RJ 103335 ((95210) December 6, 2004
Computer Science

IBM Research Report

Mirrors and Authenticity: an Enhanced Protocol for Content Delivery

Roberto Bayardo

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Jeffrey Sorensen

IBM Research Division
Watson Research Center
Yorktown Heights, NY

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Mirrors and Authenticity: an Enhanced Protocol for Content Delivery

Roberto Bayardo
IBM Almaden Research Center, San Jose, CA
bayardo@almaden.ibm.com
Jeffrey Sorensen
IBM Watson Research Center, Yorktown Heights, NY
sorenj@us.ibm.com

ABSTRACT

News events or other spontaneous occurrences can result in high demand for web servers. For example, the availability of a popular new software release can swamp the serving capacity of a company. To compensate, the internet has evolved a number of different strategies to help mitigate these problems. Web caching and load-balancing are widely used techniques to help build redundancy and distribute load. Companies such as Akamai provide these services to third parties, and the practice of "mirroring" content, such as is typically done for software such as the Linux distribution, has grown organically as individual sites copy and publish large repositories of software or other digital content.

All of these techniques, intended to provide faster and more reliable access to content, suffer from a lack of authentication. For mirror servers, many users have adopted the custom of computing and publishing an authentication "signature" of the content in an effort to detect tampering. This practice is less than ideal, as the distribution of these digital signatures themselves suffers from the same breakdowns of trust as the original content. In addition, distributions can consist of thousands of files, and attempting to check all of these signatures can create a new load on already overburdened servers of the content originators.

We propose and demonstrate a system to allow automatic authentication of web content that is transparent to the user when incorporated into web browsers and other clients. Through simple extensions to several already deployed web protocols, publishers can provide a single signature that can be used to selectively verify any portion of the content on a web server, regardless of who is hosting the content. We discuss the additional, but small, burdens that providing these authentication features will require of publishers, mirror providers, and clients.

Keywords

Web Servers; Mirror; Merkle Tree; Authentication; HTTP; TLS

1. INTRODUCTION

In the fall of 2003, an attempt was made at incorporating a security flaw into the Linux kernel [1]. The change was made in the main CVS repository used to distribute the kernel source code. Because the Linux kernel developers use a different version control system to store the master copy of the kernel, and this repository uses a digital signature scheme to verify the integrity of the files it contains, the changes were quickly detected and removed. Earlier the same year, the Free Software Foundation reported that the primary file servers for the GNU project had been compromised for many months [2]. In this case, although there was never any evidence of code tampering associated with this incident, uncomfortable doubts lingered and resulted in changes to project administration.

Twenty years ago, Ken Thompson's cautionary tale [3] of incorporating Trojans into the Unix login system warned that "no amount of source-level verification or scrutiny will protect you from using untrusted code." Both the GNU project and the Linux kernel represent some of the most important infrastructure upon which the World Wide Web is built. This software is distributed both in source and compiled forms around the world to millions of users through the use of mirrors. The lack of automatic authentication methods for the distribution of files on the world wide web is part of this trust equation, and in the place of a formal mechanism a number of *ad hoc* practices have evolved and will be discussed in the next section.

Despite its many successes, our web infrastructure still suffers continued attacks. Many of these attacks involve compromises of the software that comprises the web itself. For example, attackers continue to exploit security flaws allowing them to alter or otherwise compromise the content on web servers. The "Web Defacement" attack is a typical example of content being altered, often to the public embarrassment of the publisher. While large organizations such as the *New York Times* are now more vigilant about such attacks, Figure 1 indicates that they are sill an ongoing problem for organizations such as the United States Government [4].

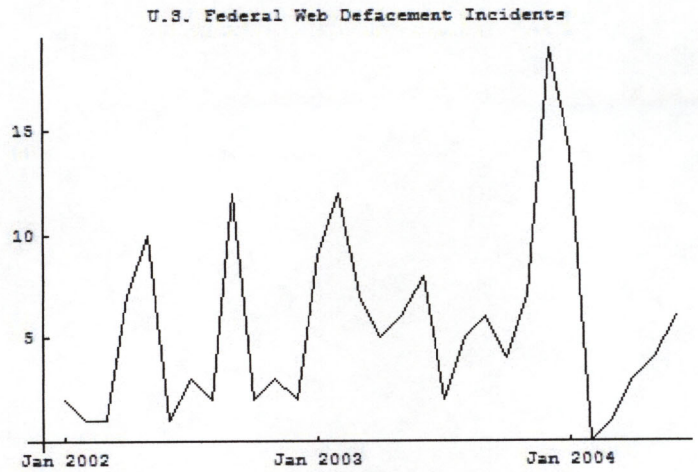

Figure 1: US CERT Federal Website Defacement Statistics

While no scheme can protect users against severe compromises of the content originators, we address a broad class of web content tampering attacks, even when applied to high-volume, load-balanced web serving deployments. In this paper, we propose and demonstrate modifications to existing web protocols that will allow proofs of authentication for each and every file transferred from a HTTP server. These proofs require low overhead and can be incorporated transparently into the existing web infrastructure.

While the proposed extensions to web protocols will not protect users from all classes of compromises, it does protect users from malicious content that is substituted in mirror repositories. This is especially important as web-caching makes mirroring implicit and the potential for man in the middle attacks high. These extensions can be used, for example, to vastly improve security of web content serving systems such as Coral [18] and YouServ [19] that use untrusted end-user machines for democratizing web publishing.

Just as the commonly used "https:" protocol specifier indicates a HTTP connection over a TLS secured internet channel, we propose a new "httpa:" protocol specifier that would indicate both to the browser, the user, and the server that a proof of authenticity is desired for this connection. Strictly speaking, a new protocol specifier is not necessary to implement our proposed authentication scheme, however, for the purposes of this paper we denote the combination of normal HTTP with proofs of authenticity as the HTTPA protocol.

2. BACKGROUND

Verifying that documents downloaded from a website are those created by the originator is an essential part of applications such as web banking. The necessary tools for establishing a secure channel between a certified content provider and a client computer across the internet, particularly HTTPS, are part of all modern browsers. The use of secure transport and certificate based authentication is an essential part of modern web applications [5]. However, HTTPS is complex, requires trusted third-parties to serve as certification providers, and, arguably, suffers because it is poorly understood by users.

In applications where high volume results in the need for mirroring, the use of HTTPS greatly increases the computational resources needed for each client access. Beyond the cost of establishing a symmetric key, the HTTPS socket communication requires computation of the stream cipher and mathematically combining it with the files being served. While some servers have dedicated hardware for this purpose, most servers perform these math operations using the general purpose CPU.

For very popular content that is downloaded by thousands of users simultaneously, distribution of server load across multiple machines is essential. Such mirroring can be done informally through volunteers creating independent copies of large archives, or more formally through the use of server farms which are often maintained by separate companies.

