Scalable Web Content Attestation

Thomas Moyer, Kevin Butler, *Member*, *IEEE*, Joshua Schiffman, *Student Member*, *IEEE*,
Patrick McDaniel, *Senior Member*, *IEEE*, and Trent Jaeger, *Member*, *IEEE*

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 to a large number of clients 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 nearly 8,000 static or over 6,500 dynamic integrity-measured web objects per second. More broadly, we identify how TPM-based content web services can scale to large client loads with manageable overheads and deliver integrity-measured content with manageable overhead.

Index Terms—Trusted computing, integrity measurement, web system, scalable attestation.

◆

1 INTRODUCTION

The web has changed the way 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 certificates [1]. The certificate indicates that the server (really the private key) has been vouched for by an authority, e.g., Verisign.

What is missing is a mechanism that offers security guarantees on the content itself. Approaches like per-document XML signatures [2] 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 1) the origin of content and 2) that the origin was functioning properly when the content was generated and delivered. This latter requirement asks for proof of the server *integrity state* at the time of use, and also proofs from other systems involved in generating output, e.g., a database.

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 [3], 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 "in-flight" page changes [4], 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.

In their seminal paper on integrity measurement systems, Marchesini et al. speak directly to the requirements of building and deploying secure web systems [5]. They state, "[t]he promise of responsibly maintaining a secure site requires that the executable suite, considered as a whole, be dynamic." Here they highlight the need for more than simple boot time integrity (such as that provided by stored-sealed configurations and systems), but mandate the integrity measurement must be ongoing. They further expand to state any system providing secure content must provide a binding between this evolving system state and the content being served.

The Trusted Platform Module (TPM) [6] 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 [7], is needed

---

- *T. Moyer is with MIT Lincoln Laboratory, 244 Wood St., Lexington, MA 02176. E-mail: tmmoyer@cse.psu.edu.*
- *K. Butler is with the Department of Computer and Information Science, University of Oregon, 245 Deschutes Hall, 1202 University of Oregon, Eugene, OR 97403-1212. E-mail: butler@cs.uoregon.edu.*
- *J. Schiffman, P. McDaniel, and T. Jaeger are with the Computer Science and Engineering Department, Pennsylvania State University, IST Building, University Park, PA 16802.*
E-mail: {jschiffm, mcdaniel, tjaeger}@cse.psu.edu.

Manuscript received 11 Feb. 2010; revised 26 July 2010; accepted 15 Feb. 2011; published online 17 Mar. 11.
Recommended for acceptance by M. Eltoweissy.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-02-0104.
Digital Object Identifier no. 10.1109/TC.2011.60.

0018-9340/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

to create full attestations of the running system state. The mechanism used by the TPM to provide integrity state is the *quote* operation [6]. 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 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[2] 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, a Firefox extension) 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, which generates a fresh proof for each client request, would not work well in practice. The cost of performing a TPM quote *per request* is extraordinarily high—on the order of 900 msec—limiting the throughput of the web server dramatically. 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 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 with manageable overhead, where the throughput of an integrity measured web server reaches nearly 8,000 web objects per second—within 17 percent of an unmodified Apache server's throughput. Moreover, we show empirically that the same content can be delivered with as little as 2.7 msec 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 from a database can sustain over 6,500 web objects per second.

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 [8], the security is predicated on a certificate that vouches for the authenticity of the web server. The guarantees are linked to the machine rather than to the content, thus leaving no method of knowing whether the content has been manipulated, e.g., by a rootkit or corrupt update.

Providing guarantees on a system's state requires *measurement* of the system's integrity. Many efforts for ensuring system integrity exist, including Pioneer [9], CASS Security Kernels [10], TrustedBox [11], Copilot [12], and LKIM [13] among others. Secure processors such as AEGIS [14] and the IBM 4758 [15] provide a secure execution environment that can be used as a basis for deploying secure services. What these systems and hardware lack is a clear binding to the content they host. As an example, we examine integrity management using the Linux Integrity Measurement Architecture (IMA) [7], and its extension the Policy Reduced Integrity Measurement Architecture (PRIMA) [16], 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 is 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 requester 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 verify the integrity of the message using AIK_{pub}, and then 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 [17]. It contains a small web server that modifies Apache's output, by injecting malicious JavaScript, before it is transmitted to the victim. Under IMA, the binary would be

---

[1] Not just a web service, not just a security service, but a mesh of the two.

[2] In this work, we address the scalability in terms of number of requests processed per second.

added to the measurement list when it was loaded, and this new binary measurement would not be in the list of known-good hashes. Similarly, if a malicious 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 [4], 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 [18]. 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 [19] or advertisement removal by a proxy [20], and also types of malware that modified pages viewed by the user, such as the Adware.LinkMaker [21] which creates links in the page that the publisher did not include, or W32.Arpiframe [22], which injects content into HTTP streams on a local subnet.

2.1 Protecting Content Integrity

Several proposals have looked at providing content integrity "proofs" for web systems. Systems, such as SINE [23] and DSSA [24] provide integrity guarantees to the end-user. SINE is a system that provides integrity for web documents, while still allowing the use of caching servers on the network. DSSA is a server-side system that protects the end-user from maliciously changed documents, by checking server responses against a set of known good responses. If the response is found to be incorrect, the client receives either a backup copy, or is informed that the content is not currently available. These two systems tell the client nothing about the state of the system hosting the content. The client is still required to trust that the server is not compromised with no basis for establishing this trust.

Other systems have leveraged advances in trusted hardware to provide integrity guarantees for the system as well as the content. Two such proposals include WebALPS [25], [26], and [27]. WebALPS utilizes the IBM 4758 [15], a secure co-processor developed by IBM to protect the integrity of client-server interactions when the server accesses sensitive user data. In [27], the authors propose a trusted reference monitor, TRM, that protects the authenticity and integrity of peer-to-peer, P2P, systems. The TRM depends on all systems to have a trusted platform module, TPM, and run secure kernels such as Microsoft's NGSCB [28]. Such proposals have seen relatively little adoption, due to the requirements for expensive hardware, or requiring clients to install new operating systems.

3 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 additional semantics. In our system, by contrast, we *directly bind 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 in past proposals.

