The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

SCALABLE WEB CONTENT ATTESTATION

A Thesis in

Computer Science and Engineering

by

Thomas Michael Moyer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Thomas Michael Moyer was reviewed and approved* by the following:

Patrick McDaniel
Assistant Professor of Computer Science and Engineering
Thesis Adviser

Trent Jaeger
Assistant Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

The web is a primary means of information sharing for most organizations and people. Currently, a recipient of web content knows nothing about the environment in which that information was generated other than the specific server from whence it came (and even that information can be unreliable). In this paper, we develop and evaluate the Spork system that uses the Trusted Platform Module (TPM) to tie the web server integrity state to the web content delivered to browsers, thus allowing a client to verify that the origin of the content was functioning properly when the received content was generated and/or delivered. We discuss the design and implementation of the Spork service and its browser-side Firefox validation extension. In particular, we explore the challenges and solutions of scaling the delivery of mixed static and dynamic content using exceptionally slow TPM hardware. We perform an in-depth empirical analysis of the Spork system within Apache web servers. This analysis shows Spork can deliver almost 8,000 static or 7000 dynamic integrity-measured web objects per-second. More broadly, we identify how TPM-based content web services can scale with manageable overheads and deliver integrity-measured content.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to take a moment to thank all of the people who made this thesis possible.

First, I want to express my deepest appreciation to my advisor Dr. Patrick McDaniel. With an enormous amount of patience, he helped to realize the full potential of this thesis. Throughout the process, he has provided me with many ideas and is always willing to go the extra mile to help.

I would also like to thank all the members of the SIIS lab. They have always been available to lend a helping hand be it just by listening to problems, or suggesting new solutions. My deepest gratitude goes to Kevin and Steve who spent a great deal of time poring over code and analyzing results.

I am grateful to my parents and extended family. They have been incredibly understanding of my prolonged absences as I worked on my thesis. With every pitfall, they were there to provide encouragement.

Most importantly, I want to express my deepest gratitude and appreciation to my wife and daughter. Without fail, they have been there to provide the support that carried me through the hardest times. They have been incredibly patient and understanding as I worked long hours to complete this work. To them, I dedicate this thesis.

Chapter 1

Introduction

1.1 Introduction

The web has changed the way that users and enterprises share information. Where once we shared documents via physical mail or through specialized applications, the web enables sharing content through open protocols. Web server validation, if done at all, is performed via SSL/TLS certificates [11]. The use of the certificate indicates that the server (really the certificate's private key) has been vouched for by some authority, e.g., Verisign.

What is missing is a mechanism that offers security guarantees on the content itself. Approaches like per-document XML signatures [17] provide document authentication, but only work where the data is static and the signing authority is separate from the web server, i.e., the user must either engage external signing authorities or trust the web server to create/handle the content correctly. Ideally, content receivers desire to know *a*) the origin of content and *b*) that the origin was functioning properly when the received content was generated and delivered. This latter requirement asks for proof of the server *integrity state* at the time of use.

Consider an online banking application. Users of the system provide credentials, account information, and other sensitive data to the web server as part of its use. For this reason, users need to know more than the identity of the server it is communicating

with (as provided by SSL). The users desire some assurance that the server has not been compromised. Similar requirements exist for any web application using sensitive data over untrusted networks, e.g., online auction systems, e-voting systems, online medical applications. Many of these applications must support thousands or millions of clients. Thus, an implicit requirement largely unaddressed by current integrity management approaches is that they scale to large communities.

Augmenting these applications with content integrity information will provide a means to detect and prevent real-world attacks. For example, if a server is compromised with malware, like the Mood-NT kernel rootkit [15], the proof of the system integrity will reveal the presence of the malicious software to the browser. Further, when bound to the content, the integrity proof exposes man-in-the-middle "in-flight" page changes [31], including advertisement injection, advertisement removal, and URL replacement, independent of whether the man-in-the-middle is present on the server, network, or web cache.

The Trusted Platform Module (TPM) [38] provides hardware support that enables remote parties (such as content-receiving browsers) to securely identify the software running on the host, i.e., to *measure* the integrity state of the system by identifying its software. Along with the TPM, some form of integrity measurement system, such as the Linux Integrity Measurement Architecture [32], is needed to create full attestations of the running system state. The mechanism used by the TPM to provide integrity state is the *quote* operation [39]. Each quote provides an iterative hash of the code loaded as recorded by the tamper-resistant hardware platform configuration registers (PCRs). The TPM signs the PCR state and a 20-byte *challenge* using a public key associated

with the host. The challenge provides freshness of the quote (the remote party initiating an attestation offers a challenge as a nonce). We observe that *the quote challenge can be used for other purposes such as binding data to the integrity state of the server that created or delivered it.*

In this paper, we explore the requirements and design of the Spork[1] web server service that supports scalable delivery of web content from integrity-measured web servers. Web documents are cryptographically bound to a TPM-based integrity state proof of the server software. The proof is generated from a cryptographic hash of the content, a timestamp retrieved from an integrity-verified time service, and other meta-information. Client browsers (in practice, Firefox extensions) retrieve proofs by acquiring a document indicated in the target page's meta-information and validate them using the appropriate authority keys.

A naive implementation of this approach would not work well in practice. The cost of performing a TPM quote *per request* is extraordinarily high–on the order of 900 milliseconds. We address this limitation by using cryptographic dictionaries to efficiently generate content proofs. Cryptographic dictionaries requiring only a single integrity quote are created periodically. Succinct proofs are extracted from the dictionary and delivered to requesting clients. Because such dictionaries can be created frequently (in under a second), proofs for both dynamic and static content can be created efficiently and delivered to clients.

A detailed analysis of the performance of the Spork system illustrates the costs associated with the delivery of proofs for static and dynamic web pages. Here, we explore

[1]Not quite a web service, not quite a security service.

optimizations that reduce the "bytes-on-the-wire" and computational overheads. Our experiments show that the Spork system can deliver static documents with integrity proofs at near line-speed, where the throughput of an integrity measured web server reaches almost 8,000 web objects per-second—16.5% of an unmodified Apache server's throughput. Moreover, we show empirically that the same content can be delivered with as little as 2.7 milliseconds latency. Because dynamic documents must be bound to the current state of the system at the time it is requested (they cannot be pre-computed), their delivery is limited by the TPM. We introduce optimizations to amortize these costs across requests and over embedded objects within the same web page. Further experiments demonstrate that a single Spork-enabled web server serving dynamic pages can sustain almost 7,000 web objects per-second with approximately 1,000 msec latency (most of which is attributable to the TPM).

An interesting aspect of Spork content proofs is that they can be used asynchronously. Proofs acquired from the web server can be cached with the content itself, e.g. in a Squid cache [7]. Because each proof includes a timestamp acquired from a globally accessible time service, the browser can make a policy decision on whether the cached proof is stale or not. If it is not, the content and proof can be used as if they were obtained from the server. Otherwise, they can be discarded and new ones acquired from the web server. Note also that such policies can be transparently implemented by web proxies via TTL policies.

1.2 Background

Content served over unsecured HTTP provides no indication as to whether the server or the communication channel have been compromised. If the content is served over an SSL connection, either directly or via a proxy [24], the security is predicated on a certificate that vouches for the authenticity of the web server. The guarantees are linked to the machine rather to the content itself, thus leaving no method of knowing whether the content itself has been manipulated, e.g., by a rootkit or corrupt server patch.

Providing guarantees on a system's state requires *measurement* of the system's integrity. Many efforts for ensuring integrity measurement exist, including Pioneer [33], CASS Security Kernels [27], TrustedBox [21], Copilot [30], and LKIM [25] among others. Secure processors such as AEGIS [35] and the IBM 4758 [16] provide a secure execution environment that can be used as a basis for deploying secure services. As an example, we examine integrity management using the Linux Integrity Measurement Architecture (IMA) [32] for attesting the state of the code executed and running on a system, as IMA does not require changes to programs and its only hardware requirement is the presence of a commodity TPM, which are readily available on desktop and server systems. In brief, the system is measured by taking a SHA-1 hash over every pertinent executable file, a process that begins at system startup, when the BIOS and boot loader are measured. The measurement process continues during the boot process to include the operating system kernel and loaded modules, and upon boot includes all executed applications and supporting libraries. These hashes are collected into a measurement list, which provides an ordered history of system execution.