Unfortunately, the use of HTTPS and distributed mirrors of content are mutually incompatible. In order for a server to provide HTTPS secured content, it must have a (certified) private authentication key associated with the url from which the content is being served. Distributing the private authentication key amongst many computers, especially when these machines are often not under the direct control of the content originator presents a difficult key management issue that threatens to undermine the security of the HTTPS model.

The use of cryptographically secure hashes, usually based upon PGP or GPG, MD5, or SHA, which are distributed by content originators has become a custom amongst organizations such as the Linux kernel developers. In fact, the CERT organization recommends this practice [2]. There have even been attempts to formalize and automate this process [6].

The use of file hashes to solve the problem of authenticating mirrored content presents its own problems. The file hashes are often distributed with the content [7], defeating the whole point as one need only replace the file hash with the hash of the altered content. More typically, as file hashes are small they can be distributed by the original publisher, possibly, although not typically, using HTTPS. For example, Red Hat currently signs all of their distribution packages using Gnu Privacy Guard (GPG) and publishes these signatures via HTTPS.

A typical web server can host thousands of files, and individually authenticating each of them would require, likewise, thousands of hashes. The need to request hashes from the original publisher competes with the distribution of the content itself. If authentication is to be done automatically, some mechanism must be found to distribute this load.

With minor modifications, web servers can provide additional metadata along with the requested content that allows browsers to compute authentication proofs that demonstrate the the content delivered is identical to the content provided by the originating publisher, to the high degree afforded by the cryptographic hash scheme. This scheme allows both formal and informal replication of the content (without

modifications) to be hosted across the network without fear of tampering. Because all of this data is precomputed by the server, and is sent as clear-text over the TCP/IP socket, there is no additional computational burden beyond the small increase in data transmitted.

3. RELATED WORK

The pioneering work by Ralph Merkle [8, 9] provides the basis for our method of authenticating web content. His method of hierarchically organizing a collection of digital signatures, or content hash values, provides a mechanism whereby a single value can be use to verify small portions of a large document. More recently, such *Merkle trees* have been proposed as an authentication mechanism for a number of applications including public-key cryptography certificate management [10], XML documents [11], Web services registries [12], and, quite generally, to databases [13]. The use of Merkle trees to authenticate a large set of data structures is presented in [14].

There has been other work on improving the robustness of web content caching infrastructure. One proposal [15] is intended to support caching of dynamic as well as static content, while allowing publishers to receive access logs from cache servers. The scheme relies on periodic probing of caching servers by clients and publishers in order to detect tampering. This approach is intended only to *eventually* detect servers that are involved in tampering attacks. Our approach in contrast allows for tampering to be detected instantly and unconditionally, and with very small overhead on clients and almost negligible overhead on content publishers and its mirrors.

4. WEB CONTENT SERVING WITH MERKLE TREES

One of the fundamental primitives in cryptography is the one-way hash function. A one-way hash function takes as input a document, or *pre-image*, and produces a smaller, fixed length, *hash value*. An effective cryptographic hash function insures that, given a particular hash value, it is computationally infeasible to find a pre-image that, when hashed, results in that value. Cryptographically strong hash functions are standard parts of most internet platforms, for example the Java language provides an implementation of SHA-1 which we will be using for our discussions.

As discussed in the introduction, for mirrored content, if a separate file containing a cryptographic hash of each file can be obtain through a *trusted channel from the originating publisher*, every file could be verified as being authentic (as provided by the security parameters of the hash function.) But maintaining and distributing hashes for each individual file poses a logistical challenge. While it is simple for any host to compute the hash of a particular file, it is far more difficult to establish a trusted channel to the publisher for each and every authentication request.

In this paper we propose and evaluate the use of Merkle hash trees for authenticating responses to web requests. A Merkle hash tree is a technique for combining cryptographic hashes recursively in a binary tree structure. Merkle hash trees have proven useful for numerous authentication problems since only the root of the tree structure needs to be delivered securely in order to support authentication of a wide class of responses delivered over untrusted channels.

One application of Merkle trees is in directory response authentication. Figure 2 illustrates a Merkle hash tree computed for a simple directory of four names. H(string) is an appropriate cryptographic hash function such as SHA-1. H(L,R) is a secure binary input hash function, for example, SHA-1 on the concatenation of its two input arguments. The elements are arranged into a binary tree by recursively pairing nodes until only a single node remains. The value of each internal node is itself a hash of the hashes of its immediate children. By publishing (or *committing*) just the hash value of the root of the tree, it is possible to prove that any of the four names at the leaves of the tree were part of the tree when the root hash value was published.

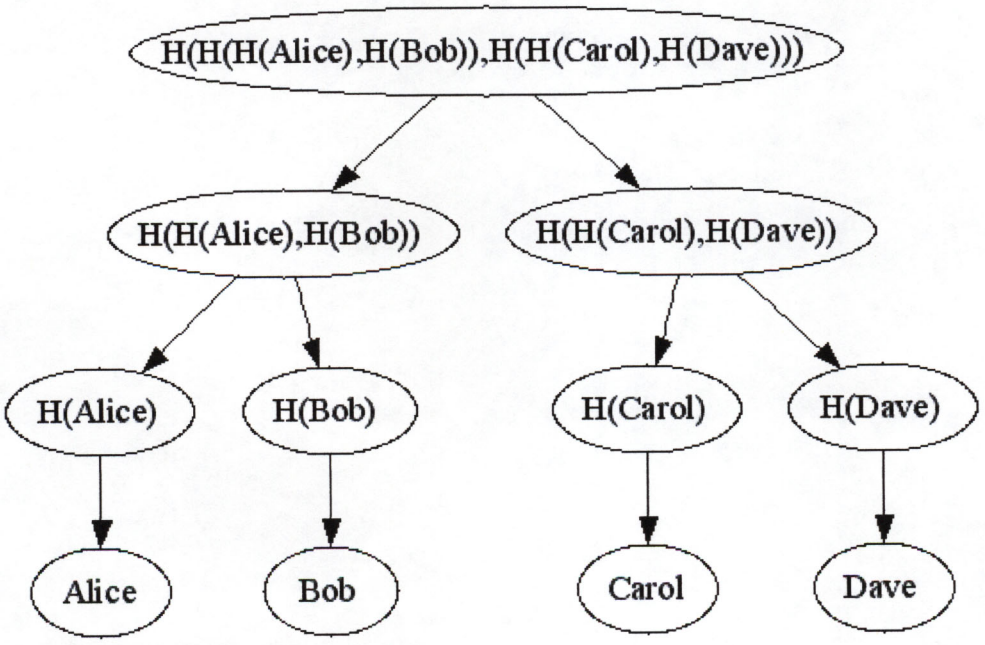

Figure 2. Example of a Merkle Hash Tree

To illustrate, in response to the question "Is Carol in the directory?" the directory host (which need not be a trusted machine) provides the hashes H(H(Alice),H(Bob)) and H(Dave). From these two pieces of information, the client has all information necessary to compute all

hash values along the path from Carol to the tree root. The response is proven authentic if the so-computed root hash matches the root hash obtained securely from the publisher or some other trusted source. If the directory host attempts to tamper with the response, the root hash value will fail to match. In this case, the client knows the host's response is invalid and can respond appropriately (for example, by directing the query to another host.)

