arrow[dl, "p"'] \arrow[dr, "q"] & & & \text{auth}(N) \arrow[llll, "\texttt{auth}"'] \arrow[dl, "\texttt{hash}"'] \arrow[d, "\texttt{cert}"] \\ R & & L & h & c \end{tikzcd}$$

Figure 4. The graph of Figure 3, certified

Computing the hash value of a graph

For our implementation of CIS, a document is represented by a graph (or subgraph) in the DMP Repository. Specifically, a document is named by giving the URI of a node in the repository, and the document consists of the subgraph spanned by a breadth-first search of the repository beginning at the given node.

To compute a hash value for a document, we need to compute hash values for every node or resource in the document, mirroring the structure of the graph as we do so. Our algorithm is inspired by the sort-hash approach described in [13]. Pseudo-code for the two main functions that make up our algorithm is shown as Algorithm 1 and Algorithm 2 below.

```
input  : URI uri
output: Hash as a byte array
if (uri.type == RESOURCE) then
    return ResourceHash(uri);
else
    return NodeHash(uri);
end
```

Algorithm 1: Function URIHash

```
input       : URI nodeURI
output      : Hash as a byte array
description: Hash the set of (property,URI) tuples that make up
             the node
buffer = NodeMapHash (URI nodeURI);
foreach property in lexicographic order do
    if (the property points to a URI) &&
    (this URI has not been hashed before) then
        buffer = concat(buffer, URIHash(URI));
    end
end
return buffer;
```

Algorithm 2: Function NodeHash

The URIHash function simply checks to see whether the URI points to a resource or a node. If it is a resource, then we simply hash the bit-string representation of the resource. If the URI points to a node, then we call the *NodeHash* function to hash the hierarchical structure in the repository with this node as its root. The *NodeHash* function first hashes the lexicographically ordered set of (propertyname, uri) pairs that is part of the node. It then begins to traverse the graph, following the edges given by the outgoing properties of the node. Specifically, our *NodeHash* algorithm hashes the graph in a lexicographically ordered depth-first traversal, beginning at the root, to recursively hash all the nodes, resources and literals that form the graph. At each level, intermediate hash values capture the bit-string representation of the resources and literals as well as the structure of the graph itself. The *NodeHash* algorithm avoids any cycles while traversing the graph. The hash value and the resulting time-stamp certificate are stored in the repository, linked to the document graph.

We illustrate our algorithm by showing what it computes, given the graph shown in Figure 5. We start with the URI of N_1.

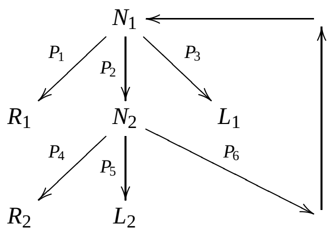

Figure 5. *A cyclic graph*

Since the URI points to a node, we invoke the *NodeHash* function and mark the node as “traversed”. The first step in *NodeHash* is to serialize and hash the property-map of the node using the *NodeMapHash* function. As part of this step, the literal L_1 iss hashed along with the property P_3; this is because the literal is stored as a part of the node N_1 and not stored as a separate entity in the DMP Repository. Then we traverse the document graph, by lexicographic order of properties. (Let’s say that $P_1 < P_2 < P_3$, and $P_4 < P_5$.) Since P_1 points to a resource R_1, we simply hash the bit-string representation of R_1. Next, we examine property P_2 and find that it too points to a node, N_2. Hence the function *NodeHash* is called recursively on node N_2. In this step, the resource R_2 and literal L_2 are handled in the same way as were R_1 and L_1, respectively, when we visited N_1. By contrast, by following the edge with property P_6, pointing to node N_1, the function detects that node N_1 has been traversed and therefore does not invoke *NodeHash* on it again. However, we have captured the fact that node N_2 does have a property P_6 that points to N_1, as part of the computation of *NodeMapHash* on N_2. Thus no information is lost in this process.