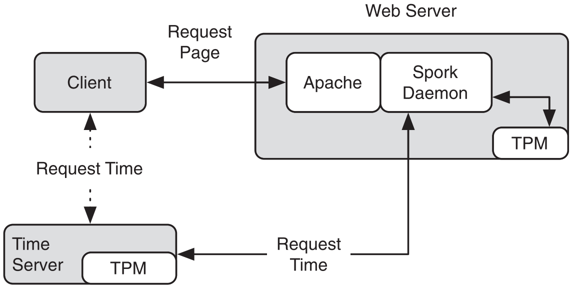


Fig. 1. An overview of the system architecture for asynchronous attested content. The time server provides an attested timestamp to the web server, which uses this to provide integrity-measured content to the clients. The web browser can directly verify the current time from the time server.

3.1 System Overview

An overview of the basic system architecture is shown in Fig. 1. The core elements of the system are 1) a web server that generates static or dynamic web content and provides clients with content integrity proofs, 2) 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 3) a web browser to which we have added an extension that verifies the proofs received from the web server and can securely query the time server 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 TPM quotes from the time server and database 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.

3.2 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 1) that a particular page was served by a given web server, 2) that the web server and supporting backend systems had a verifiable integrity state (which can be assessed for validity), and 3) 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.

$$\underbrace{\underbrace{Quote(H_w, pcr_{H_w}, h(h(p_i)}\,|\,\underbrace{Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))))}}_{\text{web server quote (content proof + time server quote)}}\,|\underbrace{Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))}_{\text{time server quote}}\,|\,\underbrace{t_i}_{\text{time}}$$

Fig. 2. A content proof construction that ties content to both the originating host and the time.

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.[3] 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.

Consider a simple content proof to be received by a client from a 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 1 (web server identity) and 2 (integrity state) from above are provided. What is missing from the simple proof is 3 (the element of time), and any statement about the other backend systems that assisted in the content's generation. 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 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*, providing a self-certified timestamp (that uses the time itself as the quote challenge). The time server is trusted to provide the correct time (by definition of a root of trust [30]), and its quote mechanism is a means of tying a specific timestamp to that service. We revisit design and security issues of the time service in Section 4.2.

During the validation process, the client acquires a timestamp from the time server directly (or uses a suitably fresh timestamp from its cache). 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 [31]. Thus, we have provided a proof whose semantics provide all of the required properties.

The central limitation of the 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 round-trip time (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 throughput would be* ≈1 *request per second.* What is a needed is a means to amortize quote costs.

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 than the proof system as a whole. Thus, authentication costs are amortized over collections of objects. While more sophisticated techniques exist [29], [32], we concentrate on a conceptually simple proof system based on Merkle hash trees [33]. 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 Fig. 3, 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 Fig. 3 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 proof recipient can then validate the content by hashing the file and computing the p_3 leaf and interior nodes on the path to the root. 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 *and* S *are the sizes of the hash and signature, respectively.*

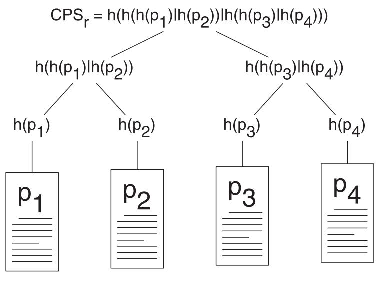


Fig. 3. A Merkle hash tree base for the cryptographic proof system. The leaf nodes are hashes of the pages served to clients.

---

[3] 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.

$$\underbrace{\underbrace{Quote(H_w, pcr_{H_w}, h(CPS_r}\,|\,\underbrace{Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))))}}_{\text{web server quote (content proof + time server quote)}}\,|\,\underbrace{CPS_r}_{\text{proof sys. root}}\,|\,\underbrace{Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))}_{\text{time server quote}}\,|\underbrace{Pf(p_i)}_{\text{page proof}}|\,\underbrace{t_i}_{\text{time}}$$

Fig. 4. Extended content proof that uses a cryptographic proof system as the challenge rather than a document hash. A succinct page proof is also included.

The proof system used to generate an extended content proof for page p_i is shown in Fig. 4. 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.

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 [34]. 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 time to live, TTL, policies.

3.3 Proof Scheduling

Content proofs are delivered to browsers through *integrity proof pages*. The web server inserts an extension `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, or perform other actions deemed appropriate. We discuss the design and operation of the Firefox extension in Section 6.

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 Fig. 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 TPM can generate quotes, 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 Fig. 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.

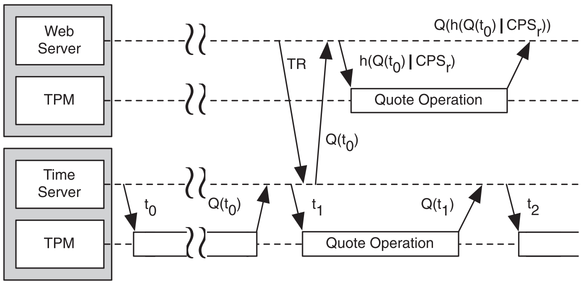


Fig. 5. Server quote generation—The server requests the most recent timestamp from the time server ($Q(t_0)$), and then generates a quote using the most recent hash tree computed (CPS_r).

Dynamic content presents other challenges. Centrally, the page content only becomes available after the request arrives from a client. For example, consider a `.php` [35] web page. PHP allows the web designer to create content programmatically. The inputs to this process include referrer page, URL, query strings, 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 Fig. 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, as in normal operation. *However, the proof identified in the* `X-Attest-URL` *header identifies a proof that does not yet exist.* The generated content is hashed and this hash is added to a cache. This cache contains the hashes of content that was generated between the last TPM quote operation and the pending quote operation. 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 TPM. The cache containing the hashes of dynamic content is then cleared and the process repeats. The proof system is available as soon as the quote operation completes.

The browser will observe additional latency when receiving dynamic content. Assuming a 900 msec quote

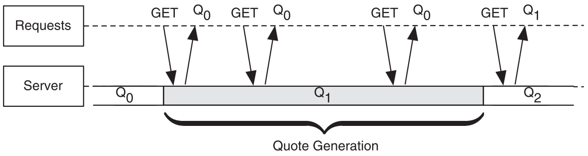


Fig. 6. Static page scheduling—For static pages, the server provides the most recently generated quote (Q_0) to all incoming requests while it is generating the next quote. Once the next quote is generated (Q_1), this new quote is provided to each incoming request.

