Scalable Integrity-Guaranteed AJAX

Thomas Moyer[1], Trent Jaeger[1], and Patrick McDaniel[1]

Pennsylvania State University
{tmmoyer,tjaeger,mcdaniel@cse.psu.edu}

Abstract - Interactive web systems are the *de facto* vehicle for implementing sensitive applications, e.g., personal banking, business workflows. Existing web services provide little protection against compromised servers, leaving users to blindly trust that the system is functioning correctly, without being able to verify this trust. Document integrity systems support stronger guarantees by binding a document to the (non-compromised) integrity state of the machine from whence it was received, at the cost of substantially higher latencies. Such latencies render interactive applications unusable. This paper explores cryptographic constructions and systems designs for providing document integrity in AJAX-style interactive web systems. The *Sporf* systems exploits pre-computation to offset runtime costs to support negligible latencies. We detail the design of an Apache-based server supporting content integrity proofs, and perform a detailed empirical study of realistic web workloads. Our evaluation shows that a software-only solution results in latencies of just over 200 milliseconds on a loaded system. An analytical model reveals that with a nominal hardware investment, the latency can be lowered to just over 81 milliseconds, achieving nearly the same throughput as an unmodified system.

1 Introduction

Sensitive, high-value, information–such as banking, enterprise, and intelligence data–are now commonly being distributed through increasingly complex, interactive web systems. Unfortunately, current web systems are not designed to host high-assurance content. At best, the server-side authentication provided by SSL is of limited use, and as it often built on dubious trust relationships [14] and oft-invalid certificates [39]. More fundamentally, web systems provide no content authentication other than identifying the server from which it was obtained. In this current model, there is no way for a user to determine if the content was corrupted by a compromised web server.

Document integrity systems [16,40,21,22,28] augment content with proofs of the correctness of both the document and the system from whence it was received. Such services allow the consumer of the content to validate not only that the document is authentic, but the content was received from an un-compromised system. This prevents otherwise legitimate but compromised systems from providing mis-information, and preemptively prevents that system from silently manipulating and/or exposing user operation and data. For example, a compromised banking site would be immediately detected by the user when attempting to validate the document integrity of the login screen [34]. The user will simply stop interacting with that site, and therefore no additional damage can be done.

In the Spork project [28], the authors explored the creation of document integrity systems for high-throughput web systems, using the Trusted Platform Module. In order to achieve high throughput, a trade-off for increased latency was made. Such a trade-off poses a challenge for interactive AJAX applications, which require low latency responses to maintain the interactive nature of the web application. What is needed is a document integrity system that supports low-latency responses, to support systems that require low latency, while still sustaining an acceptable throughput.

This paper explores methods and systems designs for providing document integrity in AJAX-style interactive web systems. Chiefly, our *Sporf* system exploits pre-computation to offset runtime costs of providing document integrity. We benchmark a range of *off-line/on-line signature* algorithms and develop new content proof constructions built on them. We detail the design of the Apache-based Sporf server system. A detailed empirical analysis of AJAX applications under

realistic workloads is performed. This analysis shows a software-only system results in latencies of approximately 200 milliseconds, with a throughput of 1,500 requests per second. Further modeling shows that a hardware solution, using nominally priced hardware, results in latencies of just over 81 milliseconds, close to that of an unmodified server. In [31], Nielsen states that web application response times lower than 1 second are optimal. Our software-only prototype system can support response times that are approximately 200 milliseconds, as shown in our evaluation. We begin in the next section by providing an overview of document integrity systems and the cryptographic constructions we explore to support low-latency responses.

2 Related Work

In this section we outline several areas of related work. We begin with a description of off-line/on-line signatures. Next we examine mechanisms to provide proofs of system integrity. Finally, we detail mechanisms that provide integrity for web applications.

2.1 Off-line/on-line Signatures

In many operations involving digital signatures, e.g. electronic wallets and high throughput web systems, the signing operation must be very fast. Typical signature schemes, such as RSA [36] and Rabin [33] are too slow for these types of operations. In [15], the authors propose off-line/on-line signature schemes where the heavyweight computations are performed prior to the content being generated, and then a faster signing operation is carried out once the content is presented for signing. This is done by using both *ordinary* public key signature schemes and *one-time* signature schemes, in a two-phase signing operation. In the off-line phase, a one-time key is generated and signed by the ordinary key. This one-time key is then used to sign a single message, or piece of content, in the on-line phase. This on-line signing phase is significantly faster than the ordinary signatures being generated in the first phase. The full construction for off-line/on-line signature schemes can be found in our technical report [29].

2.2 System Integrity Solutions

Clients communicating with web servers over untrusted HTTP connections are given no guarantees about the security of the server or the network communications. Content accessed over SSL, either directly or via a proxy [24], is afforded some protection from network based attacks. However, SSL does nothing to protect the server, or the content hosted on the server. The client cannot be sure that the server, or the content, is not compromised in some way. The SSL certificate simply vouches for the identity of the server, or more specifically of the private key used by the web server. What is needed is a means of providing a "proof" of the server's integrity.

Several proposals exist for software-based attestation, requiring no hardware changes at all. Such solutions are often targeted at mobile devices, but some have also looked at general purpose systems. Early proposals for software-only solutions include those proposed by Spinellis [43] and Kennel and Jamieson [23]. These projects have looked at ways that software can measure itself, in order to provide proof to a remote party that the code executed has not been tampered with. Other projects, such as SWATT [42] and Pioneer [41] look at performing computations over the code being executed that are difficult to compromise, such as executing code that is highly optimized, or walking memory locations in a pseudo-random order. Other works have looked at ways to increase the robustness of such projects [17], or modifying the kernel or shell to measure code before it is executed [27,19]. *Measurement*, in this context, refers to computing a hash of the executable code right before execution. Popular hash functions include MD5 and SHA1.

