BUILDING SCALABLE DOCUMENT INTEGRITY SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Thomas Moyer

© 2011 Thomas Moyer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2011
The dissertation of Thomas Moyer was reviewed and approved* by the following:

Patrick D. McDaniel  
Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Trent R. Jaeger  
Associate Professor of Computer Science and Engineering

Thomas F. LaPorta  
Professor of Computer Science and Engineering

Constantino Lagoa  
Professor of Electrical Engineering

Mahmut Kandermir  
Professor of Computer Science and Engineering  
Director of Graduate Affairs

*Signatures are on file in the Graduate School.
Abstract

Web systems have become the de-facto method for exchanging information. Currently, users must blindly trust that the systems they are communicating with over a secure connection are high-integrity. Even if the remote system is able to provide proof of its integrity, the client has no easy way to verify that the content being delivered is genuine. This thesis explores the construction of document integrity systems, a mechanism for building high-integrity web systems that clearly shows that content being delivered came from a system with a known-integrity state.

Web applications pose a number of different performance challenges for document integrity systems, and we develop several systems to satisfy different performance requirements. Our first system, Spork, provides high-throughput for at the expense of high latency for dynamic content. In order to address the high latency for dynamic content, we explore new cryptographic constructions that reduce the end-user’s perceived latency, while maintaining the guarantees provided by a document integrity system. Finally, we develop a general framework for building document integrity systems, and explore the use of the framework to build a document integrity system for a provenance-aware database system.
# Table of Contents

List of Figures .................................................. viii
List of Tables .................................................... xii
Acknowledgments ................................................... xiv

Chapter 1
Introduction ....................................................... 1
1.1 Web Transaction Integrity ................................. 3
   1.1.1 Information storage and retrieval .................. 3
   1.1.2 Transmission ........................................... 4
   1.1.3 Rendering .............................................. 4
1.2 Providing Content Integrity ............................... 5
1.3 Thesis Statement ............................................ 5
1.4 Contributions ............................................... 7

Chapter 2
Background ......................................................... 8
2.1 Overview ...................................................... 8
   2.1.1 Generation ............................................. 10
      2.1.1.1 Mashups .......................................... 10
      2.1.1.2 Frameworks ..................................... 10
   2.1.2 Storage ................................................ 11
   2.1.3 Transmission ......................................... 11
   2.1.4 Rendering ............................................. 12
2.2 Attacks ....................................................... 14
   2.2.1 Generation ............................................. 14
   2.2.2 Storage ................................................ 15
   2.2.3 Transmission ......................................... 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4</td>
<td>Rendering</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4.1</td>
<td>Cross-site Scripting</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4.2</td>
<td>Cross-site Request Forgery</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4.3</td>
<td>Other Attacks</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Defenses and Countermeasures</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Generation</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1.1</td>
<td>Static Analysis</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1.2</td>
<td>Dynamic Analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1.3</td>
<td>Hybrid Approaches</td>
<td>22</td>
</tr>
<tr>
<td>2.3.1.4</td>
<td>Mashups</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1.5</td>
<td>Frameworks</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Storage</td>
<td>25</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Transmission</td>
<td>26</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Rendering</td>
<td>26</td>
</tr>
<tr>
<td>2.3.4.1</td>
<td>Cross-site scripting</td>
<td>26</td>
</tr>
<tr>
<td>2.3.4.2</td>
<td>Cross-site Request Forgery</td>
<td>28</td>
</tr>
<tr>
<td>2.3.4.3</td>
<td>New Browsers</td>
<td>28</td>
</tr>
<tr>
<td>2.3.4.4</td>
<td>Modifications to Existing Browsers</td>
<td>30</td>
</tr>
<tr>
<td>2.3.4.5</td>
<td>New Policies</td>
<td>32</td>
</tr>
</tbody>
</table>

**Chapter 3**

**Document Integrity Systems**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Trusted Computing Hardware</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1</td>
<td>IBM 4758</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Integrity Measurement</td>
<td>38</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Software Solutions</td>
<td>38</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Hardware Solutions</td>
<td>39</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Privacy Concerns</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Other Document Integrity Systems</td>
<td>42</td>
</tr>
</tbody>
</table>