The tree in Figure 2 was purposely constructed by ordering the names in the directory lexicographically from left to right. So ordered, the tree also supports negative response authentication [13]. For example, suppose the client asks the directory host "Is Betty in the directory?" In this case, since Betty is not in the directory, the directory host will deliver information that shows that Alice and Bob are both in the directory, and nothing falls between them (thereby proving that Betty could not be in the directory.) To do so, the host need only deliver the names Alice, Bob, and the value H(H(Carol),H(Dave)) for the client to compute the root hash and perform the necessary verification.

Merkle-tree approaches boast the following desirable properties: (1) only a very small amount of information needs to be delivered securely. For example, if SHA-1 is the hash function of choice, this amounts to a mere 20 bytes. All other response information can be delivered via untrusted channels, which are more plentiful and less costly. (2) The additional information that must be delivered by the untrusted hosts to allow client authentication of the responses is also very small. If the merkle tree is perfectly balanced, it amounts to only $\log_2(N)$ hash values where N is the number of directory elements for authenticating positive responses, and up to twice that for negative responses.

4.1 Webserver Response Authentication via Merkle Trees

The problem remains of applying Merkle hash trees to support webserver response authentication in a manner that is efficient, secure, maintainable, and can be easily deployed within the existing web infrastructure. For a given site, our solution first involves arranging hashes of both resource contents and the *canonical web paths* of those resources into a special Merkle tree structure. A canonical webpath is a normalized form of the resource URL whereby the scheme and hostname portion is removed, escape sequences decoded, and so on. For example, `http://mysite.com/a+file.htm` and `http://mysite.com/a%20file.htm` would both canonicalize to the webpath "a file.htm."

The tree structure must support efficient yet strong authentication of both 200 (found) and 404 (not found) responses by the client. Its arrangement is as follows: Leaf nodes represent resources, and store the hash of the resource contents, the hash of the resource's canonical webpath, and the result of hashing the concatenation of these two values. Leaf nodes are ordered according to the canonical webpath hash value (order can be imposed by treating this value as an unsigned integer). This ordering provides a mechanism for generating proofs that particular path names are not present in the directory without disclosing the path names of files that are present. Internal nodes consist of pointers to left and right children that enforce the ordering. Internal nodes themselves maintain hashes computed over the result of concatenating the hash values of their left and right children.

Table 1: Hypothetical web directory to be mirrored.

Filename	Contents (HTML 2.0)	Rendered	SHA-1 Content	SHA-1 Path

alpha.html	`<html>`
	`<head>`
	`<title>Alpha Page</title>`
	`</head>`
	`<body>`
	`<p>`
	`I am alpha.`
	`</p>`
	`</body>`
	`</html>`
		Alpha Page
		I am alpha.
			`3fed e020 bbfd d125 c8eb 8693 ad4e cd00 2cde 7001`	`f8c9 dbeb 83db df33 be07 ab75 fd36 357b 1535 c54b`

gamma/delta.html	`<html>`
	`<head>`
	`<title>Delta Page</title>`
	`</head>`
	`<body>`
	`<p>`
	`I am delta.`
	`</p>`
	`</body>`
	`</html>`
		Delta Page
		I am delta.
			`ee1a be6d 01df f8b2 44c1 3558 b2b2 d8b9 b92a 5e69`	`7aa1 429c 20f1 ace4 ff95 a239 8f87 32ef c8c0 b783`

index.html	`<html>`
	`<head>`
	`<title>Home Page</title>`
	`</head>`
	`<body>`
	`<p>`
	`click <a href="alpha.html">alpha</a> or`
	`<a href="gamma/delta.html">delta</a>.`
	`</p>`
	`</body>`
	`</html>`
		Home Page
		Click alpha or delta.
			`2484 fe4c 21cf c96f 3da9 0fcc 4147 61ae f2e1 b61f`	`3019 4810 013f 11e6 a990 8f76 037f 65bb fc1d 9efb`

Table 1 contains a directory of three files that are to be published by a hypothetical publisher, along with the twenty byte SHA-1 hashes of their contents and canonical webpaths. A tree built for this hypothetical web site is depicted in Figure 3. Note that a "leaf" as we have defined it above is depicted using three nodes, one for the webpath hash, one for the resource content hash, and one for the result of combining them. We call a tree for a given repository that meets the above requirements a *web hash tree*.

As with binary search trees [16], there are many possible trees for a given repository that meet the web hash tree requirements. One goal is

to generate one with a short average path length, since as we later show, this value can be used to bound request overhead.

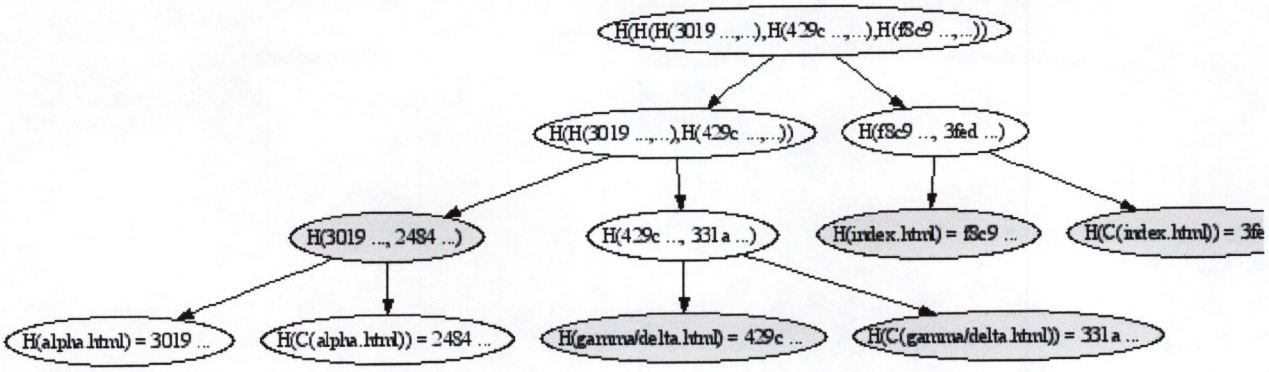

Figure 3. Example of a web hash tree

Before diving into detailed algorithm descriptions, we illustrate through examples how web hash trees enable efficient client-side authentication of responses to requests. Consider again Figure 3. Suppose the client issued a request for the resource "/beta.html", which does not exist in the repository. To prove that the resource "beta.html" is not present on the server, the server computes the value of H(C (beta.html)) (where C denotes the canonicalization function). Assume this value is `f2e2 3350 0f74 3442 9f25 2303 d484 081c 572d 7fd2`. It then locates the two leaves that are nearest (one greater than and one less than) this value in the ordering (delta.html and index.html). From these leaf nodes, the server traverses the tree up to the root, returning those hash values that are highlighted in the figure. These nodes correspond to the children of nodes along those two paths that are not themselves on the paths. Note that this information is sufficient to allow the client to compute the root hash. To verify authenticity, the client compares the computed root hash value with a securely obtained root hash for the repository, and then performs a few additional checks which we describe later. In general, the number of hash values the server must deliver to allow the client to authenticate such a HTTP 404 "file not found" response is bounded by the sum of the lengths of the two paths from the two leaves to the root.