Software-only solutions have the advantage of not requiring specialized hardware that could be prohibitively expensive. However, such solutions have shown inherent weaknesses, as attacks have been developed on such solutions [49,7]. As such, hardware based solutions are increasing in popularity. Another reason for this increase in popularity is the decrease in the cost of some of the proposed hardware-based solutions.

Hardware based solutions have been proposed as a means of providing tamperproof storage and execution environments. Projects such as AEGIS [44] and the IBM 4758 [13] provide a secure execution environment. These environments are designed to execute security-sensitive code.

One limitation of using hardware in this manner is the cost of deploying the hardware and software[1]. Copilot [32] is another coprocessor based system that monitors the integrity of the kernel. The Trusted Computing Group [46] has developed a set of specifications, including the Trusted Platform Module, TPM, specification [47]. The TPM, unlike other hardware, is designed to be low-cost. Due to the relatively low cost, many commodity systems are now coming equipped with TPMs. Several projects have examined the use of the TPM as a means of measuring the *integrity state* of the system.

Several proposals have looked at using the TPM to provide system integrity reports [25,38,20]. The Linux Integrity Measurement Architecture [38], Linux-IMA or IMA, and its extension, the Policy Reduced Integrity Measurement Architecture [20], PRIMA, measure code before it is loaded and create a hash chain of all executed code. IMA measures every single executable and library, while PRIMA uses a policy to determine what code should be measured, reducing the overall size of the measurement list. The measurements are stored in the TPM's PCRs, as described above, and a list of all measurements is stored in kernel memory. When an attestation is requested by a remote verifier, the TPM quote is provided, along with the current measurement list. The verifier can examine the measurement list to determine if the expected software is running on the system and that no un-expected software has run that could potentially compromise the system. One such example of un-expected software would be the Random JavaScript Toolkit [12]. This particular piece of malware is a rootkit that modifies Linux-based Apache webservers. The rootkit contains a small webserver that proxies Apache's responses, by injecting malicious JavaScript before sending the response to the client.

2.3 Web Application Integrity

Several proposals exist looking at providing content integrity for web applications. Some systems look to provide guarantees to the client that the content is correct. SINE [16] and DSSA [40] are two systems that aim to provide such guarantees. SINE provides content integrity to the client, while still allowing the client to retrieve content from caches, instead of requesting content from the server every time. DSSA is a server-side solution that monitors the content hosted by the server. The monitor has a set of known-good pages, and any deviations will cause DSSA to either serve a backup of the content to the client, or simply inform the client that the content is currently unavailable. These solutions still require that the user blindly trust the server, providing no basis for establishing trust that the server is not compromised in any way. Another approach is web tripwires [35] that aims to detect "in-flight" pages changes, by comparing the received content to a known good copy. The tripwire concept assumes that the server is not compromised, again potentially misleading the client.

Other works have looked at utilizing trusted hardware to provide integrity guarantees for the system as well as the content being hosted. Two such systems include the WebALPS project [22,21] and [50]. WebALPS uses the IBM 4758, a secure co-processor developed by IBM to protect the integrity of client-server interactions when the server accesses sensitive client information. In [50], the authors propose a trusted reference monitor, TRM, that protects the integrity and authenticity of peer-to-peer, P2P, systems. The TRM depends on all of the systems in the P2P network to have a TPM, and the clients are required to run secure kernels, such as Microsoft's NGSCB [11]. Such proposals have seen relatively little adoption due to the expensive hardware requirements, or requiring the clients to abandon their current operating systems in favor of new systems.

There has been a large effort to provide secure environments for web applications. Such efforts include [10] and [9]. Both are proposals for new web application programming paradigms. Such solutions work well only if the developer is writing their application from scratch, but does not apply well to existing code bases. Hicks, et. al. [18] looked at ways of building web applications that enforce and end-to-end information flow policy. Other efforts have looked at protecting the integrity of content by specifying the canonical form of the content, such as Document Structure Integrity [30], or Blueprint [45], or by relying on type systems provided by programming

[1] The IBM 4758 and successors are very expensive for consumers to purchase and program, and one is required for every system participating in security-sensitive operations.

languages, as in [37]. Another approach is to verify computations done on multiple system, as is done in Ripley [48]. While these solutions protect against popular attacks, such as cross-site scripting (XSS), they all assume that the server hosting and generating content is trustworthy with no means of establishing this trust.

3 General Design

In this section we detail the design of a proof construction that uses off-line/on-line signature schemes [15,8] to sign dynamic content, the mechanism used by Sporf. The advantage to this construction is that it removes the TPM from the critical path of binding dynamic content to the system integrity state. We begin with a discussion of document integrity systems and the guarantees they provide.

3.1 Document Integrity Systems

Document integrity systems provide several guarantees about the content they are hosting and the integrity of the system itself. Such systems provide proofs of the origin of the content, as well as proofs of the current system integrity. The following are guarantees provided by document integrity systems:

a) that a document, d, came from a given server, s
b) that the server has a *known integrity state*
c) that the server was in a *known integrity state* at the time the document was generated

Below, we show how such a system can be constructed, from a set of primitives. We leave the details of specific systems for later discussion. We begin by examining the second guarantee, namely the known integrity state. In a system, s, supporting *system integrity proofs*, a verifier connecting to the system will first validate the integrity of the system. This is done using a challenge-response protocol, where the verifier provides a challenge, or nonce, n, and the remote system generates a *system integrity statement*, denoted:

$$\mathsf{IS}_s(n)$$

Here n is a nonce, or challenge, that ensures the freshness of the generated proof. This proof satisfies guarantees b and c. Next, we show how we can build the document integrity proof for a given document.