**Chapter 4**

**Spork**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>System Overview</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Content Proofs</td>
<td>45</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Proof Scheduling</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Supporting Backend Systems</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Spork Implementation</td>
<td>53</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Proof-Generating Web Server</td>
<td>53</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Time Server</td>
<td>55</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Database</td>
<td>56</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Client-side Validation</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>Bandwidth Optimizations</td>
<td>64</td>
</tr>
</tbody>
</table>
Appendix A
Spork 116
A.1 Attestation XML Example . . . . . . . . . . . . . . . . . . . . . . . . . . 116
A.2 Reducing Attestation Sizes . . . . . . . . . . . . . . . . . . . . . . . . . 117

Appendix B
Sporf 119
B.1 Off-line/on-line Signature Constructions . . . . . . . . . . . . . . . . . . 119
B.1.1 Many-times Signature Schemes . . . . . . . . . . . . . . . . . . . . . . . 119
B.1.1.1 GHR Signature . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
B.1.1.2 CS Signature . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
B.1.2 One-time Signature Schemes . . . . . . . . . . . . . . . . . . . . . . . . 120
B.1.2.1 Lamport Signature . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
B.1.2.2 Chameleon Hash Functions . . . . . . . . . . . . . . . . . . . . . . 121
B.1.2.3 Discrete Log Based Chameleon Hash Function . . . . . . . . 121
B.1.2.4 RSA Based Chameleon Hash Function . . . . . . . . . . . . . 121
B.1.3 Off-line/on-line Signature Schemes . . . . . . . . . . . . . . . . . . . 121
B.1.3.1 Using One-time Signatures . . . . . . . . . . . . . . . . . . . . . . 122
B.1.3.2 Using Chameleon Hash Functions . . . . . . . . . . . . . . . . . 122
B.2 Chameleon Hashing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
B.3 Off-line/on-line Signature Benchmarks . . . . . . . . . . . . . . . . . . . 123

Appendix C
Splayd 126
C.1 Database Schema . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

Bibliography 132
List of Figures

2.1 The phases of a static web request. The processing begins once a request arrives at the server. During the generation phase, the web server will communicate with the static storage mechanisms. This response is then transmitted to the client, where it is rendered in the browser. 9

2.2 The phases of a dynamic web request. The processing begins once a request arrives at the server. During the generation phase, the web server will pass the request to the web application code to generate the response. The web application code will often access static storage to load templates and then populate the templates with data from other storage mechanisms, such as databases. Once the response is generated, it is passed back to the web server, which transmits the response to the client. The client then renders the content in the browser. 9

3.1 During power-on, the chain of trust starts with the BIOS measuring itself and the bootloader. The bootloader then measures the kernel and initial RAM disk (initrd). The kernel is then responsible for measuring the applications. This chain allows a verifier to validate all code loaded by the system. 37

4.1 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. 45

4.2 A content proof construction that ties content to both the originating host and the time. 45

4.3 A Merkle hash tree base for the cryptographic proof system. The leaf nodes are hashes of the pages served to clients. 46

4.4 Extended content proof that uses a cryptographic proof system as the challenge rather than a document hash. A succinct page proof is also included. 48

4.5 Server quote generation - The server requests the most recent timestamp from the time server (Q(t0)), and then generates a quote using the most recent hash tree computed (CPSr). 50
4.6 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.

4.7 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).

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

4.9 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.

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

4.11 An overview of the Spork system communications – 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.

4.12 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.

4.13 Dialog notifying user of an invalid content proof.

4.14 Unaltered web server throughput – sustained requests per second (RPS) during a 70 second experiment.

4.15 Integrity measured web server throughput – sustained requests per second (RPS) during a 70 second experiment.

4.16 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.

4.17 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.

5.1 A document proof construction that binds the document proof to the system integrity state and a recent timestamp.

5.2 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.
5.3 An example of a hash tree with four keys. The circled values are a succinct proof for key $k_1$. The client computes the hash of the provided key and then computes the values up to the root of the tree. This computed value is then compared to the provided value.