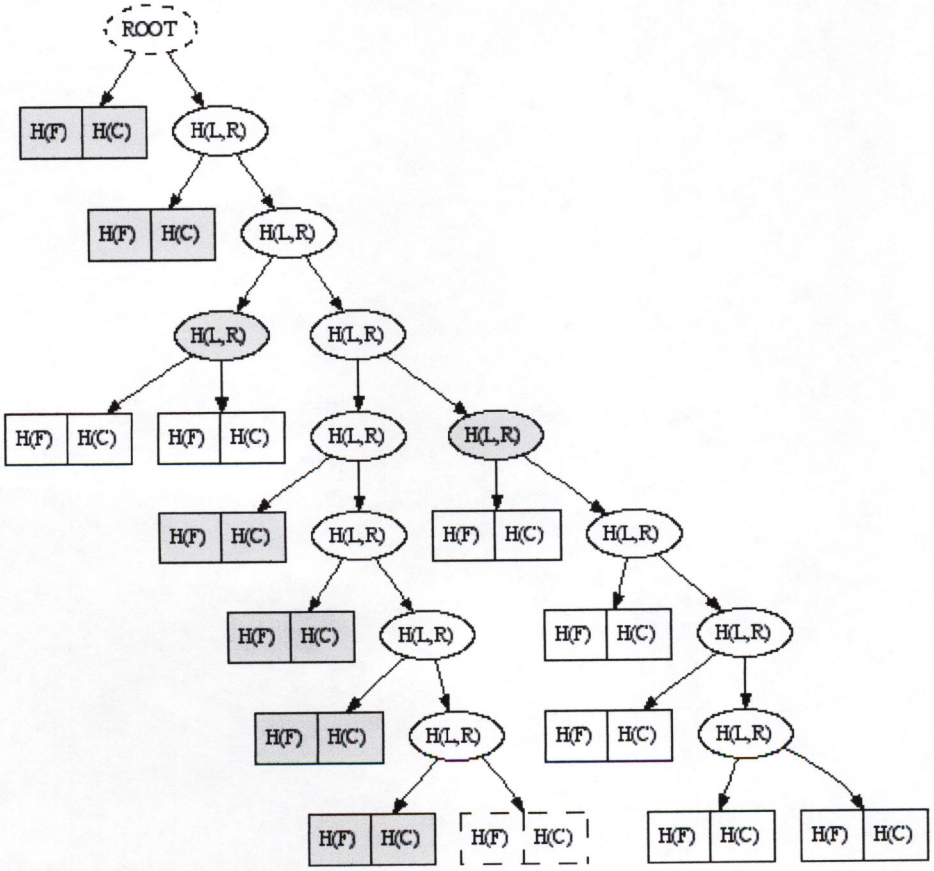

Figure 4. Hash path to verify the contents of a leaf

Figure 4 illustrates a larger collection of files in an randomly constructed web hash tree. In order to verify the indicated leaf, the server

must provide hash values associated with each of the shaded nodes. The client can readily compute both H(F), the hash of the contents of the requested web resource, and H(C) the hash of the canonical webpath. By computing the intermediate hash values up the tree, the client can verify that the root hash is consistent with the committed root hash. In general, the number of hash values the server must deliver to allow the client to authenticate such a HTTP 200 "found" response is bounded by the length of the path from the requested leaf to the root. *Note that the client never needs to obtain the entire web hash tree to perform authentication.* The insecurely delivered path information and file contents, along with securely delivered root hash is all that is required.

The expected path length from leaf to root dictates how much "extra" information beyond standard HTTP that must be provided by the server to the client. Sites can generate balanced web hash trees to guarantee a logarithmic bound on the number of hashes per request that must be delivered. However, since the web hash tree is ordered by the hash of the canonical webpath, even a simple tree building algorithm based on iterative insertion is sufficient for a probabilistic logarithmic bound. Cryptographic hash algorithms produce randomly distributed outputs even if their inputs are dependent. The expected length of a path in an insertion-constructed web hash tree should be roughly predicted by that of random binary search trees, which is $1.4\text{Log}_2(N+1)$ where N is the number of elements [21]. The primary difference is that the web hash tree stores all elements in the leaves whereas binary search trees allow elements at internal nodes. The iterative insertion-constructed web hash tree may therefore have a slightly larger expected path length, however our experiments indicate that the actual height closely match the value predicted by the random binary search analysis (Section 7).

4.2 Detailed Algorithm Description

Detailed pseudo-code for both proof generation (as invoked by the server) and proof verification (as invoked by the client) appears in Table 2. The hash computation and combination functions are assumed to be cryptographically secure.

The pseudo-code assumes a WebHashTree object that consists of two node types, LeafNode and InternalNode. LeafNode objects represent web resources and always reside at the leaves of the tree. They consist of three hash values, one for that of the resource webpath, one for that of the resource contents, and one hash that is the result of combining these two hashes (i.e. concatenation) and hashing the result. The "getHash" method of a leaf node returns this combined value. Recall that we assume the web hash tree is constructed in a manner such that the leaves are ordered from left to right in ascending value of the (unsigned 20 byte) webpath hash value. Each internal node points to a left and a right node. It also contains one hash value, returned by getHash, that is the result of hashing the concatenation of hashes returned by invoking getHash on the nodes left and right children.

The case for 200 responses is relatively straightforward. The server first canonicalizes the webpath by URLdecoding it (e.g. converting escape sequences into their respective characters) and performing other normalizations. The goal is to ensure that different URLs that correspond to the same resource are always transformed into an identical representation. These normalizations must be applied consistently by both clients and servers.

Following webpath canonicalization, the proof generator computes the hash of the result. This hash is used to quickly search the web hash tree for the resource corresponding to the hash. If such a resource exists, the path from child to root is traversed, and for each node, the hash of the child that does NOT reside on the path is placed in a list. (While omitted in the pseudocode, the server must also associate information with each hash that specifies whether the hash value arose from the left or right child of each node to allow the client to concatenate the hashes in the appropriate order.) This list comprises all the information necessary for the client to verify authenticity of the response when coupled with the webpath and the resource contents. To verify the proof, the client performs its own canonicalization and hash of the webpath, and combines this value with the hash of the resource contents. It then iteratively combines this hash with each hash in the list returned by the server. If the result equals the securely obtained root hash value, the download is authentic.