That document is represented by d, and the server generates a proof for the specific document by computing a cryptographic hash of the document, written $h(d)$. In order to bind a document to the system integrity statement, we *replace the nonce with the document proof*:

$$\mathsf{IS}_s(h(d))$$

This binds the document to the proof, proving to the verifier that the server stored, or generated, document d when the integrity state was reported. To verify, a client validates the $\mathsf{IS}(\cdot)$ and the hash of the received document, $h(d)$. This construction satisfies a and b, but the client can no longer be sure that the proof is relatively fresh. Specifically, a compromised server can replay such proofs, even after the system has been fully compromised, and the client would be unable to detect an malicious behavior.

To overcome this limitation, the server relies on a trusted time server to provide *verifiable timestamps*, that can be bound to the system integrity statement, in addition to the document proof. Here, | denotes concatenation. The timestamp is written:

$$\mathsf{IS}_{ts}(h(t_i)) \mid t_i$$

Where $\mathsf{IS}(h(t_i))$ is a system integrity proof from the time server, bound to some time, t_i. After obtaining a timestamp from the time server, ts, the server, s, generates the following proof which satisfies the three guarantees outlined above:

$$\mathsf{IS}_s(h(d \mid \mathsf{IS}_{ts}(h(t_i)))) \mid \mathsf{IS}_{ts}(h(t_i)) \mid t_i$$

The hash of the document binds the document to the system integrity proof, the timestamp allows the verifier to determine how fresh the integrity proof is, and the system integrity proof allows the client to validate the integrity of the system. Next, we show how to build a document integrity system that uses commodity trusted hardware and software.

3.2 Example System: Spork

In the Spork project [28], the Trusted Platform Module, TPM [47,46], is used as a means of generating the integrity statements. The TPM provides a limited amount of tamper-evident storage for measurements and cryptographic keys. The measurements are stored in the Platform Configuration Registers, PCRs, and provide a very limited interface for adding values to the set of measurements. The keys stored in the TPM serve a number of different functions, one of which is to sign TPM Quotes. These quotes are the basis for the integrity statements described above. The TPM accepts a nonce and a list of PCRs to report. The TPM reads the selected PCRs and signs the nonce and read values. The quote is written as:

$$\mathsf{Quote}_s(H_s, pcr_s, n)$$

Here, H_s is the key used by the TPM to sign the quote, pcr_s represents the set of reported PCR values and n is the nonce. H_s represents the *identity* of the system, much like an SSL certificate does for secure web transactions. The TPM itself is a passive device, and as such requires support from the system to gather measurements.

Integrity measurement systems gather *measurements* of the current system integrity state. These measurements can later be reported to a verifier to ensure that the system is high integrity. One such system is the Linux Integrity Measurement Architecture [38], which measures executables before they are loaded. A list of these measurements are stored in kernel memory, as well as being reported to the TPM. By including the list of measurements with a TPM quote, a verifier can know what software has been loaded by the system, allowing the verifier to determine if they trust the system, or not. The measurement list is noted as ML_s, where s indicates which system the list is from.

Relying on the TPM leads to very high latencies if each request is signed by the TPM. On average, the TPM takes 900 milliseconds to generate a single quote. In order to amortize this quote generation cost, a Merkle hash tree [26] is used to generate proofs for multiple pieces of content. A client retrieving a proof will get the TPM quote and measurement lists, and a succinct proof from the Merkle hash tree. This allows the Spork system to sign multiple pieces of content with a single TPM quote, and utilize this same quote to service multiple requests. The root of the hash tree, CPS_r, replaces the single document as the document proof, written:

$$\mathsf{Quote}_s(h(CPS_r \mid \mathsf{Quote}_{ts}(h(t_i)))) \mid \mathsf{Quote}_{ts}(h(t_i)) \mid t_i \mid CPS_r$$

By using the hash tree, the cost of generating a TPM quote is amortized over many documents. This amortization works well for static content, where the TPM can generate a single quote for all content that the web server could potentially serve. However, even using the hash tree, Spork introduces a high amount of latency to dynamic requests, as each client must wait for the TPM to sign a hash tree that includes the requested dynamic content. This additional latency cripples AJAX applications, which require low latency responses to maintain the appearance of desktop-like functionality. Next, we examine the design of a system that reduces the latency for dynamic content.

4 Sporf Overview

Next, we describe the Sporf system, where we examine several potential designs for supporting low-latency, high-throughput integrity-assured web documents, detailing the limitations of each approach. We show how to construct the document proof, before showing the full details of the document integrity system using off-line/on-line signatures to sign dynamic content.

4.1 Binding using Off-line/on-line Signatures

First, we will introduce some notation that is used throughout the rest of the paper. Keys are denoted as SK and VK for signing and verification, respectively. Keys for one-time signature schemes are super-scripted with ot, i.e. SK^{ot} and VK^{ot}. σ and π represent signatures.