5.4 Timeline for a single request. After the first quote is completed, at time $t_{\text{quote}}$, 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 $t_{\text{bind}}$ in the figure. At this point, the client has a proof of the server’s integrity state up through time $t_{\text{quote}}$, and can optimistically begin using the response.

5.5 Detailed system implementation. The numbered arrows represent communications that occur in the Spork-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.

5.6 Distribution of latency measurements for varying client population sizes, including 16 clients, 128 clients, and 512 clients. Here, we see that in each case, the latency for each client is relatively stable.

5.7 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.

5.8 Latency for a varying number of clients using the four different signature schemes to obtain content. This chart reflects only the latency for obtaining content, and not the latency for obtaining the indicated proof.

5.9 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.

5.10 Quote construction using an RSA key to sign dynamic content. This construction binds the system’s integrity state to a recent timestamp and a freshly generated RSA key. The cryptographic proof system root is still included for static content.

5.11 Quote construction using either an off-line/on-line scheme, or a traditional RSA/DSA public-private key-pair to sign dynamic content. This construction binds the system’s integrity state to a recent timestamp, a set of off-line/on-line keys, and an RSA/DSA key-pair. The cryptographic proof system includes all of the static content, the one-time keys generated during the TPM quote window, and all dynamic content that has been generated since the previous TPM quote.

6.1 Splayd framework modular design. The two interface modules are provided by an application developer. The request service module presents an HTTP interface, making integration simple for most systems.
6.2 A system built using the Splayd framework to bind provenance logs to system integrity proofs. The framework handles the communication with the TPM and the time server, while the developer provides the code to fetch provenance logs, i.e. the data to be bound to the proof.

6.3 Sustained throughput for SELECT query with a varying number of records.

6.4 Sustained throughput for INSERT query with a varying number of records.

6.5 Sustained throughput for UPDATE query with a varying number of records.

6.6 Average steady-state throughput for continuous SELECT queries from each of 160 clients over a two-minute experiment. Here, each query is a SELECT to obtain the records from the tables, followed by a second SELECT to obtain the provenance log for the records.

6.7 Average steady-state throughput for continuous SELECT queries from each of 160 clients over a two-minute experiment. Here, each query is the initial SELECT to obtain the data, followed by an HTTP request to obtain the data integrity proof.
List of Tables

4.1 Static and dynamic system measurements. Latencies are measured in milliseconds. The various forms of integrity measurement used are discussed in Section 3.2. Uncompressed and compressed versions of each system are measured. .................................................. 62

4.2 Proof creation latency measurements – latency of proof system generation measured in milliseconds. For the static content, a pool of 125 files was used. .................................................. 63

4.3 Proof Amortization Performance – the expected and measured performance of the amortized proof serving. .................................................. 66

4.4 Throughput, measured in requests per second (RPS), with the database on a separate host system. Uncompressed and compressed versions of each system are measured. .................................................. 68

4.5 Amortized proof performance when the database is on a separate host system. .................................................. 70

5.1 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 Appendix B.3. ......................... 82

5.2 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. ......................... 83

5.3 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. .......... 88

5.4 A benchmark of key-generation, signing and verification for RSA and DSA signature schemes. These measurements were performed using the OpenSSL library. .................................................. 90
A.1 The set of trusted subjects ................................................................. 118

B.1 Off-line/on-line signature benchmarks based on the Gennaro et. al. [1] many-times signature scheme, measuring key generation, off-line signing, on-line signing and verification. The second column of the table shows the settings for various tunable parameters. .................................................. 124

B.2 Off-line/on-line signature benchmarks based on the Cramer-Shoup [2] many-times signature construction, measuring key generation, off-line signing, on-line signing and verification. The second column of the table shows the settings for various tunable parameters. .................................................. 125
Almost six years ago, I sat, much like I am now, staring at a blank computer screen, wondering what to write. Back then, I was trying to define exactly why I wanted to go to grad school. I had grand ideas about becoming a professor and teaching classes in computer science. I talked with many of my professors at the time and got their take on the graduate school experience, and what career opportunities it would open up. The most influential of these conversations occurred with Dr. Heller, my operating systems professor. His discussions about graduate school cemented my decision to attend and get my PhD. Now, on the other side, I look back at all of my experiences and realize that things never work out as planned.