The measurement list is stored in kernel memory but to prevent tampering, the aggregated hash value is stored on a TPM, which provides protected registers known as Platform Configuration Registers (PCRs). These can only be modified by either rebooting the system, which clears the PCR values to 0, or by the *extend* function, which aggregates the current content of the PCR with the hash of the executable to be included, hashing these values together and storing the resulting hash back in the PCR. The TPM provides reporting of PCR values through the *quote* operation. To prevent replay of the measurement, the requestor issues a 160-bit random nonce to the attesting system, creating a challenge. The TPM has a Storage Root Key stored inside it, which only it knows. It uses this key to generate an Attestation Identity Key (AIK), which comprises an RSA key pair, the public portion of which (AIK_{pub}) is available through a key management interface. The TPM is bootstrapped by loading the private portion of the AIK pair (AIK_{priv}) and performs the *Quote* function, where it signs a message containing the values of one or more PCRs and the nonce with AIK_{priv}. The attesting party can then verify the integrity of the message using AIK_{pub}, and subsequently, every element of the measurement list up to the value stored in the PCR may be validated.

Measurements of the system detect deviations from known good software. For example, the Random JavaScript Toolkit is a rootkit that affects Linux-based Apache servers [13]. It contains a small web server that injects JavaScript into an HTML file after being served by Apache but before it is transmitted to the victim. Under IMA, the binary would be added to the measurement list when it was loaded, and the measured value received would differ from the known-good hash. Similarly, if a malicious or unauthorized patch was made to a system binary, or if an unapproved or outdated

binary was being used, these would be discovered through measurement and comparison with the known good hashes.

A byproduct of the content integrity information is that it also protects against "in-flight" page modifications, e.g., within web caches. In [31], the authors show that the content of web pages is modified in a number of different ways including advertisement injection, such as provided by the NebuAd service [5]. Our system is able to ensure that "in-flight" page changes are discovered. The authors identify several other classes of modifications, including page modifications such as image distillation [18] or advertisement removal by a proxy [1, 6], and also types of malware that modified pages viewed by the user, such as the Adware.LinkMaker [36] which creates links in the page that the publisher did not include, or W32.Arpiframe [37], which injects content into HTTP streams on a local subnet.

Chapter 2

Design

In this section, we provide a detailed description of an architecture for scalable web content attestation. A central observation is that to date, attestation-based systems present a challenge to the TPM in the form of a randomized nonce, in order to receive a TPM quote. The nonce ensures the freshness of the quote but provides no semantics beyond that. In our system, by contrast, we *directly tie the content to the system's integrity state* through the use of a *cryptographic proof system* that succinctly represents the content served; this is used along with the current time as a challenge to the TPM. In this manner, we provide stronger guarantees about content origin, and when it was served, than have been found in past proposals.

2.1 System Overview

An overview of the system architecture is shown in Figure 2.1. The core elements of the system are *a*) a web server that generates static or dynamic web content and provides clients with content integrity proofs, *b*) a time server that supplies the web server with an attestation of the current time, providing bounds on when the web server's attestations were generated, and *c*) a web browser client, to which we have added an extension that verifies the proofs received from the web server and can directly query the

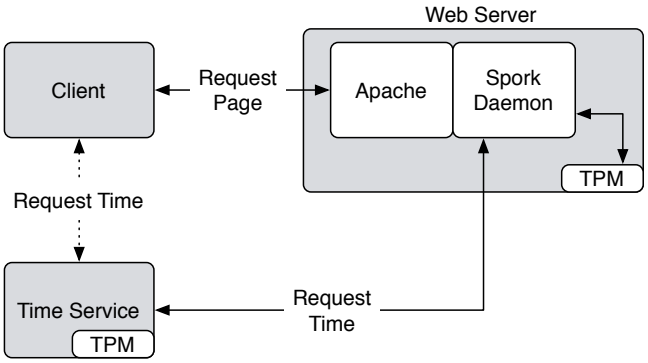

Fig. 2.1. An overview of the system architecture for asynchronous attested content. The time server provides an attested timestamp to the web server, which uses this to provide integrity-measured content to the clients. The web browser can directly verify the current time from the time server.

time server over a secure connection (e.g., SSL) to independently verify its attestation. The system operates as follows:

- A client requests a page from the web server, which returns the content and a URL to the content attestation.
- The server hashes a TPM quote from the time server concatenated with a cryptographic proof system similar to an authenticated dictionary [29]. It uses the resulting hash as a challenge to the TPM to generate a system attestation.
- The client acquires and validates attestations from the web server and the time server, and computes the root of the cryptographic proof system based on the proof received from the server.

The rest of this section describes how content proofs are generated and scheduled, and in the next section, we describe in greater detail how each of the system components are implemented and how they operate.

2.2 Content Proofs

Each document received by a client is tied to the integrity state of the web server via its *content proof*. Ideally, we desire a proof with the following semantics: the proof should state *a*) that a particular page was served by a given web server, *b*) that the web server had a verifiable integrity state (which can be assessed for validity), and *c*) that the binding between the page and integrity state occurred at a verifiably known time. For ease of exposition, we begin with a simple proof and build toward more semantically rich and efficient constructions that provide these properties.

First, let us introduce the notation used throughout. The function $h(d)$ denotes a cryptographic hash over some data d, and concatenation of different data elements is denoted as $|$. The quoting hosts are denoted H_w for the web server and H_{TS} for the time server. pcr_i denotes the integrity state of host i. A TPM quote is denoted $Quote(h, s, c)$, where h is the host identity performing the quote, s is the PCR state, and c is the quote challenge.[1] The served pages are denoted p_i, where each i represents a unique page. t_i is a time epoch returned from a hardware clock on the time server. Lastly, described below, CPS_r represents the root node of a cryptographic proof system and $Pf(p_i)$ is a succinct proof for page p_i from that system.

[1] In practice, the quote mechanism uses *attestation identity key* (or simply the *signing key*) to perform the quote. Thus, the key acts as a proxy for the host. For the purposes of this section, we blur this distinction between the host and the signing key.

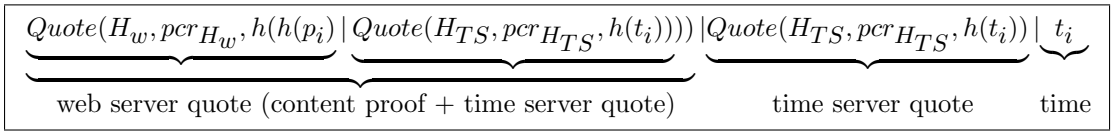

Fig. 2.2. A content proof construction that ties content to both the originating host and the time.

Consider a simple content proof to be received by a web client from a web server for a page p_i, as follows:

$$Quote(H_w, pcr_{H_w}, h(p_i))$$

The quote operation provides a clear binding: document p_i was generated by (or is at least present on or known to) H_w with PCR state pcr_{H_w}. Of course, the proof is not tied to any particular time. In tangible terms, properties a (web server identity) and b (integrity state) from above are provided. What is missing from the simple proof is c (the element of time). Thus any page delivered to a client at any time could be replayed forever, i.e., a compromised server delivering stale content could not be detected.