We begin with the system integrity proof, showing how to bind a single document to the system integrity state:

$$\mathsf{Quote}(H_s, pcr_s, h(h(VK) \mid h(VK^{ot}))) \mid VK^{ot} \mid \sigma \mid \pi \mid ML_s$$

where σ is the signature of the one-time key generated with the many-times key, and π is the signature of the document using the one-time key, SK^{ot}. This construction shows that the server, s, with *known integrity state* (guarantee b from Section 3.1), possessed the one-time key-pair used to sign a *document*, d (guarantee a). What is missing is that the document came from the server at the time when the integrity state was reported (guarantee c).Next, we show how to bind multiple verification keys to a single TPM quote and include a recent timestamp from a trusted time server. We rely on cryptographic proof systems, namely a Merkle hash tree, to bind multiple

$$\underbrace{\underbrace{Q(H_s, pcr_s, h(CPS_r \mid VK}_{} \mid \underbrace{Qts)}_{})}_{\text{server quote}} \mid \underbrace{CPS_r}_{\substack{\text{proof}\\ \text{sys.}\\ \text{root}}} \mid \underbrace{Q_{ts}}_{\substack{\text{time}\\ \text{server}\\ \text{quote}}} \mid \underbrace{ML_s}_{\substack{\text{measure-}\\ \text{ment}\\ \text{list}}} \mid \underbrace{VK^{ot} \mid Pf(VK^{ot}) \mid \sigma \mid \pi}_{\textbf{document proof}}$$

where:
$Q_{ts} = Q(H_{ts}, pcr_{ts}, h(t_i)) \mid t_i \mid ML_{ts}$

Fig. 1. Quote construction using an off-line/on-line key-pair to sign dynamic content. This construction binds the system's integrity state to a recent timestamp, and recently generated one-time keys. The cryptographic proof system includes all of the static content and recently generated one-time keys. Here $Pf(VK^{ot})$ represents a succinct proof from the hash tree.

keys to the TPM quote while being able to generate succinct proofs for each key. The leaves of the hash tree are the individual verification keys used to sign documents, and the root of the tree is used as the challenge for the TPM. When a client obtains a document, it obtains a succinct proof for the verification key in addition to the signatures. Figure 1 shows the full construction, using a cryptographic proof system instead of a single key. The succinct proof is constructed by providing the values of sibling nodes on the path to the root for a given key. Providing this information allows a client to reconstruct the root value from the key it obtains, and compare the computed root to the provided root to ensure that the key is the correct key. This is similar to the method Spork uses to bind multiple documents to a single integrity proof.

4.2 Latency Improvement

The construction in Figure 1 allows the web server to bind a dynamically generated document to the TPM quote, by using the one-time key to sign the dynamic content. This differs from the Spork project which directly binds the content to the proof. In the Spork project, the TPM is on the critical path for serving dynamic content, leading to high latency for each request. With Sporf, the TPM is no longer on the critical path, allowing the system to process requests at much higher speeds, leading to lower latencies for each request.

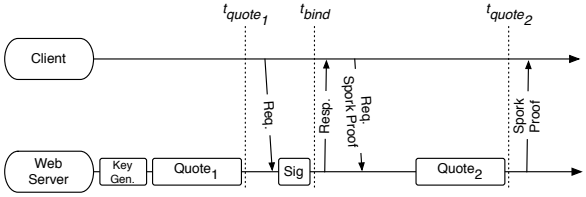


Fig. 2. Timeline for a single request. After the first quote is completed, at time t_{quote_1}, the server is able to sign content. A client makes a request and the server uses a previously generated key to sign the content, time t_{bind} in the figure. At this point, the client has a proof of the server's integrity state up through time t_{quote_1}, and can optimistically begin using the response.

The construction in Figure 1 provides a statement of the system integrity *at the time the keys are generated*, which occurs before the content has been generated. Figure 2 shows the request timeline for a single client requesting content. At time t_{quote_1}, the TPM has generated a quote that can be used to service client requests. When the client requests content, the server generates a signature for the content, using a key included in the quote. This proof, shown in Figure 1, is generated at time t_{bind}, and sent to the client. The proof in Figure 1, is called a *Sporf-integrity proof*. This distinguishes the proof from a *Spork-integrity proof*, where the client gets the proof at time t_{quote_2}, after the server includes the requested content in a TPM quote.

In order to understand the differences between a Spork-integrity proof and a Sporf-integrity proof, we reconsider each of the guarantees outlined in Section 3.1, in terms of the time at which each guarantee is satisfied. The first guarantee (a) that the document d came from the server. This document comes from the server at time t_{bind}. The second guarantee (b) is that the server, s has a verifiable integrity state. This guarantee is satisfied at time t_{quote_i}, when the TPM generates a quote. The third guarantee is satisfied at time t_{quote_2}, when the client can determine the integrity of the system at time t_{bind}.

This is different than the Spork system, where the binding and reporting occur at the same time, i.e. $t_{bind} = t_{quote_2}$, adding additional latency. For Sporf, the proof is delivered at time t_{bind}, eliminating the latency for obtaining content. While delivering the proof at t_{bind} enables clients to obtain content with lower latency, this delivery also presents a *window of uncertainty*, where the server's integrity cannot be determined by the client.

4.3 Window of Uncertainty and Countermeasures

We define the time between t_{quote_1} and t_{bind}, in Figure 2 as a *window of uncertainty*, as the client cannot be certain of the current state of the system up through time t_{bind}, when the content is generated and signed by the server. In order to validate the integrity of the server during this time, the generated content is included in the *next TPM quote* (Quote$_2$ in Figure 2, i.e. a Spork-integrity proof. The proof obtained is the proof described in Section 3.2, obtained by the client at t_{quote_2} in Figure 2. In this section, we describe mechanisms to mitigate the impact of waiting for the Spork-integrity proof.

To mitigate the impact of waiting for the Spork-integrity proof (at time t_{quote_2}), the client can begin using the data after time t_{bind}, i.e. after the Sporf-integrity proof is obtained, and issue a request for the Spork-integrity proof, to arrive after t_{quote_2}. While waiting for the Spork-integrity proof, any content not validated is highlighted in the client's browser to indicate that it is still not fully validated. Any requests resulting from this content are delayed until the Spork-integrity proof arrives. Next, we describe example applications and how this technique operates.

Example Applications Below, we consider two popular applications, and how the developers would integrate Sporf into the application. We first consider the popular Gmail [3] application and also a framework for building AJAX-based instant messaging clients [1]. We show how the typical functions of each application would operate within the Sporf system.