In applying to graduate school, I decided to work in theoretical computer science. It was a topic that always fascinated me as an undergrad, and I figured that I would be most challenged in that area. However, after a year, I realized that I was an engineer at heart. This meant building new things, and also dismantling broken things. This is about as far from theoretical computer science as one can get, so I started to search around for new topics. In one class I was taking, I met a student working in security, William Enck. Will spoke volumes about my advisor and the interesting work they were doing. Without his advice, I would not have ever talked to my advisor and ended up where I am today. In addition, Will has helped me through many tough times in grad school, always willing to listen and give advice, often leading me to deeper understandings of the problems I was focused on.

Shortly after joining the SIS lab, I met Kevin Butler and Steve McLaughlin. Kevin was working on some interesting ideas in storage security that I found intriguing. We worked on several projects together, and he helped me through my first conference presentation, something I thought was going to kill me. He was always willing to listen and provide advice, either with an insightful question to help me find the answer, or a joke to lighten the mood. Steve has been a constant source of deep, interesting discussions on a wide range of topics. Discussions with Steve often cause me to challenge my own beliefs about many different things, and often makes me think more critically about the work I do.

Every group has a comedian, and the lab had Sandra Rueda. Sandra was often dubbed the “social director” as she would plan the outings that kept the lab close. She
was always willing to help make someone laugh, even if she herself was having a rough
day. Without her comedic relief, I would not have survived graduate school, as I often become too serious for my own good.

Before moving into my current desk, I had the privilege of sharing an office with Damien Octeau and Devin Pohly. This office had the distinct benefit of windows to see the outside world. However, these windows soon turned into a double-edged sword as the summer months saw a drastic increase in temperature. While others fretted about a lack of windows to see the sun, we fretted about the lack of cool air. Devin is another person that has taught me to think more critically about things. We often discuss system design and administration, and every time we finish, I felt that I have learned something new, even after so many years of managing systems. While Sandra has provided comic relief to the lab, Damien has been the target of many jokes, always smiling and never getting angry.

Often when I get frustrated, I find it best to talk to someone. For the past year and half, the most common targets for my frustrated rants have been Mike Lin and Hayawardh Vijayakumar. Even when very busy, they have been willing to listen to my rants, until I calm down enough that I stop talking and get back to work. Mike and I took several classes together and without his help, I would never have gotten through Adam Smith’s cryptography class. Hayawardh is another example of patience, often listening to me rant, and never telling me he was too busy to listen, even when I knew that he had many pressing deadlines.

Another person who was always willing to listen is Dave King. Dave was finishing up as I was getting started, but in the short time that our paths crossed, Dave taught me many things. Dave is an early riser, much like myself, and we would often have early morning discussions about a variety of topics. While our research interests never aligned, we did teach a course together during my first year of grad school. His support helped me survive my teaching experience, and my teaching skills are better for having worked with him.

In the almost six years I have been at Penn State as a graduate student, I have met many other students that have helped me in many ways, and I could go on forever listing all of them, but the list would be very long. Without the support of all of these students, I would not be where I am today.

There is one peer in particular that stands out among the others and that is Joshua Schiffman. Josh and I started grad school at the same time, but it was not until my second year that we got to know each other. We began working on a project together when I joined the lab. That project has led to my dissertation and his as well, although focused on very different topics. Josh was always willing to discuss problems I was facing, and would always be able to provide solutions that I myself would not have thought of. He helped me to better understand many concepts that now seem so simple to me. He has also been a constant support in my struggles to survive grad school, and I am lucky to call him a friend.

While the peers I have been fortunate enough to work with have helped shape who
I am today, there is one person that has had a far greater hand in shaping me than any of them, and that is Patrick McDaniel, my advisor. His support, and belief that I could succeed, has been the single greatest factor in my finishing grad school. He has taught me not only how to do research and build systems, but also how to think critically, and “ask the right questions”. We would often have discussions about problems I was having and he would always make me question myself, instead of simply providing me with the answers. This has helped me to understand the depth of my own abilities. I own him more than I can ever possibly hope to repay.