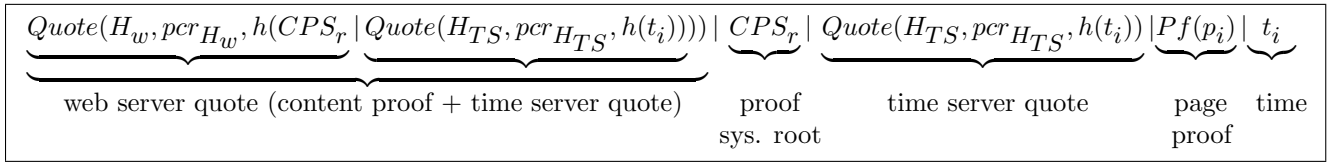

Fig. 2.3. Extended content proof that uses a cryptographic proof system as the challenge rather than a document hash. A succinct page proof is also included.

Figure 2.2 describes a more semantically rich content proof construction that simultaneously ties content to both the host and time. In this, the *time server* acts as a *root of trust* in providing a self-certified timestamp (that uses the timestamp itself as the quote challenge). The time server is trusted to provide the correct time (by definition of a root of trust [34]), and its quote mechanism is a means of tying a specific timestamp to that trusted service. We revisit the design and security issues of the time service in Section 3.2.

During the validation process, the client acquires a timestamp from the time server directly (or uses a suitably fresh timestamp it recently acquired). The client will then judge whether the content is too stale to trust, i.e., the difference between the timestamp in the proof and that received from the time service is too great. Because the time service is trusted, the client can securely make judgments on content validity based on loose clock synchronization, e.g., as seen in Kerberos [28]. Thus, we have provided a proof whose semantics provide all of the required properties.

The central limitation of proposed content proof construction is cost. Web servers may receive many hundreds or thousands of requests per second (RPS). The above proof would take about a second to generate on commodity hardware (including the RTT delay to acquire the timestamp and the 900 *msec* for the quote operation in our test environment). Because a unique proof is needed *per page/timestamp*, the web server would not be able to serve content at a reasonable rate, i.e., the web server RPS would be ≈ 1. What is a needed is a means to amortize the quote costs.

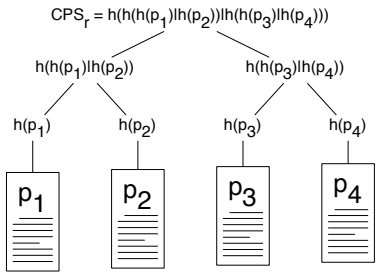

Fig. 2.4. A Merkle hash tree base for the cryptographic proof system. The leaf nodes are hashes of the pages served to clients.

A *cryptographic proof system* is a construction used to efficiently authenticate collections of objects using one or more cryptographic operations. Objects can be validated by extracting succinct proofs from the proof system. These succinct proofs are generally significantly smaller that the proof system as a whole. Thus, authentication costs are amortized over collections of objects. While more sophisticated techniques exist [29, 19], we concentrate on a conceptually simple proof system based on the Merkle hash trees [26]. We create a proof system for all of the documents that will be served by the web server. Assume for the moment that the web server has a static collection of pages that it delivers to clients (we extend our solution to dynamic content generation in the next section). To create the proof system for these static documents, all of the documents are arranged as an ordered sequence of pages $p_1 \ldots p_n$. As shown in Figure 2.4, a binary tree is initially constructed by assigning the hash of each page $h(p_i)$ as a leaf, and each interior node is the hash of the concatenation of both its children. The root node is CPS_r. The succinct proof for page p_i, denoted $Pf(p_i)$, consists of the root node and all of the siblings on the path to the root. For example, the proof system for page p_3

in Figure 2.4 is $\{h(p_4), h(h(p_1)|h(p_2)), CPS_r = h(h(h(p_1)|h(p_2))|h(h(p_3)|h(p_4)))\}$. A receiver of the proof can then validate the content by hashing the file and computing the p_3 leaf and interior nodes on the path to the root sequentially. If the computed hash root is the same as in the proof, then the page is the one used in the original proof system. The proofs are 'succinct' in the sense that they grow logarithmically in the number of documents in the proof system, i.e., the size of the proof is $((\log_2 n) + 1) * H + S$, where H is the size of the hash output and S is the size of the signature.

The proof system is used to generate an extended content proof for page p_i is shown in Figure 2.3. The two differences between this construction and the preceding one are that the CPS_r is used as the challenge (instead of a document hash), and that a succinct proof for p_i is included. Because a single quote is used to bind any number of pages to the time quote and host integrity state, we can efficiently support serving a large body of pages. As we discuss below, the challenge is knowing exactly what the body of documents is.

2.3 Proof Scheduling

Content proofs are delivered to browsers through *integrity proof pages*. The web server inserts an extenstion `X-Attest-URL` HTTP header in each delivered page whose URL points to a proof for that page. The browser parses the header, retrieves the proof from the web server, and validates the proof. If the validation fails, the browser can log the error, notify the user, refuse to display the page, or perform any other action the browser or user deems appropriate. We discuss the design and operation of the Firefox-based client software in section 5.2.

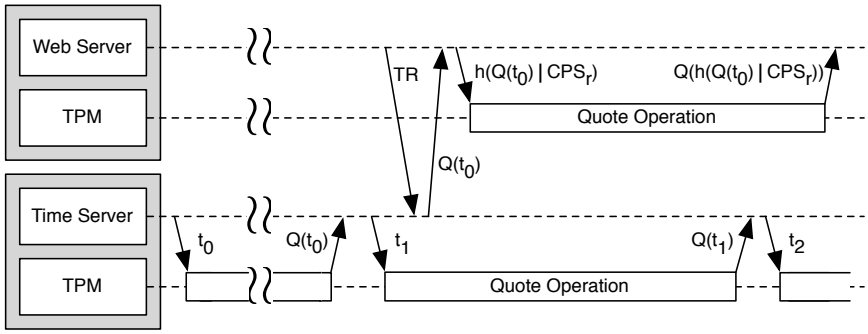

Fig. 2.5. Server quote generation - The server requests the most recent timestamp from the time server ($Q(t_0)$), and then generates a quote using the most recent hash tree computed (CPS_r).

Determining what pages should be included in a proof system is essential to supporting the browsing community. Static web pages represent the simplest case. As illustrated in Figure 2.5, the web server generates a Merkle hash tree of all pages it will be serving to clients. The web server will then generate proofs at the rate at which the timestamps are received from the time service, e.g., once a second. When a browser asks for a proof for a given page, the succinct proof is extracted from the most recent proof system completed and returned to the browser, as shown in Figure 2.6. A proof is always available because the content is unchanging. Thus, the latency induced by the integrity proofs is bounded by the proof acquisition (a web page `GET`) and browser validation costs (described in more detail in Section 4).

Dynamic content presents other challenges. Centrally, the page content only becomes available after the request arrives from a client. For example, consider a `.php` [8] web page. PHP allows the web designer to create content programmatically. The inputs

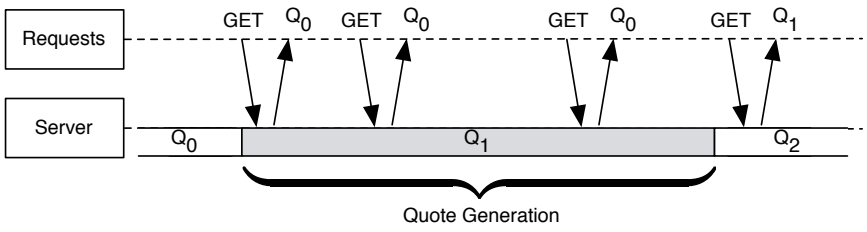

Fig. 2.6. Static Page Scheduling - For static pages, the server provides the most recently generated quote (Q_0) to all incoming requests while it is generating the next quote. Once the next quote is generated (Q_1), this new quote is provided to each incoming request.

to this process include referrer page, URL, HTTP `POST` fields, database contents, cookies, and other information. Because the inputs are unknowable, precomputation of pages is infeasible in many cases, and the web server must create integrity proofs in real time.