For the first application, consider the web-based email application, Gmail. For this discussion, we will consider what happens when a user receives a message and replies. When the user first logs in, the current contents of the inbox is displayed, and the browser validates the initial requests. The browser will periodically issue AJAX requests to update the inbox and unread message counts for other labels[2]. When a new message arrives, the browser requests the Sporf-integrity proof as part of the AJAX request. After the initial validation, the view of the inbox is updated. The browser then requests the Spork-integrity proof from the server. Until the Spork-integrity proof is received, the message is highlighted to indicate that the content is not fully validated. The client reads the message while waiting for the Spork-integrity proof and begins writing a reply. If the client finishes the reply and clicks the send button before the Spork-integrity proof is validated, the reply is queued until the Spork-integrity proof is validated, and the client is returned to the inbox. If the proof is valid, the reply is sent in the background, otherwise, the client is notified of the failure and the reply is discarded.

[2] A Gmail label corresponds to an IMAP folder, except that a message can have more than one label.

AJAX IM [1] is a framework for building instant messaging clients into web applications. The back-end is a set of PHP scripts, while the front-end is a JavaScript script. Clients send messages to the server, which are then delivered to the other client in real-time. In this case, when the client receives a message from the server, the browser will request the Sporf-integrity proof and highlights the message as only partially validated. The browser requests the Spork-integrity proof after delivering the message. The client can immediately see the message after the Sporf-integrity proof is validated, and can begin writing a response. When the client clicks the send button, the browser first checks that the Spork-integrity proof has been received. If the proof is not received, the message is queued. Once the Spork-integrity proof is validated, the message is sent.

In each of the above examples, the browser is responsible for validating proofs and queueing requests. In future work, we plan to explore the functionality of a Sporf-integrity proof validating browser, and also deploying AJAX applications on Sporf systems.

5 Implementation

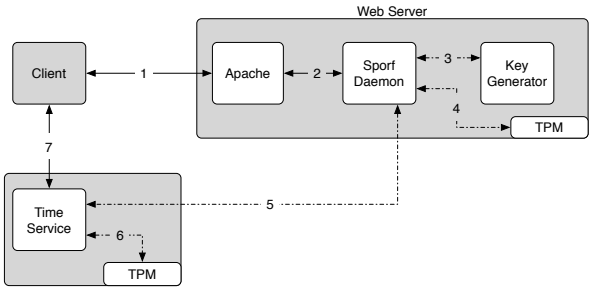


Fig. 3. Detailed system implementation. The numbered arrows represent communications that occur in the Sporf-enabled web server that is serving content integrity proofs. This includes the web server and clients fetching timestamps from the time server, and the daemons generating keys and signing content for the web server.

This section details the implementation of our Sporf system that supports signing dynamic content using off-line/on-line signature schemes. We begin by describing the various systems, and the functions they perform. In addition to the web server, there is a time server that is generating periodic TPM quotes that use a hash of the current hardware clock value as a nonce, previously described in Section 3.2. The web server includes additional daemons to handle generating TPM quotes, generating off-line/on-line signatures, signing dynamic content, and generating document integrity proofs.

The Sporf daemon is responsible for generating proofs and servicing requests from clients for such proofs. In order to support the off-line/on-line schemes presented in the previous section, the daemon is split into several distinct processes. One process, labeled *Key Generator*, generates off-line/on-line keys, and send these to the main *Sporf daemon* (3 in Figure 3). The Sporf daemon handles signing dynamic content generated by Apache (2 in Figure 3), and storing the signatures of previously generated dynamic content. In addition, the main Sporf daemon carries out several other threads of operation.

The main Sporf daemon uses a number of different threads to aid in the content integrity proof generation process. One thread receives off-line/on-line keys from the process generating keys (3 in Figure 3), adding each key to the cryptographic proof system that will be generated in the next TPM quote window. Another thread interfaces with the TPM (4 in Figure 3), to generate the TPM quotes that form the core of the content integrity proofs. This thread also fetches the time attestation from the time server, to be included in the TPM quote generated by the web server (5 in Figure 3). Another thread is responsible for servicing requests for proofs from the web server.

This thread compiles all of the proof pieces, such as the content signature, time attestation, TPM quote, and measurement lists, and sends the generated proof back to the server, which returns the proof to the client.

In our current implementation, the main Sporf daemon is started, and spawns the other processes. The number of spawned key generators is configurable, to take advantage of a varying number of processing cores on the web server. This allows the system to be tuned based on expected workloads and available hardware.

6 Evaluation

In this section, we evaluate the throughput and latency of the Sporf system presented in the preceding sections. We begin our evaluation with a comparison to an unmodified Apache web server. These results lead to several optimizations, which are explored to determine the throughput and latency tradeoffs. In addition to the macro-benchmarks, we perform a series of micro-benchmarks to highlight bottlenecks present in the Sporf system.

All tests were performed on Dell PowerEdge M605 blades with 8-core 2.3GHz Dual Quad-core AMD Opteron processors, 16GB RAM, and 2x73GB SAS Drives (RAID 0). Six blades running Ubuntu 10.04.1 LTS Linux kernel version 2.6.32 were connected over a Gigabit Ethernet switch on a quiescent network. One blade ran the Apache web server, one blade ran the time server, and four were used for simulated clients. All experiments use the Apache 2.2.14 server with `mod_python` 3.3.1 modules for dynamic content generation. The Spork daemon is written in Python 2.6.5 and uses a custom TPM integration library written in C. All load tests were performed using the Apache JMeter [6] benchmarking tool, version 2.4.