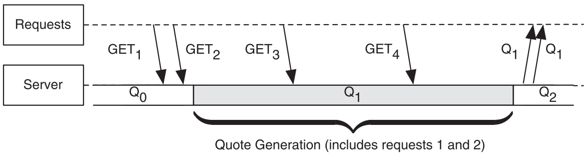


Fig. 7. Dynamic page scheduling—Incoming requests for an *integrity proof page* are delayed until the quote including the page is ready. At this point, a hash tree is generated that includes the cached requests (GET_1 and GET_2) and the hash tree is used to generate the next quote (Q_1).

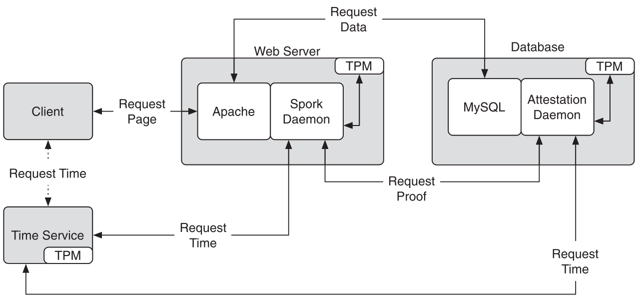


Fig. 8. Adding a database to the system architecture. By having additional backend systems, the client must verify that all systems are high integrity, not just the database.

operation (which is the case in our test environment[4]) and uniform distribution of arrivals, the expected latency would be about 1,350 msec plus the time to deliver the quote itself (which is network dependent). More specifically, the expected arrival in the previous quote epoch is $0.5 * 900 = 450\ \mathrm{msec}$ 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. As the quote operation time is reduced, the latency is also reduced.

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.

3.4 Supporting Backend Systems

In a typical web system, a database provides backend storage for the web application. If the web server uses data from a database to generate content for a client, the integrity of the generated content depends on the integrity of both the web server and the database, as shown in Fig. 8. As such the proof must cover both the web server and the database. One property still missing from the proof in Fig. 4 is the proofs from any supporting systems, such as a database. The client has no means of verifying the integrity of backend systems used to generate the dynamic content. To provide the client with the necessary information to verify the integrity of the backend systems, the web server contacts the database server that provided data to generate the response and retrieves a proof of the database host's *integrity state*. The database proof is constructed as shown in Fig. 9. This quote is bound the time using a time server, much like the web server's proof. This allows the database to serve many servers with a single proof, instead of generating fresh quotes for each server that requests a proof. The database hashes the quote from the time server and uses this as the challenge for its own quote, binding the quote to a recent timestamp. Note that no database content is included in this quote: it simply attests to the current integrity state of the database server.

The web server includes the database attestation in the proof that is sent to the client. To do so, the web server binds the proof to its own quote in much the same way as the time server proof is included. The web server hashes the concatenation of the time server and database quotes and the root of the cryptographic proof system. This becomes the challenge to the web server's TPM. The database quote is also appended to the attestation returned to the client by the web server. Fig. 10 shows the full attestation that the client receives.

4 IMPLEMENTATION

We have developed a version of the architecture detailed in the preceding sections that supports static, dynamic, and mixed content. Fig. 11 shows the structure of the basic Spork web environment. In addition to external clients and the time service, there are two functional elements processing the requests on the web host: the web server and *Spork daemon*.

4.1 Proof-Generating Web Server

As directed by the requested URL, the Apache web server supporting Spork directs all client requests (1 in Fig. 11) 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` header of the retrieved page, and the result is returned to the client (6 in Fig. 11). Dynamic requests occur in substantially the same way except that the content is generated using the appropriate content generation code, e.g., ASP [36], instead of 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 in Fig. 11) (i.e., sockets). The processing thread then sleeps waiting for a "proof ready" event. When the requested proof (5 in Fig. 11) is received by the master thread from the Spork daemon (see below), it wakes the processing thread, which then returns the proof to the client (6 in Fig. 11).

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 in Fig. 11). The remaining threads update the internal state from which the proof systems are constructed. A TPM thread schedules and executes quote operations (4 in Fig. 11) as governed by the algorithms defined in Section 3.3, and separate threads similarly retrieve time attestations (3 in Fig. 11) and database attestations (12 in Fig. 12), if a database is present. Separate threads maintain the dictionary of static documents (by monitoring the filesystem) and the current set of dynamic pages awaiting proof generation.

Client browsers receive the content proof from the web server (6 in Fig. 11) and acquire time attestations from the time server (7 in Fig. 11). 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

---

[4] TPMs from different manufacturers have different optimizations, leading to different times of operation.

$$\underbrace{Quote(H_{DB}, pcr_{H_{DB}}, h(Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))))}_{\text{database quote}} \,|\, \underbrace{Quote(H_{TS}, pcr_{H_{TS}}, h(t_i))}_{\text{time server quote}} \,|\, \underbrace{t_i}_{\text{time}}$$

Fig. 9. A database proof construction that is bound to a recent timestamp.

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 6.

4.2 Time Server

The time service uses a hash of the current hardware timestamp as a challenge to the TPM (8 in Fig. 11). 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 replay attacks.

The time server plays a critical role in operation of the system, because of the importance of freshness to verifying attestations, i.e., the client can be sure that a compromised server will be detected within a short window, since the current time can be validated without relying on the web server. 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 [30]. 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.

$$\underbrace{Quote(H_w, pcr_{H_w}), h(\underbrace{CPS_r}\,|\underbrace{Q_{TS}}|\underbrace{Q_{DB}}))}_{\substack{\text{web server quote (content proof + time}\\ \text{server quote + database quote)}}} \,|\, \underbrace{CPS_r}_{\substack{\text{proof}\\ \text{sys. root}}} \,|\, \underbrace{Q_{TS}}_{\substack{\text{time}\\ \text{server}\\ \text{quote}}} \,|\, \underbrace{Q_{DB}}_{\substack{\text{database}\\ \text{quote}}} \,|\, \underbrace{Pf(p_i)}_{\substack{\text{page}\\ \text{proof}}}$$

where:

$Q_{TS} = Quote(H_{TS}, pcr_{H_{TS}}, h(t_i)) \mid t_i$

$Q_{DB} = Quote(H_{DB}, pcr_{H_{DB}}, h(Quote(H_{TS}, pcr_{H_{TS}}, h(t_i)))) \mid Q_{TS}$

Fig. 10. Full content proof sent to the client, including quotes from the database and time server, and the succinct page proof.