While Patrick has been my greatest supporter in grad school, his wife Megan deserves just as much, if not more, thanks. She often sends in food for the students, and is always willing to have students over. Everytime students invade her home, she has delicious food and a welcoming atmosphere. Without her support, Patrick could not dedicate himself to his students like he does, and for that I am grateful.

I am also grateful to the many friends in undergrad and graduate school that I have not named. They are far too numerous to list, but their support has also shaped the person I am today.

My desire to dismantle things can be directly attributed to my parents. Anytime something would break, it would find its way to my room, where I would be allowed to take it apart and see how it worked. This fascination was never discouraged, and led to my desire to build things. I still remember the first computer that my dad and I built. I was amazed that we could put together something and have it actually work on the first attempt. In doing this, my parents helped me realize my desire to be an engineer. Without their support, I would not be where I am today.

Most recently, the core of my support system has been my wife Sam and my daughter Tori. Without their support, I would never have finished grad school. Sam is the most forgiving person I know, even understanding that I had to go to New Jersey while she was pregnant. Her constant support has allowed me to focus on grad school and her willingness to take care of everything she possbily could, to be a “Supermom”, is a quality that I don’t think I could ever match. My daughter Tori is always excited to see me, and she is always willing to forgive me when I have to go back to work after dinner instead of playing games with her. The two of them make coming home every night the high point of my day. Even when things seem like they are falling apart, I know that they are at home, and will be as excited to see me as I am them. Without their unwavering support, I would not be here today and it is to them that I dedicate this dissertation.
Dedication

To Sam and Tori, without whom, I could not have done any of this.
Chapter 1

Introduction

The web is unarguably one of the most prevalent technologies today. What was once a means of disseminating academic information between universities [3] has become a means for the common person to connect to people both across the street and around the globe. It is estimated that 28.7% of the world’s population is connected to the web, representing an almost 445% increase over the past ten years [4].

This tremendous growth began at CERN with the efforts of Sir Tim Berners-Lee [5] and his colleagues, with the creation of the first web browser supporting the hypertext transfer protocol [6]. Initially, the idea was to create a series of text based pages with hyperlinks [7]. The fundamental goal was to make the sharing and finding of information as simple as clicking a series of links. This has exploded into the web we know today, where a huge variety of content exists, ready to be consumed by the masses. Everything from photos and videos, shared with many people, to sensitive information, such as banking and medical records, that users wish to keep private.

Private information is increasingly becoming available via web interfaces. Common examples include web banking, on-line retailers, and medical records. Users expect that when they provide information to these sites that the provided information will remain confidential, only being released to those parties that the user expects. In many cases, users have been let down, as the sites have been compromised, leading the release of sensitive information. Banks and retail chains are just two common examples of web sites storing sensitive personal information. These sites are expected, by users, to maintain a secure infrastructure, safeguarding the sensitive information entrusted to them. In many cases, these companies have fallen short of this goal.

Despite the large amount of effort expended to secure the computing infrastructure for
companies storing sensitive personal information, in an increasing number of incidents, users are finding their information stolen. Take for example banks running on-line web banking services for their customers. Banks in India and Pakistan have experienced data breaches [8, 9], leading to the compromise of many user accounts. In addition, attackers are using trojans to modify the pages seen by clients [10, 11], often times asking for additional account details, like account numbers. In addition, a survey in 2009 of 113 financial service providers indicated that most felt that large data compromises, such as stealing data from databases was by far more common than phishing attacks [12] targeting users of the services [13]. Banks are not the only targets though, as exhibited by the compromise of retail databases like the TJX data breach [14], the Heartland breach [15], or the attack on a popular on-line retailer [16]. In all cases, sensitive customer information, such as credit card details, were stolen from the databases of the retailer. These attacks on the systems providing web services highlight the fact that clients are placing trust in systems that they know little to nothing about.

Systems run a wide variety of software, from off-the-shelf solutions to custom developed software solutions. Users know nothing about how their data is processed on the server, once submitted. In many cases, attackers have simply looked for vulnerable systems, gaining access and siphoning off sensitive information. These are the cases that see the most press, as they are often followed by a release of the stolen information in a high-profile manner. However, there are more subtle attacks that have been a source of frustration for users and administrators alike.