6.1 Microbenchmarks

Scheme	Off-line Key Gen	Off-line Thpt.	On-line Thpt.	Verify Per Sec.
GHR-OTS	1772.400	525.210	3289.474	762.195
GHR-DL	2348.200	509.165	50000.000	632.911
GHR-RSA	1982.600	510.204	628.141	634.518
GHR-DL2	2182.400	512.295	5813.953	617.284
GHR-RSA2	2001.200	543.478	586.854	672.043
CS-OTS	2282.200	312.500	3289.474	1336.898
CS-DL	2099.600	305.250	62500.000	925.926
CS-RSA	2126.000	304.507	623.441	968.992
CS-DL2	2191.000	309.023	5813.953	961.538
CS-RSA2	1517.200	326.371	585.480	1086.957

Table 1. Benchmarks of off-line/on-line signature schemes. There are a number of parameters that can be tuned, with varying effects on performance. This table only shows the variations that result in the highest throughput for on-line signing. A complete table of microbenchmarks, for various parameter settings, is included in the Appendix of our technical report [2].

In the first experiment, we evaluate the throughput of the off-line/on-line signature schemes on our experimental test bed. The implementation of the signature schemes was provided by the authors of [8], and were compiled for the machines in our test environment. Table 1 shows throughput measurements for the off-line/on-line selected schemes from [8]. In this table, we consider only parameter combinations that give the highest throughput for on-line signing. A full table is presented in the Appendix of our technical report [2]. In looking at Table 1, we see that the on-line signing phase for some schemes is able to achieve very high throughputs, specifically, GHR-DL and CS-DL achieving over $50,000$ signing operations per second. This indicates that such a scheme would be ideal for signing dynamic content. For our evaluation, we will consider the schemes that achieve the highest throughputs for on-line signing. This includes the following signature schemes from Table 1: GHR-DL, GHR-DL2, CS-DL, and CS-DL2. These schemes provide the highest on-line signing throughput for a single process, and will introduce the least latency when signing dynamic content.

	Content		Proof	
	Thpt.	Lat.	Thpt.	Lat.
Baseline	6134.4	80.8	-	-
GHR-DL	384.2	358.9	381.1	316.1
GHR-DL2	390.7	558.6	387.6	256.2
CS-DL	270.8	984.1	266.8	531.7
CS-DL2	274.5	713.6	270.9	415.1

Table 2. Macrobenchmarks of the four selected off-line/on-line signature schemes. Jmeter was configured to measure the throughput and latency of the content requests and the proof requests. This configuration allows us to view the individual bottlenecks, as well as the overall throughput and latency experienced by each client. Throughput is measured in requests per second and latency in milliseconds.

It should be noted that the schemes labeled GHR-DL and CS-DL provide high throughput in the on-line signing phase. Intuitively, this is due to these signing operations only requiring one integer multiplication operation, unlike other schemes which require more complex operations to complete. Full details are presented in the Appendix of our technical report [2].

6.2 Baseline Macrobenchmarks

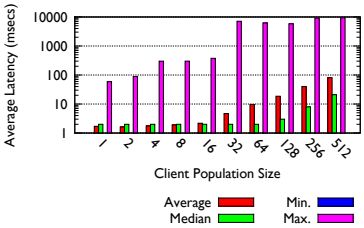
In order to understand the impact of Sporf on serving web documents, we first present a set of macrobenchmarks that examine the throughput and latency characteristics of our Apache web server. In addition to looking at maximum throughput, we examine the latency under several different client populations, ranging from a single client to 512 clients. All tests for maximum throughput will use 512 clients, as adding more clients did not exhibit an increase in throughput, and had an adverse impact on the latency experienced by each client. According to [5], the average size of a page updated via AJAX was approximately 2.5KB, versus 10KB for a full-page refresh. For our experiments, we will use a response size of 2.5KB for each AJAX update.

First, we consider the throughput and latency for an unmodified web server. In our tests, the unmodified Apache web server was able to sustain over 6,100 requests per second, with an average latency of 81 milliseconds per request, as measured by the client. Next, we consider the Sporf-enabled web server. Table 2 shows the throughput and latency average measurements for 512 concurrent clients. The average is taken for a two minute sample, once the system has reached a steady state, i.e. we are ignoring start-up times as these are not indicative of a server's response under load.

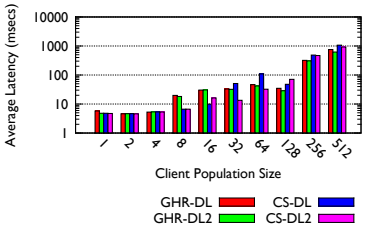
In this experiment, we consider the throughput and latency of the Sporf system. For this test, each client requests an update, followed by the proof for that update. Again, we consider the latency experienced by each client as well as sustained throughput.

Table 2 shows the results using the four schemes selected in the previous section. The columns labeled content and proof measure the throughput and latency of each type of request. When looking at the table, it should be noted that the overall throughput for each of the signature schemes is much lower than the throughput shown in Table 1. This is due to the off-line signing phase. There is a single process generating keys and signing them with the off-line key, i.e. running the off-line phase of the signature algorithm. With a single process doing this, the maximum number of keys generated by the system in a single second maxes out at just over 500 keys per second, thus limiting the throughput of the system, and adding additional latency. In later sections, we will consider methods for alleviating this bottleneck.

In addition to measuring the latency under maximum load, we also measured the latency under varying client loads. This reveals how the server responds to various loads, and what potential bottlenecks exist. We begin with a single client, showing the minimum possible latency, and then consider a maximum of 500 clients. For each client population, clients make serial requests for a period of one minute. All measurements are the average latency during that one minute period, as experienced at the client. Figure 4(a) shows the results for the unmodified web server. We see that the minimum latency experienced is approximately 1.66 msecs, while the average under load is 81.34 msecs. In addition to average latency, the figure shows the median, minimum and maximum latencies for each client population.