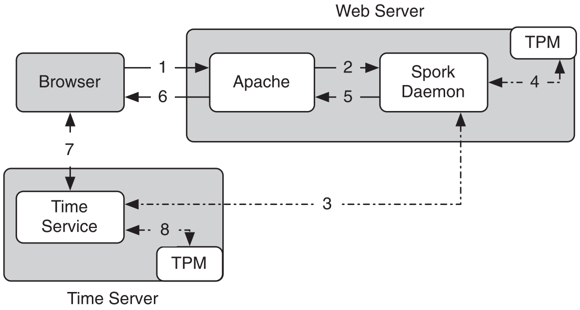


Fig. 11. An overview of the Spork system architecture—The time server provides an attested timestamp to the web server, which is bound to the content delivered to the browser and local software integrity information.

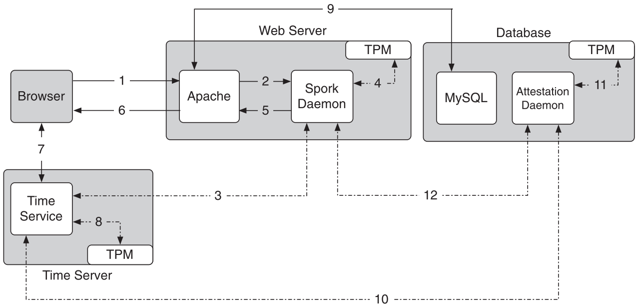


Fig. 12. The extended Spork system that includes a database serving data to the web server. The database resides on a separate system, which is common in web application development.

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.

4.3 Database

Most web applications rely on a backend database that is often hosted on a different server, for performance and security reasons. In order to account for this structure, we have augmented a database server with the ability to provide proofs of the database's integrity state. Fig. 12 shows the extended system structure that includes the database system. The database hosts data that is retrieved by the web server and used to generate the final output for the client (9 in Fig. 12). The database has a daemon that generates periodic attestations using the database system's TPM. The database daemon fetches a recent time quote from the time service (10 in Fig. 12) and hashes this as the challenge for the database system's TPM (11 in Fig. 12). This quote is cached to service later requests (12 in Fig. 12).

5 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.[5] We propose a number of optimizations and evaluate the performance impact.

All tests were performed on Dell PowerEdge M605 blades with 8-core 2.3 GHz Dual Quad-core AMD Opteron processors, 16 GB RAM, and 2x73 GB 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

---

[5] Here we use a single client, and not the full client load determined below.

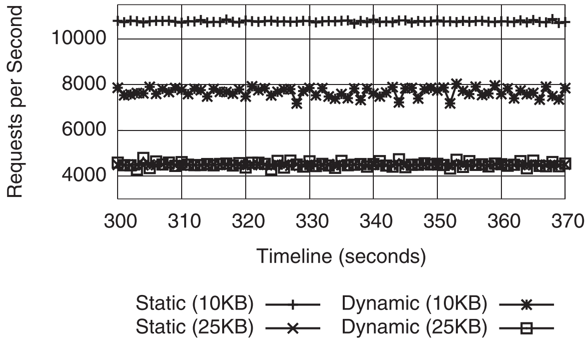


Fig. 13. Unaltered web server throughput—Sustained requests per second (RPS) during a 70 second experiment.

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 5,000 lines of code. All load tests were performed using the Apache JMeter benchmarking tool.

A recent study of web pages indicated that the average web page size is about 130 KB total, with an average HTML source size of 25 KB and the average non-flash object being just under 10 KB [37]. More focused studies of popular websites indicate somewhat larger total sizes (≈300KB) [38]. 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 10 KB and 25 KB file sizes in all experiments.

An analysis of the test environment showed that the maximum throughput of an unaltered Apache web server can be reached with a relatively small number of clients (on the order of 200-300) for static content. In dynamic experiments, client requests are delayed a random period (up to two times the TPM quote period, 1,900 msec) before requesting another page. This ensures uniform arrival of requests at the server,[6] but necessitates significantly more clients to sustain maximal throughput. After experimenting with a number of different client community sizes, we found the highest throughput could be achieved in static experiments with 500 clients and dynamic experiments with 8,000 clients without incurring significant latencies. Thus we use 500 clients to drive all static tests and 8,000 for all dynamic tests, in order to measure the maximum achievable throughput of each system.

5.1 Benchmarks

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 under heavy client loads. 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 each for the content and the proof.

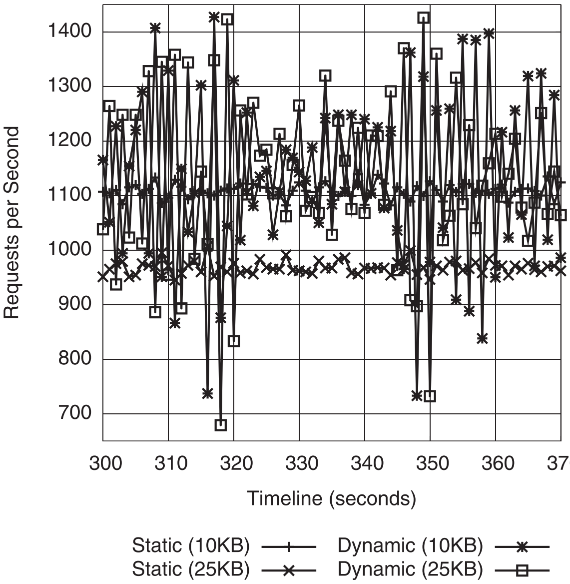


Fig. 14. Integrity measured web server throughput—Sustained requests per second (RPS) during a 70 second experiment.

Fig. 13 shows throughput of an unaltered web server measured in *requests per second* (RPS). The throughput of the 10 KB static content (average 10,770 RPS) has about 29 percent higher throughput than the dynamic case (average 7,600 RPS) for 10 KB 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 25 KB pages for dynamic content are 4,486 and 4,508 RPS for static and dynamic content, respectively. The throughputs are similar because the network is fully utilized.