As illustrated in Figure 2.7, our approach is to exploit the periodicity of quote generation. The web server creates and delivers content through dynamic generation interfaces (e.g., PHP, mod_perl [2], mod_python [3]) as in normal operation. However, the proof identified in the `X-Attest-URL` header identifies a proof that does not yet exist. The web server caches hashes of the dynamic content delivered since the last quote was completed. As soon as the TPM becomes available (by completing a previous quote), a hash tree of recent dynamic content is generated and used as the challenge to the quote generation. The proof system becomes available as soon as the quote operation completes.

The browser will observe additional latency when receiving dynamic content. Assuming a 900 *msec* quote operation (which is the case in our test environment) and uniform distribution of arrivals, the expected latency would be about 1350 *msec* plus

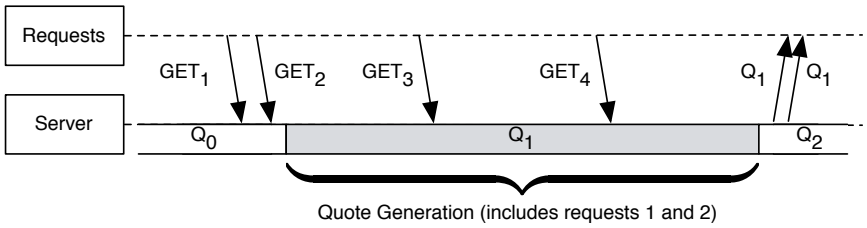

Fig. 2.7. Dynamic Page Scheduling - Incoming requests for an *integrity proof page* are delayed until the quote including the page is ready. At this point, a hash tree is generated that includes the cached requests (GET_1 and GET_2) and the hash tree is used to generate the next quote (Q_1).

the time to deliver the quote itself (which is network dependent). More specifically, the expected arrival in the previous quote epoch is $0.5 * 900 = 450$ *msec* plus the quote cost itself 900 *msec* is the expected delay observed by a browser. Note that this will be interleaved with the delivery (and possibly rendering) of the content itself, and thus the observed delay may be somewhat less.

Most web servers simultaneously support static and dynamic content. The above processes can support this operation by simply joining the static and dynamic hash trees at the root, and using the resulting hash as the challenge. In all other respects, the web content is processed as before—proofs for static content can be extracted from the most recent proof system, while proofs for dynamic pages will become available at the completion of the following quote epoch. No other modifications to the web server are needed.

Chapter 3

Implementation

We have developed a version of the architecture detailed in the preceding sections that supports static, dynamic, and mixed content. Figure 3.1 shows the structure of the Spork web environment. In addition to external clients and the time service, there are two functional elements processing the client requests on the web server host; the web server and a *Spork daemon*.

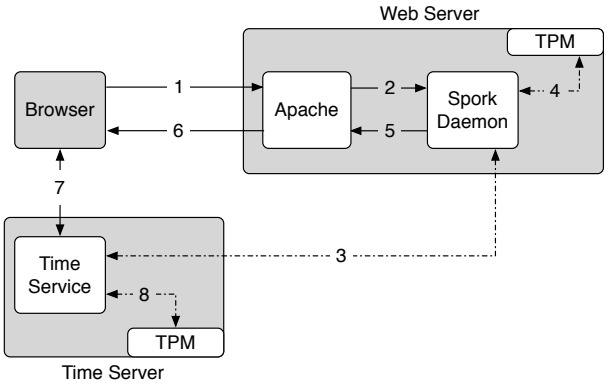

Fig. 3.1. An overview of the Spork system architecture – The time server provides an attested timestamp to the web server which is bound to the content delivered to the browser and local software integrity information.

3.1 Proof-Generating Web Server

As directed by the requested URL, the Apache web server supporting Spork directs all client requests (1 in Figure 3.1) to Spork threads processing requests running in the `httpd` address space. If the request is for a static page, the content is retrieved from the local filesystem. A URL to a proof page (which may not yet exist) is inserted into the `X-Attest-URL` HTTP header of the retrieved page, and the result is returned to the client (6). Dynamic requests occur in substantially the same way except that the content is generated automatically using the appropriate content generation code, e.g., ASP [12], rather than being retrieved from the filesystem.

If the received request is for a proof, the Spork request processing thread passes proof identity information to a Spork master thread (one per Apache process) which passes the proof request to the Spork daemon over standard UNIX IPC (2) (i.e. sockets). The processing thread then sleeps waiting for a "proof ready" event. When the requested proof (5) is received by the master thread from the Spork daemon (see below), it fires a directed event that wakes the processing thread. The woken thread then returns the proof to the client (6).

The *Spork daemon* generates the content proofs by interleaving a number of utility threads. The main thread receives requests from Apache, extracts and marshals the succinct proofs from available proof systems, and returns the result to the main Spork thread in Apache (5). The remaining threads update the internal state from which the proof systems are constructed. A TPM thread schedules and execute quote operations (4) as governed by the algorithms defined in Section 2.3, and a separate time thread similarly

retrieves time attestations (3). Separate threads further maintain the dictionary of static documents (by monitoring the local filesystem) and the current set of dynamic content awaiting proof generation.

Client browsers receive the content proof from the web server (6) and acquire time attestations from the time server (7). If the proofs validate correctly, the page may be rendered. Note that it is a matter of policy of what to do when a proof validation fails; the browser may block rendering, warn the user, confirm the rendering, or place visual indicators on the display, .e.g, icons or red shading over failed objects. We briefly touch on this policy further in the description of the browser extension in Section 5.2.

3.2 Time Server

The time service uses a hash of the current hardware timestamp as a challenge to the TPM (8 in Figure 3.1). This *time attestation* is provided to requesters such as the web servers for inclusion in content proofs or to clients for clock synchronization, e.g., to detect content replay attacks.

The time server plays a critical role in operation of the system, because of the importance of freshness to verifying attestations. While the web server has a file system that is mutable, due to the ability to add, delete, or modify web files to be served, the time server's file system can become largely static after it is installed. As a result, we can provide deeper validation than what is afforded with typical integrity measurement. We provide trust guarantees from the system clock all the way to the software, forming a *time root of trust* in a similar manner to how a root of trust installer fully guarantees the system from installation up to applications [34]. This approach provides a smaller

base of components that need to be trusted: the BIOS core root of trust measurement (CRTM), the TPM, and the clock.

Another requirement solved by this approach is the ability for the client to directly verify the attestation from the time server itself. If the client establishes an SSL connection with the time server, it can receive the same time update that is presented to the web server, allowing confirmation of the validity of the time attestation and verification of functionality. Once the client has established trust with the time server, it can rely on attestations that are carried in the HTML document presented to it by the web server.

Chapter 4

Evaluation

In this section, we empirically evaluate the performance and scalability of the Spork system presented in the preceding sections. We begin by measuring the throughput and latency of the system compared to an unmodified Apache web server, and expose the underlying costs via microbenchmarking. We propose a number of optimizations and evaluate the performance impact.

All tests were performed on Dell PowerEdge M605 blades with 8-core 2.3GHz Dual Quad-core AMD Opteron processors, 16.0GB RAM, and 2x73GB SAS Drives (RAID 1). Six blades running Ubuntu 8.04.1 LTS Linux kernel version 2.6.24 were connected over a Gigabit Ethernet switch on a quiescent network. One blade ran Apache web servers (one normal install and one running the integrity proof system described in the preceding sections). One blade ran the time server, and four were used for simulated clients. All experiments use the Apache 2.2.8 server with `mod_python` 3.3.1 modules for dynamic content generation. The Spork daemon is written in Python 2.5.2 and uses a custom TPM integration library written in C. The server and client browser extension exceeds 5000 lines of code. All load tests were performed using the Apache JMeter benchmarking/testing tool.