The complete calculation of NodeHash(N_1) is shown here:
$NodeHash(N_1)$
$= Hash(NodeMapHash(N_1)||Hash(R_1)||NodeHash(N_2))$

$NodeMapHash(N_1) \quad = \quad Hash([P_1,r_1],[P_2,n_2],[P_3,L_1])$, where $r_1 = URI(R_1), n_2 = URI(N_2)$, etc.

$$NodeHash(N_2) = Hash(NodeMapHash(N_2)||Hash(R_2))$$

$$NodeMapHash(N_2) = Hash([P_4,r_2],[P_5,L_2],[P_6,n_1])$$

Given all these pieces, our *Transform-certify* operation works as follows. As mentioned above, DMP transformations are represented in XML. An invocation of *Transform-certify* takes as inputs the URI d and d' for the original and transformed documents, the XML representation t of the transformation, and and the auth-node a of the original document. The CIS certificate is a time-stamp certificate computed for a request consisting of the hash value h computed as $h =$ $Hash[NodeHash(d)\,|\,NodeHash(d')\,|\,NodeHash(a)\,|\,Hash(i)]$.

It is clear that the same technique can be used to compute a hash value, and then a CIS certificate, for any digital object that is constructed according to an edge-labeled graph data model similar to that used in W3C’s Resource Description Framework [14]. Recent applications of such data models include work in biological research [15].

Conclusion and Future Work

In this paper we presented the design requirements for a Content Integrity Service for long term digital archives. We then described the architecture of such a service as a solution and discussed the implementation of a protoype version of the same. This imlementation uses the highly modular and extensible repository and content services framework provided by HP’s Digital Media Platform.

Our prototype of the CIS can be extended to implement the capability (1) to renew integrity certificates for documents; (2) to certify parts of a document, using the certificate of the main document; (3) to handle complex versioning scenarios for the repository; and (4) to process optional attributes that may be associated with node properties in the future.

References

[1] “HP Digital Media Platform White Paper,” Available at `http://www.hp.com/.`, 2006, To appear.

[2] Alfred Menezes, Paul van Oorschot, and Scott Vanstone, *Handbook of Applied Cryptography*, chapter 9, CRC Press, 1996.

[3] Stuart Haber and Henry Massias, “Time-stamping,” in *Encyclopedia of Cryptography and Security*, H.C.A. van Tilborg, Ed. Springer, 2005.

[4] Petros Maniatis and Mary Baker, “Enabling the archival storage of signed documents,” in *FAST ’02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[5] Stuart Haber and W. Scott Stornetta, “How to time-stamp a digital document,” *Journal of Cryptology*, 1991.

[6] Dave Bayer, Stuart Haber, and W. Scott Stornetta, “Improving the efficiency and reliability of digital time-stamping,” *Sequences II: Methods in Communication, Security and Computer Science, Springer-Verlag*, 1993.

[7] J. Benaloh and M. deMare, “Efficient broadcast time-stamping,” Tech. Rep. TR-MCS-91-1, Clarkson University Department of Mathematics and Computer Science, 1991.

[8] R. Merkle, “Protocols for public key cryptosystems,” in *Proceedings of the 1980 Symposium on Security and Privacy*. 1980, pp. 122–133, IEEE Computer Society Press.

[9] R. Sproull, H. Besser, J. Callan, C. Dollar, S. Haber, M. Hedstrom, M. Kornbluh, R. Lorie, C. Lynch, J. Saltzer, M. Seltzer, and R. Wilensky, *Building an Electronic Records Archive at the National Archives and Records Administration: Recommendations for a Long-term Strategy*, National Archives and Records Administration, 2005.

[10] Xiaoyun Wang and Hongbo Yu, “How to break MD5 and other hash functions,” *EUROCRYPT*, 2005.

[11] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, “Finding collisions in the full SHA-1,” *CRYPTO*, 2005.

[12] “Surety,” `http://www.surety.com`.

[13] J. Carroll, “Signing RDF graphs,” *Lecture Notes in Computer Science, volume 2870, Springer-Velag*, 2003.

[14] “Resource description framework (RDF),” *http://www.w3.org/RDF/*.

[15] Frank Olken, “Biopathways graph data manager (bgdm),” *http://hpcrd.lbl.gov/staff/olken/graphdm/graphdm.htm*.

Author Biography

Stuart Haber is a researcher at Hewlett-Packard Laboratories, specializing in cryptography and computer security, with a particular interest in problems associated with the integrity of digital objects. Before joining HP, he worked at Bellcore (now Telcordia), Surety (which he co-founded), and InterTrust STAR Lab.

Pandurang Kamat is a Ph.D. student in the Computer Science department of Rutgers University specializing in security and privacy research.