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. The number of bytes sent per second by the web server serving static content of both the 10 KB and 25 KB pages is essentially the same:

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

where five percent 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 10 KB web pages). This indicates that the bottleneck in the static case is bandwidth. For dynamic content, the performance does not change drastically from when varying the file size until the network becomes saturated. This indicates that dynamic content service is bound by computation, not by bandwidth.

Illustrated in Fig. 14, the average throughput of the integrity-measured web server hovers around 1,000 RPS, a significant drop in throughput compared to the unaltered

---

[6] 6. 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.

TABLE 1
Static and Dynamic System Measurements

	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	4485.5	0.50	7666.3	4.9	4507.8	5.4
IMA	1108.6	3.1	968.1	3.1	1131.5	976.2	1130.7	1058.5
PRIMA	1232.6	2.9	1062.0	3.0	1123.1	1004.2	1120.8	901.0
Compressed IMA	1504.9	2.6	1510.3	2.7	1124.2	969.2	1145.8	1020.7
Compressed PRIMA	1557.7	2.6	1526.8	2.7	1117.3	1054.2	1147.2	939.8

Latencies are measured in milliseconds. The various forms of integrity measurement used are discussed in Section 2. Uncompressed and compressed versions of each system are measured.

server. 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, discussed further in Sections 5.2 and 5.3.

Integrity-measured dynamic content shows an average throughput of 1,100 RPS in both the 10 KB and 25 KB cases, similar to the nonintegrity measured dynamic content where computation, not bandwidth, is the bottleneck. Integrity-measured dynamic content is bounded by the computation of both the content and the proof. The integrity-measured dynamic content also exhibits bursty behavior attributable to the synchronizing effect of the TPM. Clients make a request for dynamic content followed by a request for the corresponding proof and are forced to wait while the TPM generates the quote that includes their page. Once this quote is generated, clients begin the process again by requesting more content.

Table 1 shows minimum observed latency and average throughput. To compute 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 round-trip-times, 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 3.3), about 1,000 msec. These 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 quote time.

Table 2 shows latency measurements for 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 3.2), and are not on the critical path of any content delivery. Nearly 99 percent of the latency involves the acquisition of the time quote and the local quote operation.[7] 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.

5.2 Bandwidth Optimizations

Because we cannot modify the pages directly, we limit bandwidth use by reducing the size of the returned proofs. Recall that 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 and S are the sizes of the hash and signature, respectively. However, the full 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) [16] 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 integrity hashes included in a quote.[8] 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 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 82 KB. 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 1, the throughput of the web server improves under these bandwidth optimizations. Compression of static content clearly improved throughput. Simply compressing the proofs results in 10-57 percent increased throughput, with compressed PRIMA proofs seeing a

---

[7] 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).

[8] Additional information about the XML structure and PRIMA can be found in the Appendices of [39].

TABLE 2
Proof Creation Latency Measurements—Latency of Proof System Generation Measured in Milliseconds

	Static	Dynamic
Generate Merkle Hash Tree	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%)

For the static content, a pool of 125 files was used.

57 percent increase. These optimizations had negligible effect on throughput of 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 85 percent overhead for 10 KB page and 65 percent in 25 KB pages. This is largely due to every integrity-measured static page 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.

5.3 Proof Amortization

Recall that prior studies of web pages show that an average page has one root HTML page and just over 10 static 10 KB embedded objects. As a matter of practice, a client requesting that page will obtain the root 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 root document and all embedded objects at once. Thus, we can amortize 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 $\mathcal{P}$ can be computed in pages as:

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

where μ is the service time for a web server serving a 10 KB static object and ϵ is the service time for the web server serving static (dynamic) 25 KB HTML files. The model assumes that the unit "cost" per object on a hypothetical throughout budget is fixed and independent of other documents. For dynamic content, this model also assumes that the root page is generated dynamically, but that the supporting, embedded objects are static. In cases where more than than just the root page is generated dynamically, the browser would request each piece of content as before, and then request a single proof covering all of the content sent. There is the potential that the dynamic requests are separated into multiple TPM quote windows, and large proofs would be required to cover each batch of dynamic requests. Exploring different mixes of static and dynamic content is an area of future work.

Table 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 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 measured content can achieve web object throughputs within 13 percent of the maximum web server.

5.4 Adding a Second System

In order to understand the impact that Spork has on systems providing backend support, i.e., the database, we added a second system to our experimental setup. This second system hosts a MySQL database that provides data for the web server. Below, we examine the impact that this change has on the throughput of the system.

For these experiments, we use the four client machines described earlier in Section 5 with the same number of dynamic clients (8,000). As before, `mod_python` is used to generate and serve the dynamic content, but for these experiments, the web server connects to the database, fetches data from one table in the database, and returns the content to the client. A pool of database connections is used to reduce the impact of opening and closing connections between the database and the web server.

Fig. 15 shows the throughput of the unaltered web system. With 10 KB dynamic pages, the throughput averages 5,887.5 RPS, while the 25 KB pages averages a throughput of 3,549.4 RPS. The reduction in throughput ($\approx$20%) from previous dynamic experiments is due to the additional network overhead of contacting the database system to retrieve data to service requests.

TABLE 3
Proof Amortization Performance—The Expected and Measured Performance of the Amortized Proof Serving

			Expected		Actual	
	μ	ϵ	$\mathcal{P}$	Web Objects	$\mathcal{P}$	Web Objects
Baseline with Static Root Page	10769	4485.5	868.4	9552.5	867.4	9541.5
Baseline with Dynamic Root Page	10769	4507.8	869.2	9561.7	745.9	8204.8
Integ. Measured Static Root (Full IMA)	10769	968.1	509.8	5607.8	494.9	5444.4
Integ. Measured Static Root (Comp. PRIMA)	10769	1526.8	631.5	6946.4	724.3	7967.4
Integ. Measured Dynamic Root (Full IMA)	10769	1130.7	551.6	6067.3	494.4	6438.3
Integ. Measured Dynamic Root (Comp. PRIMA)	10769	1127.2	550.7	6058.1	650.5	7155.1

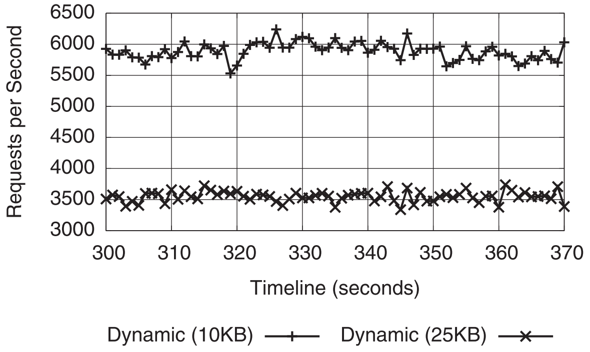


Fig. 15. Unaltered web system throughput—Sustained throughput (requests per second, RPS) during a 70 second experiment. Note that the database is on a different host for this experiment, leading to reduced throughput.