In some cases, instead of stealing user information from the database, which might be difficult, a much more subtle attack can occur. In the case of the web banking server in Pakistan [9], the server-side web banking software was modified to leak user credentials to the attacker as the user was logging in. This differs dramatically from the typical phishing attack, where users are directed to a site that simply looks like the site being attacked, but is often times hosted on a server controlled by the attacker, and not the actual server hosting the original content. Users are fooled into entering their credentials into the fake page, and after recording these credentials, the server will redirect the user to the legitimate server, allowing users to go about their business. In the subtle attack outlined above, the client will access the server using the same URL, seeing the same user interface, the same valid SSL certificate, and the same behavior from the server, all the while oblivious to the theft of sensitive information.

This highlights a problem with the way users interact with and build trust in the web services they use. Users have been trained to look for certain visual cues that the site
they are accessing is the correct site, such as the near ubiquitous padlock icon indicating that the site is protected by SSL. The presence of this icon is often enough for the common person to accept that the site is legitimate, and therefore trustworthy. More astonishingly, several works have shown that \( \approx 73\% \) of SSL-protected sites generate some sort of validation error [17], and often users will simply “click-through” the error dialogs to get to their content. So even if users are trained to look for visual cues, often times the desire to obtain the content is stronger than the training to expect certain visual cues and feedback. The fundamental problem is that users don’t know what happens with their data once it is sent to the server.

What many users fail to realize is that the only protection indicated by these visual cues is the protection of data in transit. Once at the server, the data may be leaked, as shown above, with the user feeling confident that their sensitive information is protected. In short, the feedback provided to the user does not take into account all phases of the web transaction, even though the user assumes differently.

1.1 Web Transaction Integrity

When users interact with web applications, the transaction can be broadly broken into four phases, detailed below. These phases are content generation, information storage and retrieval, response transmission, and rendering. In order to consider a transaction high integrity, each phase must be high integrity. Below, we examine each phase, and what mechanisms exist for providing integrity.

1.1.1 Information storage and retrieval

Web servers will often interact with data stores when generating client responses. This could be as simple as accessing static files on the local disk, or accessing remote databases to obtain information. There is a large body of work examining techniques for protecting data, including input sanitization and validation [18] and database integrity solutions [19]. In addition, a new field is emerging in the area of data and system provenance [20]. Data provenance [20] tracks the creation and modification of data as it moves through and between systems. This record of information, which is sometimes called a provenance chain [21, 22] allows a client to ascertain the trustworthiness of the data being accessed.
1.1.2 Transmission

Communication between clients and servers rely on SSL [23] to provide authenticity, confidentiality, and integrity of content exchanged over the network. The use of SSL to protect data in transit over the network has become the de-facto standard mechanism for securing communication, and users have been trained to look for visual cues that SSL is present and working, before they input sensitive information. The use of SSL ensures that a malicious eavesdropper cannot monitor the content being exchanged, or modify the content being sent to the client. This helps ensure that the content being sent to the client arrives as expected. This SSL protection is the only phase users have been trained to check for, mostly by means of visual feedback cues in the browser.

1.1.3 Rendering

Once the response is at the client, the final phase of the process is to render the response on the client-side. This rendering is fraught with peril, as content no longer originates from a single source, but is instead a combination of server-hosted content and user-uploaded content as well. This shift from clients simply consuming content to actively uploading content has led to a change in attacker behavior. Now instead of focusing solely on the server, often times, attackers will simply use the server as a means of storing and disseminating their malicious payload to the clients. These types of attacks have become popularized by the mainstream media, including cross-site scripting [24], cross-site request forgery [25], and clickjacking [26]. Many defenses have been proposed, with some even implemented by popular browsers. In Chapter 2, more detail is provided regarding the various types of protections, both on the server and on the client.

While all of these efforts have increased user awareness of the threats, and defined better practices for web developers, the user is still trained to assume that the presence of SSL is enough to ensure a high-integrity web transaction. *This leads to a disconnect between what the user believes is happening and what is actually happening during a transaction.* The user must be able to verify all phases of a web transaction if she has any hope of ensuring a high-integrity transaction, not just the transmission phase.