The case for 404 responses is a bit more involved. The server again canonicalizes the webpath and computes the hash of the result. Because there is no resource corresponding to that exact value, the algorithm instead fetches the leaf nodes corresponding to the nearest "less than" node and the nearest "greater than" node in webpath hash order.

The proof in the case of a 404 response consists of two lists of hashes. One list of hashes represents those nodes along the path from the "greater than" node to the least common ancestor shared with the "less than" node. This list is embedded in the list of hashes from the path connecting the "less than" node to the root. As is the case with 200 responses, the hash values in the lists are the hashes of those children of path members that do not reside on the path.

Client verification of the 404 response proof involves a bit more than simply ensuring the hashes appropriately combine into the securely obtained root hash value. In addition, the client must verify that the webpath hash of the requested resource falls in-between the webpath hash of the two returned leaf nodes. The client must also ensure that the "left child" and "right child" indicators associated with the hash values are consistent with the fact that the greater-than and less-than nodes are adjacent with respect to the leaf ordering induced by the web hash tree (ValidateLeafAdjacency). Note that the pseudo-code currently ignores boundary conditions where either the less-than or greater-than node is null.

Security of the scheme can be established by a relatively straightforward adaption of the arguments used in [13]. These details will appear in an extended technical report.

Table 2: HTTPA Algorithms.

```
;; Compute the proof necessary for the client to authenticate
;; the request specified by the given webpath. The proof is
;; returned as an ordered collection of hashes.

List GetProof(String webpath, WebHashTree t)
  List return_hashes ← new List
  webpath ← Canonicalize(webpath)
  Hash webpath_hash ← Hash-Compute(webpath)
  LeafNode n ← t.searchFor(webpath_hash)
  if n=null
    then return Get404Proof(webpath_hash, t)
```

```
  Node previous ← n
  InternalNode p ← n.getParent

  while p != null
    do if p.getRight == previous
          then return_hashes.add(p.getLeft.getHash)
          else return_hashes.add(p.getRight.getHash)
       previous ← p
       p ← p.getParent
  return return_hashes
```

```
;; Verify the proof provided by the server given the
;; webpath, the hash of the download, and the root hash,
;; assuming the request resulted in a 200 response.

boolean Is200Authentic(String webpath, Hash download_hash, Hash root_hash, List proof)
  webpath ← Canonicalize(webpath)
  Hash webpath_hash ← Hash-Compute(webpath)
  Hash result_hash ← Hash-Combine(webpath_hash, download_hash)
  for each hash_value in proof do
      result_hash ← Hash-Combine(result_hash, hash_value)
  if (result_hash == root_hash)
     then return true
     else return false
```

```
;; Compute the proof necessary for the client to authenticate
;; the "not found" request specified by the given webpath.

List Get404Proof(Hash webpath_hash, Tree t)
  ;; 404 proof consists of two ordered collections of hashes,
  ;; one of which will be nested in the top level collection
  LeafNode l ← t.searchForNearestLessThan(webpath_hash)
  LeafNode r ← t.searchForNearestGreaterThan(webpath_hash)
  ;; Note -- for simplicity we ignore the boundary cases where l or r are null

  List right_hashes ← new List
  right_hashes.add(r.getWebpathHash)
  right_hashes.add(r.getFileHash)
  Node previous ← r
  InternalNode p ← r.getParent

  while p != null and p.getRight != previous
    do right_hashes.add(p.getRight.getHash)

  Node right_branch_root ← p

  List return_hashes ← new List

  return_hashes.add(l.getWebpathHash)
  return_hashes.add(r.getFileHash)

  previous ← l
  p ← l.getParent

  while p != null
    do if p == right_branch_root
          then return_hashes.add(right_hashes)
          else if p.getRight == previous
                  then return_hashes.add(p.getLeft.getHash)
                  else return_hashes.add(p.getRight.getHash)
       previous ← p
       p ← p.getParent
  return return_hashes
```

```
;; Verify the proof provided by the server given the
;; webpath, the hash of the download, and the root hash,
;; assuming the request resulted in a 404 response.

boolean Is404Authentic(String webpath, Hash root_hash, List proof)
  if ValidateLeafAdjacency(proof) == false
     then return false
  webpath ← Canonicalize(webpath)
  Hash webpath_hash ← Hash-Compute(webpath)
  Hash left_webpath_hash ← proof.removeFirstElement
  Hash left_file_hash ← proof.removeFirstElement
```

```
  Hash result_hash ← Hash-Combine(left_webpath_hash, left_file_hash)
  Hash right_webpath_hash ← null ;; will initialize in loop
  for each element e remaining in proof do
      Hash hash_value ← null
      if e isa List
          then (hash_value, right_webpath_hash) ← Combine-Hash-Path(e)
          else hash_value ← e
      result_hash ← Hash-Combine(result_hash, hash_value)
  if (result_hash == root_hash)
      if webpath_hash > left_webpath_hash && webpath_hash < right_webpath_hash
         then return true
  return false


(Hash, Hash) Combine-Hash-Path(List right_hash_list)
  Hash right_webpath_hash ← right_hash_list.removeFirstElement
  Hash right_file_hash ← righ_hash_list.removeFirstElement
  Hash result_hash ← Hash-Combine(right_webpath_hash, right_file_hash)
  for each hash_value in right_hash_list do
    result_hash ← Hash-Combine(result_hash, hash_value)
  return (result_hash, right_webpath_hash)
```

5. ROOT HASH DISTRIBUTION

Considering the data published by a particular web server as an aggregated whole, a single hash value can authenticate each individual file using the algorithm presented above. A remaining problem is that of securely delivering the root hash value, which authenticates all of the content. It seems to be a widespread and strongly misguided practice to distribute file signatures along with files. Doing so provides a simple attack mode, where the a phony signature is substituted to (in)correctly match a phony distribution file. Failure to obtain an up-to-date and faithful root hash value threatens to invalidate the entire authentication scheme. Fortunately, several existing web standards provide the means to deliver this data, and we outline three such techniques.

In order to insure the faithful delivery of the root hash, a public-key based solution is clearly desirable. It might be reasonable to obtain the root hash directly from the content publisher without the use of cryptographic guarantees, but such a practice would open the system to new attack modes, including man in the middle attacks where a substituted root hash is propagated, authenticating malicious content.

Despite their complexity, public-key based digital signatures provide strong guarantees of authentic data delivery. Using certificated public-keys, clients can fully automate the authentication process. Already established web protocols provide three elegant methods for distribution of this small item. These techniques are not mutually incompatible, and may all be simultaneously used in accordance with a particular client's policies.