When adding integrity proofs to each request, the throughput of the web system drops dramatically. Fig. 16 shows the throughput of the web system using the uncompressed IMA proof. The throughput averages 430 RPS in both cases, an approximately 90 percent drop in throughput, as compared to the web system serving dynamic content that does not provide integrity proofs. As before, we turn to compression and measurement list reduction (PRIMA) to reduce the size of the integrity proofs, before looking at other means of increasing the overall throughput of the system.

With the addition of a second host, the size of the attestations grows. In addition to the two quotes already included, a third integrity proof is added. The uncompressed IMA proof for the web application is 180 KB. By leveraging PRIMA, the proof is reduced to 148 KB. Compressing the proofs leads to sizes of 54 KB and 44 KB, respectively. With these smaller proofs, Table 4 shows the throughput of the web application. Even with the smallest proofs, the overhead induced by Spork is approximately 90 percent.

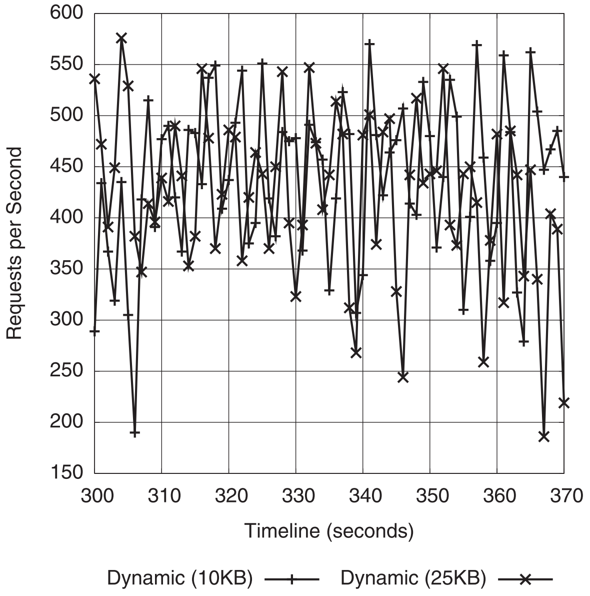


Fig. 16. Integrity measured web server throughput—Sustained RPS during a 70 second experiment, with the database on a separate system, leading to a reduction in throughput compared to the single server experiments.

TABLE 4
Throughput, Measured in Requests Per Second (RPS), with the Database on a Separate Host System

	10 KB Pages (RPS)	25 KB Pages (RPS)
Base	5887.5	3549.4
IMA	439.2	421.2
PRIMA	418.5	450.9
Compressed IMA	544.4	541.6
Compressed PRIMA	549.7	488.8

Uncompressed and compressed versions of each system are measured.

By requesting a single proof that includes proofs for each object on the page instead of "per-object proofs," we can achieve acceptable throughputs. Table 5 shows the estimated and actual throughputs using the simple model from Section 5.3. The unaltered system was able to sustain just under 8,000 RPS with the root page now being constructed with data from the database. The uncompressed IMA proof shows an average throughput of just over 3,600 while the compressed PRIMA proof shows an average throughput of just over 6,500 RPS, an overhead of 17.8 percent.

6 DISCUSSION

6.1 Client-Side Validation

Our Firefox extension validates content proofs 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 simple icons on the status bar of Firefox, similar to Privacy Bird [40], or SSL.

The Firefox interface is modified as shown in Fig. 17. In Fig. 17, 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. One limitation of the current prototype is the fact that the client still obtains the content and can "click-through" the warnings, as is often the case when errors happen [41]. This is not surprising given that $\approx 73\%$ of SSL-protected sites generates a validation warning [42]. The behavior of our prototype is similar to Firefox's default operation of allowing a user to view a page even if the server-side SSL certificate is invalid. When a page is valid, a green check mark is shown instead of a red X. No other prompting is used when the page is valid. One solution to address this limitation is to adopt an approach similar to the ForceHTTPS [43], where validation errors are treated as fatal, preventing the user from obtaining the content.

The prototype 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

TABLE 5
Amortized Proof Performance when the Database is on a Separate Host System

			Expected		Actual	
	μ	ϵ	$\mathcal{P}$	Web Objects	$\mathcal{P}$	Web Objects
Baseline with Dynamic Root Page	10769	3549.4	826.2	9088.4	726.3	7989.1
Integ. Measured Dynamic Root (Full IMA)	10769	421.2	302.8	3330.5	329.7	3627.1
Integ. Measured Dynamic Root (Comp. PRIMA)	10769	488.8	336.2	3698.2	596.7	6563.2

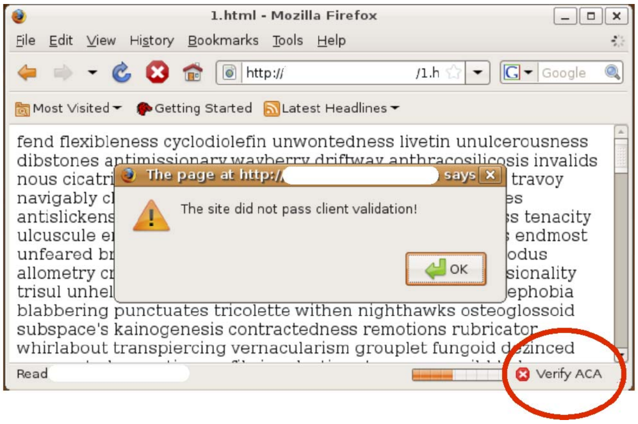
will be bootstrapped with a separate public measurement signing key associated with the services they are measuring. This key would be used to sign measurement lists provided periodically by administrators and possibly provided through the web server as separate URLs. Administrative systems supporting integrity services are being actively studied by the integrity measurement community, and we will make use of these systems as they become available.