(a) Latency for varying client population sizes for an unmodified Apache web server. The client population size is indicated on the x-axis and the latency is measured in milliseconds.



(b) Average latency of both content and proof as a single transaction. In this case, each set of content–proof requests is grouped, and the average total latency is measured.

	10 Keys			25 Keys		
	Content		Proof	Content		Proof
	Thpt.	Lat.	Lat.	Thpt.	Lat.	Lat.
GHR-DL	1511.0	195.9	780.4	1623.0	181.9	588.8
GHR-DL2	1510.3	234.4	787.7	1556.7	188.5	668.8
CS-DL	1476.7	198.9	779.8	1520.1	243.4	617.1
CS-DL2	1466.0	203.3	851.9	1135.4	198.3	669.4

Table 3. By sending multiple verification keys with a single proof, we eliminate the second round trip to obtain a proof for a large number of content requests. This leads to an increase in the content throughput. Throughput is measured in requests per second and latency in milliseconds.

Figure 4(b) shows the average latency under varying client population sizes for a Sporf-enabled web server. Under heavy client loads, we see the latency increase. The lowest increase in latency is with the GHR-DL scheme, showing an increase from 81.34 msecs to 474.6 msecs. This is due to the clients waiting for fresh keys to be generated, as the number of requests being made per second exceeds the number of keys that can be generated in a single second by Sporf. However, this additional 393 milliseconds is lower than the Spork system, which exhibits an average latency increase of over 1000 milliseconds for dynamic content. In the worst case, i.e. CS-DL and CS-DL2, we see an increase in latency to approximately 1,100 milliseconds.

In looking at Table 2, we see that the latency for content requests is below one second on average, with the additional latency coming from the signing of content. Nielsen states that responses under a second allow the client to perceive little delay [31]. In the next section, we examine an optimization that eliminates the second round-trip to fetch the proof, further reducing the overall latency to just the latency experienced for fetching content.

6.3 Pre-fetching Proofs

A naive solution, outlined above, has each client requesting a proof for each piece of content. This causes each client to make two HTTP requests for every AJAX update. As these updates are happening very frequently, very little is changing in the system's integrity state. We can leverage this by providing a proof to each client that includes verification keys for multiple off-line/on-line signature pairs, along with a single TPM quote. The proof is obtained by the client either with the first request made to the server, or when the current pool of keys is exhausted. When clients request content, the server obtains a key that the client already has an integrity proof for, signs the content and appends the signature to the content. Upon receiving the content and signature, the client has everything needed to validate the integrity proof, without making an additional request to the web server. This completely eliminates the second request, and the additional latency introduced by obtaining the proof after fetching the content.

Table 3 shows the effect of proof pre-fetching for each client. We consider the effect of sending both 10 and 25 keys per integrity proof. In order to better understand the impact of this change and exclude the impact of the off-line signing phase, the system generated a large number of keys, which are then stored, and then signed by the TPM in batches, instead of generating keys

in real-time. While this is not how the system will operate in a production deployment, it is useful to understand the potential benefits of Sporf[3]. As shown in the table, it is possible to increase throughput to approximately 1,500 requests per second, while latency for content remains around 200 milliseconds, as compared to approximately 80 milliseconds for serving dynamic content without the content integrity proof. This additional reduction helps to maintain the "sub-one second" goal to maintain the interactivity of the application, as described by Nielsen [31]. The bottleneck in this case is computation, as each client is waiting for a signature of their content.

6.4 Adding Hardware

In supporting large client populations, the system cannot generate keys fast enough to sustain the throughputs of an unmodified web server. In this section, we consider the use of a cryptographic accelerator to support the key generation process. The Silicom PXSC52 [4], which costs just under $500, can sustain a throughput of approximately 17,000 RSA operations per second. Since the off-line phase is based on an RSA signature, if we leverage one of these cards to perform the off-line signing, we can eliminate the bottleneck where clients are waiting for keys to be generated.

To understand the impact adding the accelerator would have on the system, we performed timing tests for the GHR-DL signature scheme. The off-line signing phase can be broken into two steps, with the time for each indicated:

1. Run commitment phase for on-line key (0.292 milliseconds)
2. Sign commitment using many-times key (1.672 milliseconds)

As shown above, the many-times signature operation dominates the off-line signing phase. By moving this signature to the cryptographic accelerator, the off-line signing phase time would drop from 1.964 milliseconds per key to 0.293 milliseconds per key, based on the signature taking 0.001 milliseconds to complete on the accelerator. With this timing, it is possible to generate 3,424 keys per second on a single processor. By adding a second process running the first stage of the off-line signature, we can generate 6,849 keys per second, more than our server's sustained throughput for an un-modified server. This leads to a system where, for a client obtaining an AJAX update, the only latency experienced would be in obtaining a signature for the update, based on the optimizations outlined above. This only adds an additional latency of 0.02 milliseconds per signature, taking the total latency a client can expect down to approximately 81.36 milliseconds, with the server able to sustain the throughputs for an unmodified server, or just over 6,000 requests per second.

7 Conclusion

In this paper, we presented Sporf, a system for generating content integrity proofs for dynamic content. The system provides content integrity proofs for low-latency, high throughput systems, such as AJAX-enabled web applications. We show an in-depth empirical analysis to understand the performance limitations of the prototype Sporf system, and highlight the advantages of leveraging pre-computation. In addition, we explore potential optimizations that allow the system to scale to higher loads without exhausting system resources such as bandwidth. Our results show that a software-only prototype can provide document integrity proofs for dynamic content with approximately 200 milliseconds of latency, with throughputs of 1,500 requests per second. Our analysis shows that a hardware-supported system can provide lower latency, approximately 81 milliseconds, achieving throughputs near that of an unmodified web server.