5.1 DNS-SEC

The domain name service is a distributed, dynamic directory service that is intended to provide internet based hosts fast access to essential information for locating internet hosts. The primary function for DNS is to map hostnames to internet addresses. In fact, for many mirroring applications, the IP address associated with a host dynamically changes with each query, using a round-robin rotation across multiple duplicate hosts.

The DNS system, with its distributed automatic caching, dynamic update capability, and its design for small data records all suggest this would be an excellent mechanism for publishers to use to distribute the root hash value. This can be done either using the existing TXT DNS records, or through a format extension to a new type of DNS record. Unfortunately, DNS itself lacks authentication mechanisms. While this has not made the internet unusable, it certainly is one most cited weakness.

Fortunately, DNS-SEC [17], the next generation of DNS, incorporates public-key based digital signatures to authenticate the data records returned by each query. No modification of existing protocols is needed to support distribution of the root hash as a specially formated TXT DNS record, consider the following excerpt of a **named** db.auth file:

```
www     A       192.168.1.30
www     A       192.168.1.31
www     A       172.16.1.12
www     TXT     HTTPAROOTHASH=794C7AEB0D561DA5514F247E8766B4B723564BDB
```

Here three separate web servers, each presumably configured to serve the same content, share a single HTTPA root hash value. DNS-SEC is specifically designed to support dynamic update of DNS records [20]. At this time, DNS-SEC is still in a developmental stage with only a few deployments. While this is our preferred mechanism for distribution of root signatures, we also present two others which may be more feasible in the near term as DNS-SEC matures.

5.2 HTTPS to Content Provider Server

Transport layer security, a certified public-key based of authenticating and securing TCP/IP communications, can also be used as a distribution mechanism for the mirrored site's root hash. This can result in significant traffic to the content provider's servers, and a significant computational workload for the SSL handshake. However, in this application, only authentication is required so a server may be configured to use an unencrypted socket, or "null stream cipher." Support for this type of connection is part of most TLS implementation suites.

It is important to note that the HTTPS based method of obtaining a root hash must use a well defined method of transforming the URI to

the name of the authentication server. The third-party certificate obtained by the publisher will only authenticate that the hostname matches that specified in the certificate. For example, if a URI of http://www.popularsite.org/ is mapped to multiple servers by multiple DNS A records, then TLS connections to https://www.popularsite.org/ would also connect to all of these servers as well. Therefore, by augmenting the URI, a hostname such as httpa.www.popularsite.org can be created with a DNS A records that point only to the publisher's servers, and which have the appropriate TLS credentials to allow HTTPS connections.

5.3 Certified PKI Signed Root Hash

Finally, we discuss a method for distributing the root hash from the mirror sites in the case where no direct method of contacting the publisher is available. This method requires infrastructure that is currently not part of every modern browser, but is part of the internet email infrastructure. While this technique can authenticate the content, it cannot assure the freshness of the content. This can be ameliorated through the use of digitally signed time-to-live (TTL) values.

Distributing the root hash as a file is subject to the attack of a malicious mirror altering content arbitrarily, computing the new root hash, and serving the malicious root hash instead of the true root hash. This attack can be prevented if the root hash is digitally signed using the publishers private, the public key of the publisher is made available, and the public key of the publisher is certified by one of the certificate authorities recognized by the browser client.

6. IMPLEMENTATION OVERVIEW

In this section, we sketch the roles and responsibilities needed to implement the proposed HTTPA protocol. We have successfully implemented each of these systems and will provide experimental analysis in Section 7.

6.1 Content Originators

The primary responsibility of content originators in enabling HTTPA is computing and maintaining the web hash tree of the documents hosted on the site and distributing its root hash value through one or more of the previously outlined methods of distribution. Maintenance of the tree must be performed every time site content changes. Originators can help mirror sites stay up to date by publishing file difference lists keyed upon the old root hash and the new root hash rather than republishing the entire repository. Note that for each site update, which will typically involve only a small percentage of files, a mirror does not necessarily need to download the entire web hash tree from the originator. Instead, the originating site can publish the exact modifications that need to be performed to transform the web hash tree into one that is up to date. Alternatively, the mirror can apply the same update procedure as the originator in order to compute the updates itself from the content hashes of the new, changed, and deleted resources. (Updated resources can be treated as a deletion of the previous version of the resource followed by an addition of a new resource.) If we assume a "standard" tree update procedure that does not attempt to enforce balancing, each new or deleted resource affects at most a single path from leaf to root within the web hash tree, so the amount of web hash tree change information is typically quite small.

As we have previously mentioned, the ordering of leaves by hash of the canonical webpath ensures random tree update patterns even when updates are highly correlated by webpath. Our implementation therefore uses the standard binary search tree insert and deletion algorithms to maintain the web hash tree. In the binary search tree literature, it is known that random inserts via the standard insertion algorithm leads to a logarithmic expected path length [21]. Unfortunately, it is also been empirically demonstrated that in the presence of random deletions as well as insertions, the expected path length tends to sqrt(N) [22]. However, in the case of web hash trees, deletions are restricted to leaf nodes since internal nodes are used only for structure and not for storing resource hashes. Culberson [22] argues that the tendency towards sqrt(N) average path length is due to deletions of internal nodes and not the leaves. Indeed, though our generalization of these results in binary search trees to web hash trees is at this point informal, empirically we have found an expected logarithmic path length to hold in practice under repeated insertions and deletions.

6.2 Mirror Hosts

Mirror sites must download site content from the originators or other up-to-date mirrors, and maintain their own copy of the web hash tree either by downloading or computing the changes resulting from each site update. In response to an HTTPA request from a client, the mirror host must determine the sequence of hashes necessary to allow the client to compute the hashes up to the root. These hashes can be Base64 encoded, concatenated, and returned through an HTTP response header. For each resource, the mirror has the option of precomputing the hashes that must be delivered with each resource, or it can compute them on the fly with only slight overhead. The ordered leaf arrangmenet allows the web hash tree can be searched in time proportional to the path length where the resource leaf resides. For 404 responses, some precomputation is also possible, but since 404 requests should be relatively rare, dynamically computing the result hashes from the web hash tree would typicallybe sufficient.

6.3 Web Browser Clients

Browser clients need to carefully consider how security operations and their outcomes are displayed to users. Presuming that that users can be taught to effectively interpret these outcomes, a browser needs to be equipped with an httpa protocol handler implementing the protocol, including the necessary schemes for obtaining root hash values. Browsers should apply configurable policies in dealing with authentication failures. In cases where there are potentially many mirror sites available, it is plausible for the browser might be configured to automatically fail over to a new site. At a minimum an unambiguous warning message should be provided to the user when a faulty mirror is detected. Since authentication failures can be the result of out of date root hash values or mirror content rather than malicious tampering, the policy should also dictate how root hash version conflicts should be resolved.