6.2 Multi-Tier Architectures

It is common for complex web systems to be deployed on a number of different servers. For example, it is often common to have the web server, or servers, interface with application servers, which in turn will interface with the database servers. Such multi-tier architectures present a challenge for the current design of the Spork system, as each system involved in the content generation process contributes an integrity proof to the proof sent to the client. As the number of systems grows, this can become a management nightmare. We are actively investigating potential solutions that would allow for smaller proof constructions that don't reveal the underlying architecture of the web system to the client. One potential solution would be to introduce a verifier into the web system similar to the verifier proposed in [44], where a single trusted system monitors, and "vouches" for the integrity of the individual systems comprising the overall larger system.

6.3 Virtualization

As Spork requires access to the TPM, deploying Spork-enabled web systems in virtual machines is not currently possible. There is currently on-going to work to provide virtual machines with virtual TPMs, such as work in the Xen project [45], which we plan to leverage once the technology is available. In addition, as a web system grows in complexity, adding more servers, such as a load balancer and more web servers and databases, the size of the attestations sent to the client will grow as well. What is needed is a means of ensuring that the backend systems remain high integrity without requiring the client to verify each and every backend system that contributed to the generation of the client's content. One solution, proposed by Schiffman et al. [46] and [44], allows a VMM to monitor the integrity of the virtual machines deployed on the system and provide short proofs to the client. Integrating this virtual machine verifier (VMV) with Spork reduces the size of the attestations sent to the client, and also allows Spork to be deployed in virtual machine environments. Integrating the VMV and Spork is an area of future work.



fend flexibleness cyclodiolefin unwontedness livetin unulcerousness
dibstones antimissionary wayberry driftway anthracosilicosis invalids
blabbering punctuates tricolette withen nighthawks osteoglossoid
subspace's kainogenesis contractedness remotions rubricator
whirlabout transpiercing vernacularism grouplet fungoid dezinced

The site did not pass client validation!

Fig. 17. Dialog notifying user of an invalid content proof.

7 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/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 communities. Spork delivered almost 8,000 static or 6,500 dynamic integrity-measured objects per second.

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 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 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.

REFERENCES

[1] 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," http://www.ietf.org/rfc/rfc5280.txt, May 2008.

[2] D. Eastlake 3rd, J. Reagle, and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing," http://www.ietf.org/rfc/rfc3275.txt, Mar. 2002.

[3] DarkAngel, "Mood-NT," http://darkangel.antifork.org/codes.htm.

[4] C. Reis, S.D. Gribble, T. Kohno, and N.C. Weaver, "Detecting In-Flight Page Changes with Web Tripwires," *Proc. Conf. Nat'l Spatial Data Infrastructure (NSDI '08)*, pp. 31-44, 2008.

[5] J. Marchesini, S. Smith, O. Wild, and R. MacDonald, "Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love the Bear," Tech. Rep. TR2003-476, Dartmouth College, Hanover, NH, 2003.

[6] Trusted Computing Group, "Trusted Platform Module Specifications," http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications, 2011.

[7] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," *Proc. USENIX Security Symp.*, pp. 223-238, Aug. 2004.

[8] C. Lesniewski-Lass and M.F. Kaashoek, "SSL Splitting: Securely Serving Data from Untrusted Caches," *Computer Networks*, vol. 48, no. 5, pp. 763-779, Aug. 2005.

[9] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP '05)*, pp. 1-16, 2005.

[10] G. Mohay and J. Zellers, "Kernel and Shell Based Applications Integrity Assurance," *Proc. Ann. Computer Security Applications Conf. (ACSAC '97)*, pp. 34-43, Dec. 1997.

[11] P. Iglio, "TrustedBox: A Kernel-Level Integrity Checker," *Proc. Ann. Computer Security Applications Conf. (ACSAC '99)*, pp. 189-198, Dec. 1999.

[12] N.L. Petroni, Jr., T. Fraser, J. Molina, and W.A. Arbaugh, "Copilot-A Coprocessor-Based Kernel Runtime Integrity Monitor," *Proc. USENIX Security Symp.*, p. 13, Aug. 2004.

[13] P.A. Loscocco, P.W. Wilson, J.A. Pendergrass, and C.D. McDonell, "Linux Kernel Integrity Measurement Using Contextural Inspection," *Proc. Second ACM Workshop Scalable Trusted Computing (STC '07)*, pp. 21-29, Nov. 2007.

[14] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architectures for Tamper-Evident and Tamper-Resistant Processing," *Proc. 17th Int'l Conf. Supercomputing*, pp. 160-171, June 2003.

[15] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *Computer*, vol. 34, no. 10, pp. 57-66, 2001.

[16] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: Policy-Reduced Integrity Measurement Architecture," *Proc. ACM Symp. Access Control Models and Technologies, (SACMAT '06)*, June 2006.

[17] cPanel, "Components of Random JavaScript Toolkit Identified," http://blog.cpanel.net/?p=31, Jan. 2008.

[18] "NebuAd," http://www.nebuad.org/, 2010.

[19] A. Fox and E.A. Brewer, "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation," *Proc. Fifth Int'l World Wide Web Conf. Computer Networks and ISDN Systems*, pp. 1445-1456, 1996.

[20] "Ad Muncher: The Ultimate Popup and Advertising Blocker," http://www.admuncher.com/, 2010.

[21] Symantec.com, "Adware.LinkMaker," http://www.symantec.com/security_response/writeup.jsp?docid=2005-030218-4635-99, 2007.

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

[23] C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru, "Sine: Cache-Friendly Integrity for the Web," *Proc. Fifth IEEE Workshop Secure Network Protocols (NPSec '09).* pp. 7-12, 2009.

[24] S. Sedaghat, J. Pieprzyk, and E. Vossough, "On-the-Fly Web Content Integrity Check Boosts Users' Confidence," *Comm. ACM*, vol. 45, no. 11, pp. 33-37, 2002.

[25] S. Jiang, S. Smith, and K. Minami, "Securing Web Servers Against Insider Attack," *Proc. 17th Ann. Computer Security Applications Conf. (ACSAC '01)*, p. 265, 2001.

[26] S. Jiang, "WebALPS Implementation and Performance Analysis: Using Trusted Co-servers to Enhance Privacy and Security of Web Interactions," Master's thesis, Dartmouth College, Hanover, NH, 2001.