A recent study of web pages indicated that the average web page size is about 130KB total, with an average HTML source size of 25KB and the average non-flash

object being just under 10KB [10]. More focused studies of popular websites indicate somewhat larger total sizes ($\approx$ 300KB) [9]. The sizes of the component objects (e.g., images) in popular websites is essentially the same as reported in the broader study, with the increases in the number of embedded objects accounting for the larger total page size. Thus, we use 10KB and 25KB file sizes in all experiments reported in this section.

Figures 4.1 and 4.2 show the throughput of an unaltered Apache web server in requests per second (RPS) serving static and dynamic pages to communities of clients of different sizes. In dynamic experiments, client requests are delayed a random period (up to the TPM quote period, 900 msec) before requesting another page. This ensures uniform arrival of requests at the server[1], but necessitates significantly more clients to sustain maximal throughput. After experimenting with a number of different client community sizes, we found the highest throughout could be achieved in static experiments with 500 clients and dynamic experiments with 4,000 clients without incurring significant latencies. Thus we use 500 clients to drive all static tests and 4,000 for all dynamic tests.

For static workloads, analysis shows that the relationship between the throughput and the filesize is not a linear relationship. This indicates that the bottleneck is not just bandwidth, but some processing is occuring as well. This occurs because every file served is loaded instead of being cached. This is confirmed by looking at the Apache 2.2.11 source code. The function in question is in `server/core.c` and is `default_handler()`. This function is responsible for serving static content, and calls functions to open the file each time it is served. Based on this, we examine the relationship between the

[1]Failure to evenly distribute request arrivals in dynamic tests leads to throughput oscillation. This oscillation causes client requests to arrive in bursts that overwhelm queues and cause synchronized retransmissions. Randomized arrivals of client proof requests will dampen oscillation.

throughput and the filesize. The measured throughput for filesizes ranging from 1KB to 200KB is depicted in Figure 4.3.

4.1 Macrobenchmarks

Our first set of experiments sought to identify the overheads associated with the delivery of integrity proofs by comparing operation of Spork with that of an unaltered web server. The *static* content and *dynamic* content web servers use out-of-the-box installations delivering static and dynamic content, respectively. The dynamic content is generated using `mod_python`. The *integrity-measured* web servers operate in substantially the same way as the static and dynamic web servers, except that each system creates and delivers integrity proofs with the content. Clients in the integrity-measured experiments receive the content as in normal web server operation, then retrieve the associated proof from the web server as indicated in the `X-Attest-URL` header. Thus, integrity measured content consists of two serial requests—one for the original request and one for the proof.

Figure 4.4 shows throughput of an unaltered web server measured in *requests per second* (RPS). The throughput of the 10KB static content (average 10,770 RPS) has about 29.4% higher throughput than the dynamic case (average 7,600 RPS) for 10KB web pages. Such throughput disparities are not atypical in web systems. The additional overheads are due to forking and using a `mod_python` interpreter. This disparity is further amplified by the static content being delivered from in-memory caches in all tests, i.e., the web server can easily hold all experimental static content in memory. The throughput of the web server serving non-integrity measured 25KB pages for dynamic content are 4,485 and 4,510 RPS for static and dynamic content, respectively. For the 25KB tests,

the network has reached saturation and as such the static and dynamic tests provide similar throughput.

A comparison of the relative throughput of the web server in the static and dynamic content costs highlights the bottlenecks associated with each content type. For example, the number of bytes sent per second by the web server serving static content of both the 10KB and 25KB pages is essentially the same:

$$10,770 * 10 = 107,700 KB/s \approx 4,485 * 25 = 112,125 KB/s,$$

where 5% more "bytes on the wire" are delivered by serving larger web pages. This slight advantage can be accounted for by overheads of processing individual requests (there is 2.5 times more per-byte HTTP protocol overhead in 10KB web pages). This indicates that the bottleneck in the static case is bandwidth. From Figure 4.2, we see that performance does not change drastically from when varying the file size until the network becomes saturated. This indicates that the dynamic content throughput is limited by computation, and not by bandwidth.

Illustrated in Figure 4.5, the average throughput of the integrity-measured web server is around 900 RPS in the static case and 540 in the dynamic case. The overheads relate to the creation and acquisition of proofs by the Spork daemon and their insertion in response web objects. In addition, each request involves serial requests and responses. However, opportunities exist to amortize these costs.

In the integrity-measured dynamic content experiments, there is osciallation. This arises from the workload being used to drive the server. The clients make requests for content, which results in a high throughput, followed by requests for proofs over this requested content. When making this request, the client is forced to wait for the TPM

to finish generating the proof covering their content. As such, these clients are forced to wait during the next second resulting in significantly lower throughput. Once they receive their proof, the clients immediately make their next content request. This patter repeats over and over leading to drastic oscillations between a high and low throughput[2]

Table 4.1 shows minimum observed latency. To compute these latency statistics, we averaged measurements over 150 trials in a system with a single client requesting a single page. The latency represents the time from the first byte sent from the client to the reception of the last byte of the response. Unaltered web latencies range from 490 μsec to 5.4 msec. The latencies observed in the static integrity measured case averaged about 3 msec, where the additional latency can be attributed to multiple HTTP RTTs and the costs of acquiring the proof from the Spork daemon. The dynamic integrity measured latencies were lower than expected values (as discussed in Section 2.3), about 1000 msec. These longer latencies are a reflection of the random arrival of the request within the periodic TPM quotations and the time required to create a proof system encompassing the quoted material, e.g., TPM quotation time.

Table 4.2 shows latency microbenchmarks of proof creation in an integrity-measured web server. Recall that the proof system is generated by collecting document, time, and system information over which a TPM quote is taken. Such operations are amortized over all requests during the proof system period (as discussed in Section 2.2), and are not on the critical path of any content delivery. Nearly 99% of the latency involves the

[2]These periods where the osciallations are not as drastic lead to a sinusoidal behavior that is actively being investigated. Figures 4.6 and 4.7 show the behavior and the related number of dynamic pages covered by each quote window.

	Static				**Dynamic**			
	10 KB Pages		25 KB Pages		10 KB Pages		25 KB Pages	
	RPS	Min. Lat.	RPS	Min. Lat.	RPS	Min. Lat.	RPS	Min. Lat.
Base	10769	0.49	4485.0	0.50	7604.6	4.9	4509.2	5.4
IMA	1006.1	3.1	889.0	3.1	537.0	976.2	544.0	1058.5
PRIMA	1230.0	2.9	1061.5	3.0	502.6	1004.2	527.3	899.0
Compressed IMA	1428.5	2.6	1458.3	2.7	510.4	969.2	541.8	1020.7
Compressed PRIMA	1492.5	2.6	1453.4	2.7	514.6	1054.2	570.5	939.8

Table 4.1. Static and dynamic system measurements. Latencies are measured in milliseconds.

	Static	Dynamic
Generate MHT	0.716 (0.08%)	1.9 (0.19%)
Obtain TS Quote	35.9 (3.68%)	34.9 (3.58%)
Generate Quote	938.4 (96.24%)	938.8 (96.23%)

Table 4.2. Proof creation latency micro-benchmarks – latency of proof system generation measured in milliseconds. For the static content, a pool of 125 files was used.

acquisition of the time quote and the local quote operation.[3] These operations are external to the web server processing. The remaining operations are insubstantial in terms of latency and computation. As a result, proof system creation has little impact on the throughput of the web server. Thus, our only hope at improving web server throughput is to address the network and computation bottlenecks within the content delivery process itself.

4.2 Bandwidth Optimizations

Because we cannot modify the pages directly, we limit bandwidth use by reducing the size of the returned proofs. The proofs are large ASCII XML structures in which the vast majority of content fields are integrity hashes. Because the ASCII text is highly redundant, compressing it could reduce the size of proofs considerably. Conversely, the Policy-Reduced Integrity Measurement Architecture (PRIMA) [22] provides for smaller attestations by reducing the size of the measurement list to include only the specific applications of interest, and can thus be used to significantly reduce the number of

[3] Recall that the time server simply returns the most recently created time quote. Thus, the latency for acquiring a time proof is largely determined by the RTT between the web and time servers, and *not* the time to create the time attestation (964 *msec*).

integrity hashes included in a quote[4]. We consider the performance of our web server under these strategies: *compressed IMA* compresses the proofs described in the preceding sections before transmitting to the client, *PRIMA* implements PRIMA for proofs, and *compressed PRIMA* compresses the PRIMA proof. We include the performance of a web server delivering the content proofs used in the preceding experiments as *full IMA*.