Content generation, which includes request processing on the server, is one area where research efforts have produced little in terms of solutions for clients to validate the integrity of the transaction. Users assume that the server is well protected and maintained by knowledgeable staff. As exemplified by the examples above where a server was compromised, it is easy to see that such unfounded assumptions can create
problems, both for the client and for the company running the server.

1.2 Providing Content Integrity

Companies entrusted with sensitive information must process that information in a high-integrity manner, providing proof to the client that the processing was high-integrity. By providing an end-to-end proof that the process was carried out in a high-integrity manner, the client can base their trust assumption on more than just incomplete visual feedback, i.e. icons presented in the browser, and company reputation. What is needed is a means of generating proofs of the web transaction integrity that accounts for all phases of the web transaction.

Web applications can leverage existing technologies to create content integrity proofs that prove the transaction was carried out by a high-integrity system, that the data sources are in a high integrity state, that the transmission of the response is not compromised, and finally that the client renders the content without becoming compromised. Existing mechanisms, such as trusted computing, often suffer from severe performance limitations, which need addressed if such systems hope to see wide-spread adoption.

In addition, current systems for determining system integrity require the client to verify the integrity of the system both before and after they have interacted with the system, i.e. when the client connects, they first verify the integrity of the system, then perform any interactions with the system, and finally request another proof of the systems integrity to ensure that the system remained high-integrity during the computation. These integrity proofs are not bound to the computation being carried out, meaning that fresh proofs must be generated for each and every computation.

1.3 Thesis Statement

In this thesis we address the problem of proving the integrity of web content and the systems that generate and host web content. Such a system allows the client to ascertain the integrity of the content, by examining the integrity of the system that generated the content, and ensuring that the content being consumed is the content that the server generated.

The central thesis of this work is therefore:

Web application infrastructure can leverage integrity-measurement and
commodity trusted hardware to provide an integrity-verified foundation for web applications, while still sustaining acceptable throughput and latency under heavy client loads.

While the topic of web application integrity is vast, we will focus specifically on how to prove to clients that the server they are communicating with is high integrity, and that the content coming from the server is the content the server intended the client to receive. To that end, this thesis examines the following questions:

- **How do we construct a document integrity proof that proves the content originated from a high-integrity web system?** To this end, we examine the use of trusted computing concepts and trusted hardware, like the Trusted Platform Module, TPM. We chose the TPM due to the ease of access and relative cost to other trusted hardware solutions. It is worth noting that other secure hardware that can provide similar guarantees can cost several thousand dollars per system versus several dollars.\(^1\)

- **How can we provide a scalable mechanism to support heavy client loads?** Our choice of commodity hardware means we have to understand the performance limitations of the hardware and address the bottlenecks in such a way that we do not compromise the integrity of the system to achieve higher throughput. This and the previous point are addressed by the Spork system, presented in Chapter 4.

- **How do we provide low-latency responses when needed?** By using the TPM, the additional latency introduced can be too high for some use cases. New cryptographic constructions are explored that provide low-latency responses when needed to support specific application workloads. In addition, we examine the throughput-latency trade-offs of using these cryptographic constructions. To explore the use of these cryptographic constructions, we build the Sporf system, presented in Chapter 5.

- **How do we generalize the content integrity proof beyond relatively simple web systems?** In order to understand the integration of system integrity with other data sources, we present the Splayd framework for building document integrity systems in Chapter 6. After examining the framework, we present an example system that

---

\(^1\)The TPM IC can be purchased for as little as $2 per IC, while the IBM 4765 can be acquired for ≈$14,000 per card.
integrates database provenance records with system integrity state, allowing clients the determine if the data they are accessing is high-integrity, and also ensuring that the system tracking the provenance data is high-integrity.

1.4 Contributions

In answering the above questions, this thesis provides the following contributions:

- We show how to construct a document integrity proof that allows a client to verify that a document came from a particular server, that the server hosting the content has a known integrity state, and that the integrity state of the server is bound to the content at a verifiable time.