7. EXPERIMENTS AND EXPERIENCES

We have implemented the HTTPA protocol by extending an existing HTTP client and server for experimentation and prototype development. In this section we first evaluate the overhead of hash tree enhanced web protocols in comparison to standard web protocols such as HTTP and HTTPS. We also highlight important implementation details that make the scheme attractive in practice such as, for example, in achieving backwards compatibility. We conclude the section with a description of a proof of concept deployment in the form of a Mozilla/Firefox extension.

7.1 Implementation Details

Our modified HTTP client initiates an authenticatable download by submitting a standard HTTP request for the desired file to the server, but with one additional HTTP request header specifying "HTTPA: accept". Old servers that do not support tree based authentication will ignore the header and respond with the content as if it were an HTTP request, providing backwards compatability (though the client will warn the user that no authentication is performed in such situations). Servers that do handle the protocol produce an additional HTTP response header containing the hashes from the web hash tree that are needed to authenticate the content via the root hash. The client performs a SHA-1 hash of the content while it is being downloaded to allow it to be authenticated upon download completion by the previously described procedures.

To handle situations where the client and server's root hash are out of sync, the server also returns the version number of its root hash for the served content via the HTTPA response header. If the client has not yet securely obtained the root hash, it can obtain it from any of the previously defined approaches. If the root hash version number of the server is out of date, the client can either warn the user, attempt to download the content from another mirror, or perhaps continue as normal if the client can securely obtain the older root hash, and this version's creation date is within a specified tolerance of the most recent update.

Note that as implemented, the additional burden on the server beyond that of HTTP is in interrogating the request for the HTTPA request header, and then looking up and serving the extra bytes required for the additional HTTP response header information. Most of the additional overhead (computing the SHA-1 of the downloaded file) is incurred at the client, which will typically have plenty of spare compute power.

7.2 Experimental Setup

We conducted webserving experiments using the actual web content from mirrors of 47 personal websites. The entire collection consists of over 17 Gbytes of content stored in 42,445 files.

All experiments were run on a dedicated 2-way 2.4GHZ Intel Pentium-IV Xeon class machine with 3GBytes main memory. The client and server were both run on the same machine to minimize networking artifacts. Implementation of both server and client code was in Java. All cryptographic functions such as SHA-1 and RC4/RSA as required by HTTPS and the HTTPA schemes were provided by the IBM Cryptolite and SSLite libraries, which perform within a factor of 2 from optimized C implementations such as OpenSSL.

7.3 Computing the Web Hash Tree

We first computed one web hash tree over the entire collection of 47 sites to simulate the effects of using the scheme for a single large site. The tree was built through the standard binary search tree insertion algorithm, with files inserted in lexicographic order of canonical webpath. While the files could have been added in any order to construct the web hash tree, we chose a lexicographic order to demonstrate that a highly non-uniform insert pattern in terms of webpath still provides balanced trees by virtue of sorting the leaves on canonical webpath hash. The average path length of the resulting authentication tree was 22, which matches the $1.4\text{Log}_2(N+1)$ average path length of truly random binary search trees. Average path length within the web hash trees constructed over each of the 47 sites individually also closely matched this theoretically determined value. Experiments showing that repeated deletions and insertions also preserve a logarithmic expected path length appear in our extended report.

7.4 Performance

For the performance experiments, we first categorized the content into buckets based on file size. Each file in a bucket was requested and downloaded 10 times over a single "keep-alive" connection by each tested protocol, and the time required to complete all 10 downloads recorded. The protocols evaluated were HTTP, HTTPS (1024 bit RSA/SHA-1/RC4), and HTTPA (HTTP with SHA-1 Merkle-tree authentication). To eliminate the effects of disk I/O, each file was cached in memory before the requests for the file were submitted.

The results appear in figure 5 which displays the average time required with respect to bucket size. For the smallest files, the overhead of HTTP and HTTPA was typically less than the clock granularity. HTTPS on the other hand typically required a minimum of 100ms. We first suspected this was due to the relatively fixed overhead of HTTPS connection establishment, but after factoring out this cost the results remained unchanged. On larger file sizes, the cost of HTTPA was typically 2-3 times more than plain HTTP, primarily due to SHA-1 computation at the client. HTTPS was typically 4-4.5 times more costly than HTTP overall.

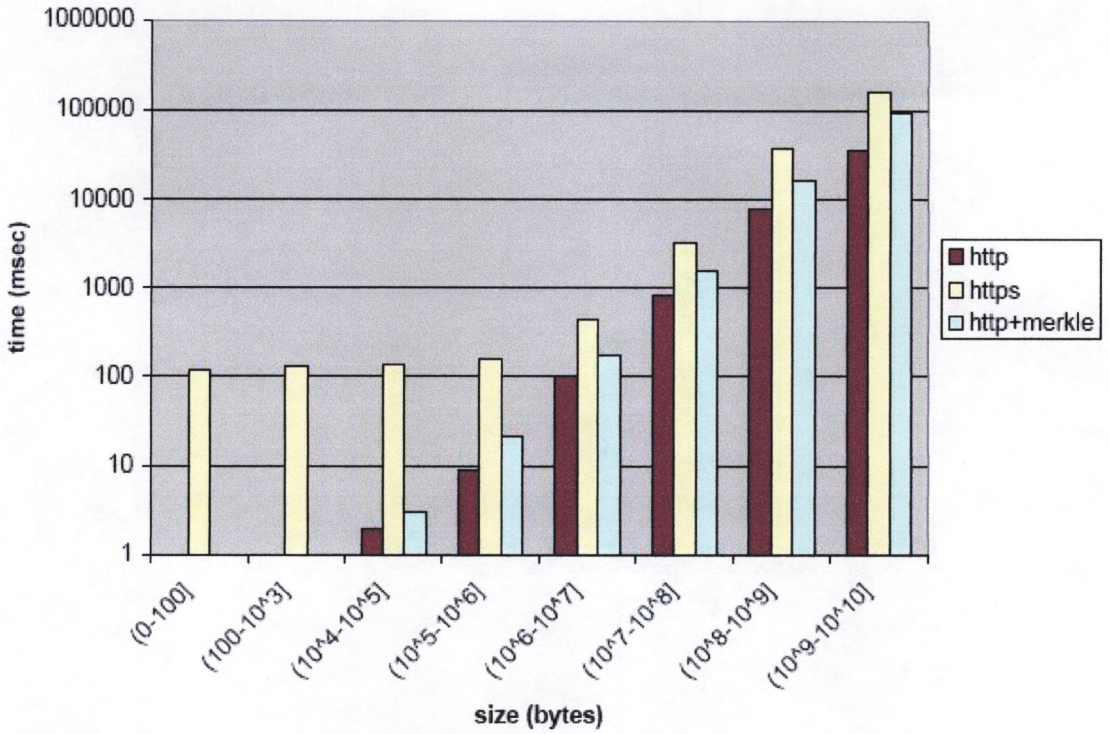

Figure 5. Performance comparison of HTTP, HTTPS, and HTTPA.