The different optimizations reduce proof size as follows. The baseline full IMA generates an 107 KB proof and the full PRIMA reduces to 82k. The reason that the reduction is not very large is that the test environment is already fairly minimal, where the number of measurements needed is smaller than in systems with more services, e.g., database systems. Thus, the policy reduction only removes a handful of services from measurements. Compressing the proof was much more successful, where the IMA and PRIMA proofs were reduced to 32 and 25 KB, respectively.

Returning to Table 4.1, the throughput the web server improves under these bandwidth optimizations. Compression of static content clearly improved throughput. Simply compressing the proofs results in 5-64% increased throughput. The compressed PRIMA proofs performed the best (64% increase). These optimizations had negligible effect on throughput of web servers serving dynamic content because bandwidth is not the bottleneck.

Compared to the delivery of static content on an unaltered server, a web server delivering compressed PRIMA proofs will still observe over 80% overhead for 10KB page and 70% in 25KB pages. This is largely due to every integrity-measured static page

[4]Additional information about the XML structure and PRIMA can be found in the Appendicies.

requiring the processing and delivery of one static *and one dynamic* page: one for the content and one for the proof. While compression techniques mitigate the delivery of the dynamic page, it does nothing to mitigate the computational costs of its creation. Thus, our next best hope is to alter the relationship between the number of requested pages and requested proofs.

4.3 Proof Amortization

Recall that prior studies of web pages show that an average page has one base HTML page and just over 10 static 10KB embedded objects. As a matter of practice, a client requesting that page will obtain the base page and all of its embedded objects for rendering. This reality presents an opportunity: a proof for a web page can be computed over the base document and all embedded objects at once. Thus, we can amortize the costs of proof generation over all elements of a web page, significantly reducing the number of proofs requested by a client.

Consider a naive calculation of the expected per-second web server throughput under this discipline. The expected throughput of a web server $\mathcal{P}$ could be computed in pages as:

$$\mathcal{P} = \frac{1}{\left(10 * \frac{1}{\mu}\right) + \frac{1}{\epsilon}}$$

where μ is the service time for a web server serving a 10KB static object and ϵ is the service time for the web server serving static (dynamic) 25KB HTML files. The model assumes that the unit "cost" per web object on a hypothetical throughout budget is fixed and independent of other documents.

	μ	ϵ	**Expected**		**Actual**	
			$\mathcal{P}$	Web Objects	$\mathcal{P}$	Web Objects
Base Static	10769	4485.0	868.4	9552.3	868.3	9551.5
Base Dynamic	10769	4509.2	869.3	9562.2	744.8	8192.8
Integ. Measured Static (Full IMA)	10769	889.0	487.0	5356.8	484.5	5329.9
Integ. Measured Static (Comp. PRIMA)	10769	1453.4	618.6	6804.3	724.7	7971.8
Integ. Measured Dynamic	10769	544	361.4	3975.7	485.4	5339.2
Integ. Measured Dynamic (Comp. PRIMA)	10769	570.5	372.9	4102.3	633.4	6967.2

Table 4.3. Proof Amortization Performance – the expected and measured performance of the amortized proof serving.

Table 4.3 shows the expected and experimentally-measured "real" throughput of the amortized proofs. We show the parameters in terms of throughput (i.e., the inverse of the service time) for clarity, with the expected throughput computed using the measurements presented in Table 4.1. Interestingly, the model underestimates throughput considerably in most cases. This is because the computation fails to model both bottlenecks at the same time, and thus misses the positive effect of interleaving requests for content (limited by bandwidth) and content proof acquisition (limited by computation). Practically speaking, the costs of finding and delivering proofs from the Spork daemon to the web server are hidden by bottlenecked delivery of content. Thus, a web server providing integrity measurement of content can achieve web object throughputs within 16.5% of the maximum web server.

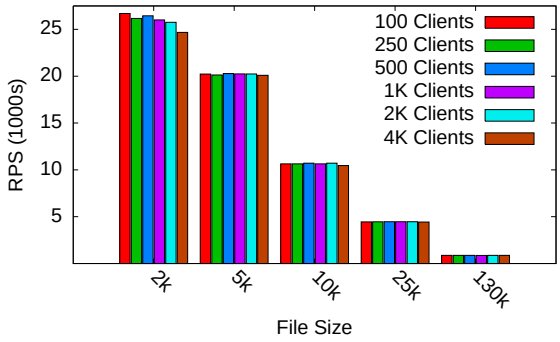

Fig. 4.1. Static content throughput – throughput of unaltered Apache web server serving static web pages of varying sizes to communities of varying clients.

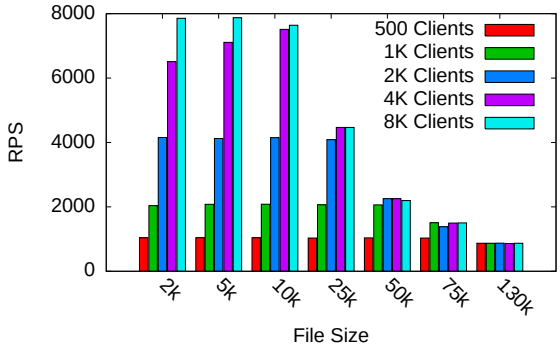

Fig. 4.2. Dynamic content throughput – throughput of unaltered Apache web server serving dynamic web pages of varying sizes to communities of varying clients.

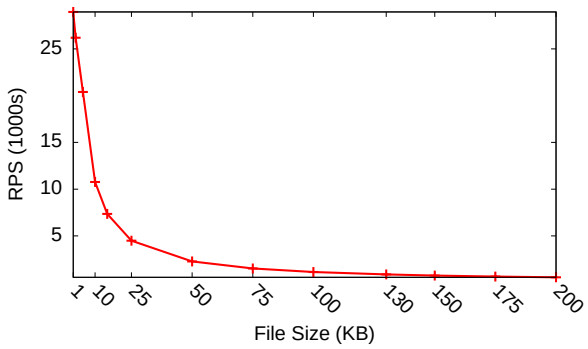

Fig. 4.3. The relationship between static filesize and throughput.

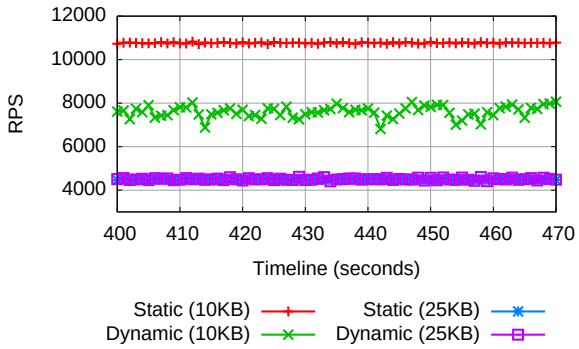

Fig. 4.4. Unaltered web server throughput – sustained RPS during a 70 second experiment.

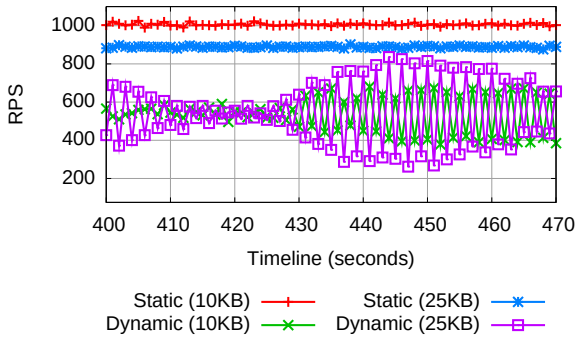

Fig. 4.5. Integrity measured web server throughput – sustained RPS during a 70 second experiment.