[27] X. Zhang, S. Chen, and R. Sandhu, "Enhancing Data Authenticity and Integrity in p2p Systems," *IEEE Internet Computing*, vol. 9, pp. 18-25, 2005.

[28] M. Corporation, "Microsoft Next-Generation Secure Computing Base," http://www.microsoft.com/resources/ngscb/default.mspx, 2010.

[29] M. Noar and K. Nassim, "Certificate Revocation and Certificate Update," *Proc. USENIX Security Symp.*, pp. 217-228, Jan. 1998.

[30] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel, "Establishing and Sustaining System Integrity via Root of Trust Installation," *Proc. Ann. Computer Security Applications Conf. (ACSAC '07)*, pp. 19-29, Dec. 2007.

[31] B.C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks," *Proc. IEEE Communications Conf.*, pp. 33-38, Sept. 1994.

[32] M.T. Goodrich, "Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing," *Proc. 2001 DARPA Information Survivability Conf. and Exposition*, pp. 68-82, 2001.

[33] R. Merkle, "Protocols for Public Key Cryptosystems," *Proc. IEEE Symp. Research in Security and Privacy*, pp. 122-134, Apr. 1980.

[34] "Squid:Optimising Web Delivery," http://www.squid-cache.org, 2010.

[35] "PHP: Hypertext Preprocessor," http://www.php.net, Sept. 2008.

[36] M. Corporation, "Active Server Pages," http://msdn.microsoft.com/en-us/library/aa286483.aspx, 2010.

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

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

[39] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger, "Scalable Asynchronous Web Content Attestation," Tech. Rep. NAS-TR-0095-2008, Network and Security Research Center, Dept. of Computer Science and Eng., Pennslyvania State Univ., University Park, PA, Sept. 2008.

[40] L. Cranor, "Privacy Bird," http://www.privacybird.org/, 2010.

[41] S.E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," *Proc. 2007 IEEE Symp. Security and Privacy (SP '07)*, pp. 51-65, 2007.

[42] Security Space, "Secure Server Survey," http://www.securityspace.com/s_survey/sdata/200906/certca.html, June 2009.

[43] C. Jackson and A. Barth, "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks," *Proc. 17th Int'l Conf. World Wide Web (WWW '08)*, pp. 525-534, 2008.

[44] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding Clouds with Trust Anchors," Tech. Rep. NAS-TR-0127-2010, Network and Security Research Center, Dept. Computer Science and Eng., Pennsylvania State Univ., University Park, PA, Apr. 2010.

[45] S. Berger, R. Cáceres, K.A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vtpm: Virtualizing the Trusted Platform Module," *Proc. 15th Conf. USENIX Security Symp. (USENIX-SS '06)*, vol. 15, 2006.

[46] J. Schiffman, T. Moyer, C. Shal, T. Jaeger, and P. McDaniel, "Justifying Integrity Using a Virtual Machine Verifier," *Proc. 2009 Ann. Computer Security Applications Conf., ACSAC '09*, pp. 83-92, Dec. 2009.

Thomas Moyer received the MS degree in computer science and engineering from Pennsylvania State University, University Park, where he is working toward the PhD degree. He is a researcher in the Systems and Internet Infrastructure Laboratory of Pennsylvania State University Computer Science and Engineering Department. His research interests include web security, systems security, cloud computing security, and large-scale network configuration. He can be contacted at *tmmoyer@cse.psu.edu.*

Kevin Butler received the BSc and MS degrees in electrical engineering from Queen's University, and Columbia University, in 1999, and 2003, respectively. He worked in the Secure Systems Group at AT&T Labs-Research. He is an assistant professor in the Department of Computer and Information Science at the University of Oregon, Eugene. His research focuses on the security of storage, large-scale systems, and networks. He has also examined malware propagation and web systems, and was a member of the EVEREST study of voting machines for the State of Ohio. He also has industrial experience in network operations and research. He is a member of the IEEE. He can be contacted at *butler@cs.uoregon.edu.*

Joshua Schiffman received the MS degree in computer science and engineering from Pennsylvania State University, University Park, where he is working toward the PhD degree, and is the lead graduate student of the Systems and Internet Infrastructure Laboratory, at Computer Science and Engineering Department. His research interests include operating systems and virtual machine security, trusted computing, and cloud computing. He is a student member of the IEEE. He can be contacted at *jschiffman@cse.psu.edu.*

Patrick McDaniel received the PhD degree, in 1996, from the University of Michigan, Ann Arbor. He is an associate professor in the Computer Science and Engineering Department at the Pennsylvania State University and codirector of the Systems and Internet Infrastructure Security Laboratory, Computer Science and Engineering Department of Pennsylvania State University, University Park. Patrick's research efforts centrally focus on network, telecommunications, and systems security, language-based security, and technical public policy. Earlier, he was a software architect and program manager in the telecommunications industry. He is the editor-in-chief of the *ACM Journal Transactions on Internet Technology (TOIT)*, and serves as associate editor of the journals *ACM Transactions on Information and System Security, IEEE Transactions on Software Engineering*, and *IEEE Transactions on Computers*. He has chaired several top conferences in security including, among others, the 2007 and 2008 IEEE Symposium on Security and Privacy and the 2005 USENIX Security Symposium. He is a senior member of the IEEE. He can be contacted at *mcdaniel@ cse.psu.edu.*

Trent Jaeger is an associate professor in the Computer Science and Engineering Department at Pennsylvania State University and the codirector of the Systems and Internet Infrastructure Security (SIIS) Lab, Computer Science and Engineering Department, Pennsylvania State University, University Park. He joined the Penn State after working for IBM Research for nine years in operating systems and system security research groups. His research interests include operating systems security, access control, and source code and policy analysis tools. He has published more than 90 refereed research papers on these subjects. Trent has made a variety of contributions to open source systems security, particularly to the Linux Security Modules framework, the SELinux module and policy development, integrity measurement in Linux, and the Xen security architecture. He is the author of the book "*Operating Systems Security*," which examines the principles and designs of secure operating systems. He is active in the security research community, having been a member of the program committees of all the major security conferences, and the program chair of the ACM CCS Government and Industry Track, as well as chairing several workshops. He is an associate editor with *ACM Transactions on Internet Technology* and has been a guest editor of the *ACM Transactions on Information and System Security. He is a member of the IEEE. He can be contacted at tjaeger@cse.psu.edu.*