7.5 Proof of Concept Deployment

As a proof of concept, we have implemented a Mozilla/Firefox protocol extension that invokes the HTTPA protocol client in response to URLs with an httpa protocol scheme instead of http. This extension warns the user through a pop-up window should the authentication fail. From the end-user perspective the effect is transparent and there was no noticeable performance degradation when visiting sites providing HTTPA authentication.

One issue that became apparent in using the proof of concept was in how to best handle partially downloaded content. Current browsers display content as it arrives to improve browsing responsiveness. However, as we have defined the HTTPA protocol, authentication cannot be performed until the entire download is complete. Malicious servers could exploit this behavior to deliver "spam" messages. Even though the browser may ultimately deliver the "content not authenticated" message, the spammer can still get the message on the screen during the download, which may be motivation enough to behave unscrupulously. One straightforward solution would be to treat files larger than a certain size as multiple individually authenticatable chunks. Browsers implementing the protocol would then avoid displaying any information from a chunk that has yet to be fully authenticated. This would also eliminate the possibility of a malicious server delivering an endless stream of bogus data to effect a denial of service. Chunk size could start of small for each file and double in size with each chunk to reduce the number of extra hashes required per file. This forces any malicious server intending to deliver n bytes of bogus data (for all n larger than the initial chunk size) to first deliver at least $n/2$ bytes of valid authenticatable data to the client.

8. CONCLUSIONS

In web applications where authentication of the content delivered to a client over the internet is essential, HTTPS, or more correctly HTTP over TLS, is currently the only practical solution. For highly customized and dynamically generated content such as personal banking or secure web based email access, HTTPS is and will continue to be appropriate. The problem HTTPA solves is instead the reliable delivery of relatively static content that is hosted on multiple (possibly untrusted) internet hosts. Since every site is a combination of both static and dynamic pages or images, the protocols could be used in tandem to improve scalability of many applications requiring secure content delivery.

The fact is, despite the security pitfalls given current web protocols, the use of untrusted hosts for load distribution is already prevalent on the internet. In addition to many well-known software distribution mirrors, the p2p caching network Coral [18] is used daily by users of the popular tech blog Slashdot to minimize the impact of the so-called "Slashdot effect" on servers hosting pages referenced in feature stories. As the popularity of such services grow, they will become increasingly attractive targets for web spammers who could set up rougue servers to deliver their own messages or redirects in place of client-requested resources. HTTPA solves this problem by endowing clients with the ability to unconditionally and efficiently authenticate resources delivered over such channels, and with little additional burden on mirrors and publishers.

Table 2: Authenticated Web Publishing Comparison

Security	HTTPA	HTTPS
Trust properties	Integrity and authenticity	Integrity, Authenticity, and Confidentiality

Hosting servers	Untrusted	Requires Trust
Authentication Workload	File by file on client	Both client and server

We acknowledge that the extension or addition of new web protocols is a slow and difficult process. However, we believe our proposed HTTPA protocol addresses a specific and growing need that is not addressed by HTTP and HTTPS -- efficient and transparent delivery of *authenticatable* web content through primarily untrusted channels. In addition, for graceful roll-out, HTTPA can be implemented within the context of existing protocols in a manner that provides backwards compatibility with currently deployed servers and clients.

REFERENCES

[1] Robert Lemos, Attempted attack on Linux kernel foiled, CNET News.com, http://news.com.com/2100-7355_3-5103670.html

[2] GNU Project FTP Server Compromise, CERT Advisory CA-2003-21, http://www.cert.org/advisories/CA-2003-21.html

[3] Ken Thompson, Reflections on Trusting Trust, Communications of the ACM, V27 N8, August 1984, pp. 761-763

[4] Statistics on Federal Incident Reports, United States Computer Emergency Readiness Team, http://www.us-cert.gov/federal/statistics/

[5] Simson Garfinkel, Web Security & Commerce, 2nd Edition, Nov. 2001, O'Reilly & Associates, Cambridge.

[6] Jose Nazario, Signed Archives: An Evaluation of Internet Trust, CanSecWest, April 2003, http://monkey.org/~jose/presentations/csw03/

[7] Isaac Jones and Colin Walters, APT Signature Checking, http://monk.debian.net/apt-secure/

[8] Ralph C. Merkle, Method of Providing Digital Signatures, U.S. Patent 4,309,569, Jan 5, 1982.

[9] Ralph C. Merkle, Method of Providing Digital Signatures, U.S. Patent 4,881,264, July 30, 1987.

[10] J. L. Muoz, J. Forn, O. Esparza, and M. Soriano, Certificate Revocation System Implementation Based on the Merkle Hash Tree International Journal of Information Security (IJIS), V2, N2, pp. 110-124, 2004.

[11] D. J. Polivy and R. Tamassia, Authenticating Distributed Data using Web Services and XML Signatures, Proc. ACM Workshop on XML Security, ACM Press, 2002.

[12] Elisa Bertino, Barbara Carminati, and Elena Ferrari, Merkle Tree Authentication in UDDI Registries, International Journal of Web Services Research, 1(2): 37-57, 2004.

[13] Ahto Buldas, Meelis Roos, Jan Willemson, Undeniable replies for database queries, Fifth International Baltic Conference on DB and IS, June 2002, Tallinn, Estonia.

[14] Charles Martel, Glen Nuckolls, Permkumar Devanbu, Michael Gertz, April Kwong, and Stuart Stubblebine, A General Model for Authenticated Data Structures, Algorithmica, 2004 V39 p. 21-41, Springer-Verlag.

[15] A. Myers, J. Chuang, U. Hengartner, Y. Xie, W. Zhuang, and H. Zhang, A Secure, Publisher-Centric Web Caching Infrastructure, Proceedings of IEEE INFOCOM 2001 Anchorage AK, April 2001.

[16] Thomas Corman, Charles Leiserson, and Ronald Rivest, Introduction to Algorithms, MIT Press, 1993.

[17] D. Eastlake, C. Kaufman, Domain Name Security Extensions, Internet Engineering Task Forcr, RFC 2065, http://www.ietf.org/rfc/rfc2065.txt.

[18] Michael J. Freedman, Eric Freudenthal, and David Mazières, Democratizing Content Publication with Coral, In Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04) San Francisco, CA, March 2004.

[19] R. J. Bayardo, A. Somani, D. Gruhl, and R. Agrawal., YouServ: A Web Hosting and Content Sharing Tool for the Masses, In Proc. of the 11th Int'l World Wide Web Conference (WWW-2002), 2002.

[20] Paul Albitz & Cricket Liue, DNS and BIND, 4th Edition, Chapter 11, May 2001, O'Reilly & Associates, Cambridge.

[21] D. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching, Second Edition (Reading, Massachusetts: Addison-Wesley, 1998).

[22] J. Culberson and J. Ian Munro, Analysis of the standard deletion algorithms in exact fit domain binary search trees. Algorithmica, 6:295-311, 1990.