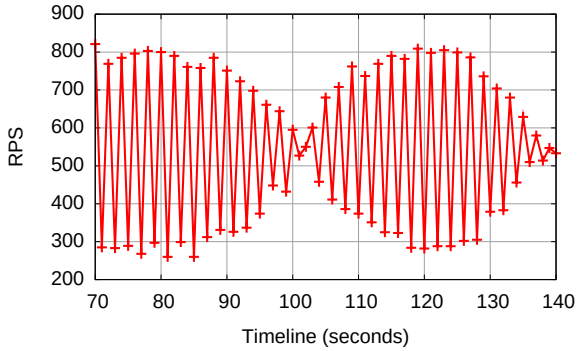

Fig. 4.6. Unaltered web server throughput – sustained RPS during a 70 second experiment.

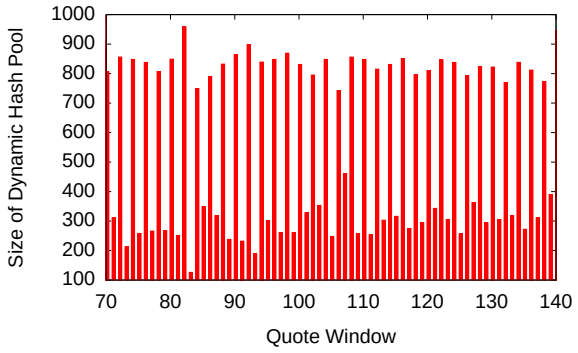

Fig. 4.7. Integrity measured web server dynamic hash pool size for each TPM quote window during a 70 second experiment.

Chapter 5

Conclusion

5.1 Discussion

5.2 Client Extension

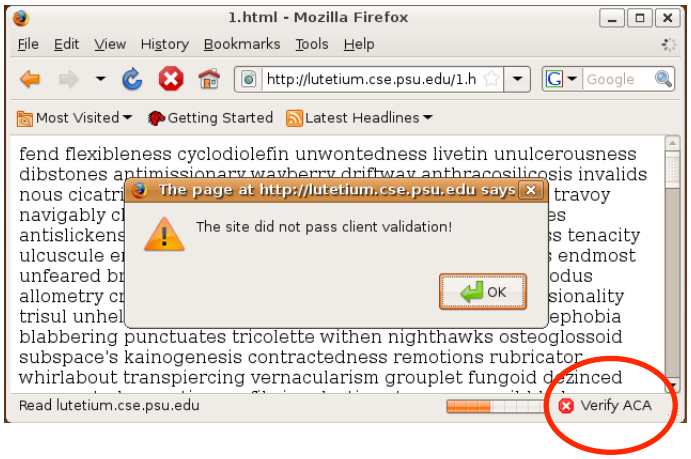

Fig. 5.1. Dialog notifying user of an invalid content proof

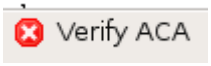

Fig. 5.2. Taskbar icon indicating invalid content proof

Fig. 5.3. Taskbar icon indicating a valid content proof

Firefox is a commonly used web browser that can be customized through the use of extensions. Extensions have access to browser internal state through interfaces like the Cross Platform Component Object Model (XPCOM) [4]. Most extensions are

implemented using a combination of these XPCOM components[1] and JavaScript. Depending on the purpose of the extension, Firefox invokes the extension in response to events occurring, such as page loads.

Our Firefox extension validates content proofs provided acquired from the modified web server at page load. The extension examines the `X-Attest-URL` header after the page loads. If this header is correctly formed, the associated content proof is requested from the web server and validated. First, the extension validates the system attestation from the web server and the attestation from the time service. Once the system and time attestations are validated, the succinct content proof is checked by reconstructing the hash tree from the provided nodes and the downloaded content. Once the root of the tree is computed, it is compared to the value provided in the signature. Once everything is validated (or invalidated), the user is notified by some simple icons on the status bar of Firefox, similar to Privacy Bird [14].

The Firefox interface is modified as shown in Figures 5.1, 5.2, and 5.3 In Figure 5.1, we see a page that is loaded, and the user has been notified via a dialog box that the validation of the content proof has failed. The user is still shown the page, but is aware that the page is invalid. This is similar to Firefox's default operation of allowing a user to view a page even if the server-side SSL certificate is invalid. In Figure 5.2, we use a simple icon to indicate the status of the content validation. In this case, X in the taskbar indicates a failed validation. Finally in Figure 5.3, a check mark indicates that the content proof tied to the page is valid.

[1] XPCOM is merely an API. Language bindings exist for a number of languages including C++, Java, and Python

The system requires that web server and the time server TPMs keys and verification measurement lists be loaded at installation. In real deployments, it is likely that the clients will be bootstrapped with a separate public measurement signing key associated with the services they are measuring. This key would be used to sign measurement lists provided periodically by administrators and possibly provided through the web server as separate URLs. Administrative systems supporting integrity services are being actively studied by the integrity measurement community, and we will make use of these systems as they become available.

5.3 Improving Caching Mechanisms

All proofs and content in our system are cacheable and transparent to the server and browser. Currently, the proof system associated with the current attestation is kept in memory. The proof generation code is called to determine whether the validity period of the attestation has expired. This code may cache the succinct proofs associated with a particular document being served to avoid re-computation when a request is made. When serving static web content, these proof systems may stay unchanged for long periods of time, and the succinct proofs associated with the documents can themselves be stored in an LRU cache.

In the case of dynamic content generation, if a client requests the same document in different quote generation windows, a *smart cache* could examine the measurement list to determine if the system integrity has changed and if not, provide the client with an already cached attestation and not force the client to wait for a newer attestation.

Being able to cache multiple generated proof systems would thus be advantageous for client responsiveness.

None of these changes will affect cooperative web proxies. For example, if the Cache Digest exchange protocol [20] for cache server cooperation is employed, differentiation of proof versus attested content would merely require different expiration fields for these documents.

5.4 Conclusions

This paper has introduced the Spork system. Spork uses the Trusted Platform Module (TPM) to tie the web server integrity state to the web content delivered to browsers. This allows a client to verify that the origin of the content was functioning properly when the received content was generated and/or delivered. We discussed the design and implementation of the Spork service and its browser-side Firefox validation extension. In particular, we explored optimizations that enable us to mitigate the inherent bottlenecks of delivering integrity-measured content. An in-depth empirical analysis of Spork confirmed the scalability of Spork to large bodies of clients. Spork can deliver almost 8,000 static or 7,000 dynamic integrity-measured web objects per-second with manageable latencies.

We are just now beginning to understand the use of integrity-measurement in web systems. In the future we will explore the extension of Spork techniques to collections of web servers, e.g., web farms, and as a mechanism to provide integrity guarantees over services spanning administrative domains, e.g., mash-ups. The system itself will also evolve, and we plan to extend and apply new cryptographic techniques to further reduce

overheads and increase the flexibility of the system, e.g., partial signatures. Lastly, we are in the processing of building real web-applications that make use the Spork services and study their use in deployed environments.

Appendix A

Attestation XML Example

Figure A.1 is an example content attestation that the client receives. The content attestation contains two system attestations and the succinct proof over the content. Each system attestation contains the list of measurements of binaries from the chosen integrity measurement architecture (either IMA or PRIMA). Following this is the list of PCRs as reported by the TPM. The final piece of the individual system attestation is the values returned from the TPM during the quote operation. In the case of the time service attestation, there is some additional information including the reported time and the machine hosting the time service.