8 Acknowledgements

The authors would like to thank Dario Catalano, Mario Di Raimondo, Dario Fiore, and Rosario Gennaro for providing access to their implementation of off-line/on-line signature schemes from their paper titled "Off-Line/On-Line Signatures: Theoretical aspects and Experimental Results" [8]. We would also like to thank the members of the SIIS lab for their comments and discussions as this paper evolved, especially Will Enck, who provided the name Sporf.

[3] The off-line/on-line signature implementations are provided by the authors of [8]. We have made no efforts to optimize the implementation of these signature schemes.

References

1. Ajax IM – Instant Messaging Framework. http://ajaxim.com/.
2. Anonymized for submission.
3. Gmail. http://mail.google.com/.
4. PXSC52 - Security Protocol Processor PCI-X Server Adapter / CN1520. http://www.silicom-usa.com/default.asp?contentID=677.
5. Performance Impacts of AJAX Development. http://www.webperformanceinc.com/library/reports/AjaxBandwidth/, October 2010.
6. Apache. JMeter – Apache JMeter. http://jakarta.apache.org/jmeter/.
7. C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, New York, NY, USA, 2009. ACM.
8. D. Catalano, M. Di Raimondo, D. Fiore, and R. Gennaro. Off-line/on-line signatures: theoretical aspects and experimental results. In *PKC'08: Proceedings of the Practice and theory in public key cryptography, 11th international conference on Public key cryptography*, pages 101–120, Berlin, Heidelberg, 2008. Springer-Verlag.
9. S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.
10. B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 269–282, New York, NY, USA, 2009. ACM.
11. M. Corporation. Microsoft Next-Generation Secure Computing Base. http://www.microsoft.com/resources/ngscb/default.mspx.
12. cPanel. Components of Random JavaScript Toolkit Identified. http://blog.cpanel.net/?p=31, Jan. 2008.
13. 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*, 34(10):57–66, 2001.
14. C. Ellison and B. Schneier. Ten risks of pki: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.
15. S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. *Journal of Cryptology*, 9:35–67, 1996. 10.1007/BF02254791.
16. C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru. Sine: Cache-friendly integrity for the web. In *Secure Network Protocols, 2009. NPSec 2009. 5th IEEE Workshop on*, pages 7 –12, 13-13 2009.
17. J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.
18. B. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An Architecture for Enforcing End-to-End Access Control Over Web Applications. In *Proceedings of the 2010 Symposium on Access Control Models and Technologies, SACMAT '10*, 2010.
19. P. Iglio. TrustedBox: A Kernel-Level Integrity Checker. In *Proc. of ACSAC'99*, Washington, DC, Dec. 1999.
20. T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proc. of ACM SACMAT'06*, June 2006.
21. S. Jiang. WebALPS Implementation and Performance Analysis: Using Trusted Co-servers to Enhance Privacy and Security of Web Interactions. Master's thesis, Dartmouth College, 2001.
22. S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, page 265, Washington, DC, USA, 2001. IEEE Computer Society.
23. R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.
24. C. Lesniewski-Lass and M. F. Kaashoek. SSL splitting: securely serving data from untrusted caches. In *Proc. of USENIX Security Symposium*, Washington, DC, Aug. 2003.
25. P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable*

trusted computing, pages 21–29, New York, NY, USA, 2007. ACM.
26. R. Merkle. Protocols for public key cryptosystems. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, Apr. 1980.
27. G. Mohay and J. Zellers. Kernel and Shell Based Applications Integrity Assurance. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC'97)*, San Diego, CA, Dec. 1997.
28. T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. Scalable Web Content Attestation. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.
29. T. Moyer and P. McDaniel. Scalable Integrity-Guaranteed AJAX. Technical Report NAS-TR-0149-2011, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Mar. 2011.
30. Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'09)*, 2009.
31. J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Thousand Oaks, CA, USA, 1999.
32. N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proc. of USENIX Security Symposium*, San Diego, CA, Aug. 2004.
33. M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Report TR-212, Lab. for Computer Science, MIT, 1979.
34. M. A. Raza. A Leading Pakistani Bank's Website Got Compromised. `http://propakistani.pk/2008/12/26/bank-got-hacked-pakistan/`.
35. C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proc. of NSDI'08*, pages 31–44, Berkeley, CA, USA, 2008. USENIX Association.
36. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
37. W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, 2009.
38. R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of USENIX Security Symposium*, San Diego, CA, Aug. 2004.
39. Security Space. Secure Server Survey. `http://www.securityspace.com/s_survey/sdata/200906/certca.html`, June 2009.
40. S. Sedaghat, J. Pieprzyk, and E. Vossough. On-the-fly web content integrity check boosts users' confidence. *Commun. ACM*, 45(11):33–37, 2002.
41. 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. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, Oct. 2005.
42. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. pages 272 – 282, may. 2004.
43. D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Trans. Inf. Syst. Secur.*, 3(1):51–62, 2000.
44. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architectures for Tamper-Evident and Tamper-Resistant Processing. Proc. of the 17th International Conference on Supercomputing, June 2003.
45. M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *30th IEEE Symposium on Security and Privacy*, 2009 2009 2009 2009 2009.
46. Trusted Computing Group. TPM Working Group. `https://www.trustedcomputinggroup.org/groups/tpm/`.
47. Trusted Computing Group. Trusted Platform Module Specifications. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications`.
48. K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, New York, NY, USA, 2009. ACM.
49. G. Wurster, P. C. v. Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138, Washington, DC, USA, 2005. IEEE Computer Society.
50. X. Zhang, S. Chen, and R. Sandhu. Enhancing data authenticity and integrity in p2p systems. *IEEE Internet Computing*, 9:18–25, 2005.