After the two attestations, there is a node list. This node list contains the nodes needed to reconstruct the hash tree. The internal representation of the hash tree was a simple array, with the numbers in the node list indicating the array index of the individual nodes.

```
<?xml version="1.0"?>
<ACA>
 <Attest_List>
  <Attestation>
   <MeasurementList>
    <Measurement num="0">
     <SHA1>43c2d4b9a0eb364cb937bdd045b4a9d8fe74e781</SHA1>
      <Name>boot_aggregate</Name>
      ...
    </Measurement>
   </MeasurementList>
   <PCRList>
    <PCR num="0">01A3AFEBE1A1197897B01F742D343C0BD9A5B807</PCR>
    ...
   </PCRList>
   <QuoteList>
    <Quote type="sCore">
     <Nonce>CCCDA67FA1213DB748781D74BD7BEFAAD14B5D02</Nonce>
     <Composite>
     0003FFFFFF000001E001A3AFEBE1A1197897B01F742D343C0BD9A5B8075B93
     ...
     FFFFFFFF0000000000000000000000000000000000000000</Composite>
     <Fixed>51554f54</Fixed>
     <Version>01010000</Version>
     <Signature>
     5A9D7D0EBE8E15C563DD64DCBFE6C20C128B679D158D0C5BE493DD72E316F9
     ...
     C35CABA587A74A45</Signature>
    </Quote>
   </QuoteList>
  </Attestation>
  <Attestation type="time">
   ...
   <TIME value="1221863157.65"/>
   <SCORE host="130.203.84.234"/>
  </Attestation>
 </Attest_List>
 <NodeList>
  <Node num="127">37c50b78f2959a676f12ba5aeed66c37953377f1</Node>
  ...
  <Node num="2">d861e2bde7baa6e449e99c13da13b26c5aec0093</Node>
  <Node num="0">795c2727a5ffb79a6614b8fc533100f822e2dc71</Node>
 </NodeList>
</ACA>
```

Fig. A.1. Example XML Attestation as seen by the client

Appendix B

Reducing Attestation Sizes

As described in Section 4.2, measurement attestations account for most of the size of the proof returned to the client, and can be as large as the document being served to the client. Reducing the size of these attestations can dramatically improve server throughput. Using the Policy-Reduced Integrity Measurement Architecture (PRIMA) [22] extensions to the Linux IMA can provide for smaller attestations by reducing the size of the measurement list to include only the specific applications of interest, under the assumption that the rest of the system is running in accordance to a mandatory policy. Central to the operation of PRIMA is the concept that information flow policies are in place to reduce the size of the measurement lists to only elements relied upon by the verifier.

Using PRIMA requires additional mechanisms above those necessary for the Linux IMA. Mandatory access control (MAC) mechanisms must be in place on the system to be attested (the *attesting* system). The system collects a list of *trusted subjects*, which must be trusted by the party verifying the system to ensure system integrity. These trusted subjects are collected in a measurement list. The code and data used by trusted subjects is also collected and measured in a list. The MAC policy itself is also measured by PRIMA to ensure correctness of the system and its information flow properties. Finally, programs acting as trusted subjects must support *filtering interfaces*, which can

anaconda_t	auditctl_t	dpkg_script_t	dpkg_t
depmod_t	firstboot_t	getty_t	ifconfig_t
inetd_t	initrc_t	init_t	insmod_t
kernel_t	ldconfig_t	load_policy_t	local_login_t
logrotate_t	mount_t	postfix_master_t	process_uncond_exempt
restorecond_t	rpm_script_t	rpm_t	unconfined_crontab_t
setfiles_t	sshd_t	staff_t	unconfined_execmem_t
semanage_t	sysadm_t	unconfined_t	unconfined_mount_t

Table B.1. The set of trusted subjects

discard low-integrity inputs or upgrade them to a higher level, and the remote party must be aware of the *filtering subject*, which are the only entities capable of receiving low-integrity input. We have defined a MAC policy for server deployment on SELinux, with the types used for our PRIMA policy listed In Table B.1. We started with the list of trusted subjects discovered with the Gokyo policy analysis tool [23], and by manual inspection added types to the list in order to ensure that the code needed by the web server and time server was measured.

References

[1] Ad Muncher: The Ultimate Popup and Advertising Blocker. http://www.admuncher.com/.

[2] mod_perl: Welcome to the mod-_perl world. http://perl.apache.org/.

[3] mod_python - Apache/Python Integration. http://www.modpython.org/.

[4] Mozilla Developer Center. http://developer.mozilla.org/En.

[5] NebuAd. http://www.nebuad.org/.

[6] Proxomitron. http://www.proxomitron.info.

[7] squid : Optimising Web Delivery. http://www.squid-cache.org.

[8] PHP: Hypertext Preprocessor. http://www.php.net, September 2008.

[9] A. King. Average Web Page Size Triples Since 2003, 2008. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

[10] A. King. The Average Web Page, 2008. http://www.optimizationweek.com/reviews/average-web-page/.

[11] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[12] Microsoft Corporation. Active server pages. `http://msdn.microsoft.com/en-us/library/aa286483.aspx`.

[13] cPanel. Components of Random JavaScript Toolkit Identified. `http://blog.cpanel.net/?p=31`, January 2008.

[14] Lorrie Cranor. Privacy bird. `http://www.privacybird.org/`.

[15] DarkAngel. Mood-NT. `http://darkangel.antifork.org/codes.htm`.

[16] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10):57–66, 2001.

[17] D. Eastlake 3rd, J. Reagle, and D. Solo. (Extensible Markup Language) XML-Signature Syntax and Processing. RFC 3275 (Draft Standard), March 2002.

[18] Armando Fox and Eric A. Brewer. Reducing WWW latency and bandwidth requirements by real-time distillation. In *Proceedings of the fifth international World Wide Web conference on Computer networks and ISDN systems*, pages 1445–1456, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[19] Michael T. Goodrich. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *In Proc. 2001 DARPA Information Survivability Conference and Exposition*, pages 68–82, 2001.

[20] Martin Hamilton, Alex Rousskov, and Duane Wessels. Cache Digest specification - version 5. `http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt`, December 1998. Internet Draft.

[21] Pietro Iglio. TrustedBox: A Kernel-Level Integrity Checker. In *Proceedings of the 1th Annual Computer Security Applications Conference (ACSAC'99)*, Washington, DC, December 1999.

[22] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006)*, Lake Tahoe, CA, June 2006.

[23] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[24] Chris Lesniewski-Lass and M. Frans Kaashoek. SSL splitting: securely serving data from untrusted caches. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[25] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux Kernel Integrity Measurement Using Contextural Inspection. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing (STC'07)*, Alexandria, VA, November 2007.

[26] R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, April 1980.

[27] George Mohay and Jeremy Zellers. Kernel and Shell Based Applications Integrity Assurance. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC'97)*, San Diego, CA, December 1997.

[28] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, pages 33–38, September 1994.

[29] M. Noar and K. Nassim. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, pages 217–228, January 1998.

[30] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[31] Charles Reis, Steven D. Gribble, Tadayoshi Kohno, and Nicholas C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, Berkeley, CA, USA, 2008. USENIX Association.

[32] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[33] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, October 2005.

[34] Luke St.Clair, Joshua Schiffman, Trent Jaeger, and Patrick McDaniel. Establishing and Sustaining System Integrity via Root of Trust Installation. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 19–29, Miami, FL, December 2007.

[35] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architectures for Tamper-Evident and Tamper-Resistant Processing. Proceedings of the 17th International Conference on Supercomputing, June 2003.

[36] Symantec.com. Adware.LinkMaker. `http://www.symantec.com/security_response/writeup.jsp?docid=2005-030218-4635-99`.

[37] Symantec.com. W32.Arpiframe. `http://www.symantec.com/security_response/writeup.jsp?docid=2007-061222-0609-99`.

[38] Trusted Computing Group. TPM Working Group. `https://www.trustedcomputinggroup.org/groups/tpm/`.

[39] Trusted Computing Group. Trusted Platform Module Specifications. `https://www.trustedcomputinggroup.org/specs/TPM/`.