- We show how the document integrity proofs can be constructed to sustain high-throughput for heavy client loads. Such constructions rely on slow trusted hardware, specifically the Trusted Platform Module. In addition, we explore different cryptographic constructions that allow for fast signing operations, to support low latency responses.

- The document integrity proof construction presented in this thesis allows a client to verify the integrity of the server and content any time after the content and proof have been served. This differs from current constructions that require a client to verify the integrity of the server before interacting with the server, and then verify the integrity of the server after the computation is complete.

In the next chapter, we examine background information, followed by a general discussion of document integrity systems. Then in Chapter 4 we look at the first document integrity system that provides high throughput for web content. In Chapter 5 we detail the Sporf system, a different document integrity system that provides low-latency responses to support AJAX systems, by leveraging new cryptographic constructions. Next, Chapter 6 presents a framework for building document integrity systems, and presents a case study using a provenance-aware database system. Finally, Chapter 7 presents future directions and concludes.
Background

In this chapter, we will examine the design and construction of web applications. After describing the technologies used, we will examine the attacks that web systems can experience. Finally, we examine the wide range of defenses and countermeasures for the attacks presented.

2.1 Overview

In this thesis, the systems being studied are web systems, due to the varied demands placed on the system. In one system, throughput is the key metric, while others demand low-latency responses. This provides a way for us to explore techniques and methods for building document integrity systems that satisfy different system constraints. Below, we describe how web systems are typically built, and highlight works that aim to provide secure web computing environments.

Web transactions can be broadly broken down into two categories, static and dynamic requests. Static requests are for content that is generated before the client requests it and is stored on the web server. Dynamic requests are generated “on-the-fly” when a user requests content. There are four major phases to generating a response for a client request, namely generation, storage, transmission, and rendering. Below, we describe both types of transactions and the phases involved.

Figure 2.1 shows the steps for generating a static response. For static content, the generation phase begins once the request arrives at the web server (1 in Figure 2.1). During the generation of the response, the web server will interact with the static storage mechanisms (2 in Figure 2.1), to retrieve the requested resource (3 in Figure 2.1). The
Figure 2.1. The phases of a static web request. The processing begins once a request arrives at the server. During the generation phase, the web server will communicate with the static storage mechanisms. This response is then transmitted to the client, where it is rendered in the browser.

Figure 2.2. The phases of a dynamic web request. The processing begins once a request arrives at the server. During the generation phase, the web server will pass the request to the web application code to generate the response. The web application code will often access static storage to load templates and then populate the templates with data from other storage mechanisms, such as databases. Once the response is generated, it is passed back to the web server, which transmits the response to the client. The client then renders the content in the browser.

The resource is then transmitted to the client (4 in Figure 2.1), where the browser renders the response.

Figure 2.2 shows the process for generating a dynamic response. Again, processing begins when the client request arrives at the web server (1 in Figure 2.2). Instead of the web server generating the response directly, the request is passed to the web application code (2 in Figure 2.2). This code will process the request to generate the final response. Often, the web application code will retrieve templates from static storage (3b and 4b
in Figure 2.2), and populate these templates with data stored in a database (3a and 4a in Figure 2.2). The final response is sent back to the web server (5 in Figure 2.2). The web server then transmits the response to the client (6 in Figure 2.2), where the browser renders the response.

2.1.1 Generation

Once the client sends the request to the server, the server runs some code to generate a response. This response could be a static file on the disk, or a complex dynamically generated response that is generated by running some external\footnote{External in the sense that it is not part of the web server code.} code, such as PHP \cite{PHP}, Ruby \cite{Ruby}, Python \cite{Python}, Perl \cite{Perl}, Java \cite{Java}, or ASP \cite{ASP}. Many projects have looked at building web applications with other languages, but the ones above are the most commonly used. The goal of each application is to generate a response for the client, typically some form of HTML \cite{HTML}.

2.1.1.1 Mashups

Mashups are a popular method of building web applications. A mashup takes several data sources and “mashes” them together. One example of a mashup is an application that shows users a map of available housing in a particular area. One way to build such an application is to build a database of available housing and then develop a map library that can visualize addresses. Instead, developers can leverage existing work, such as craigslist.com and maps.google.com. Both sites provide half of the needed resources, so a developer can write the code to read the list of available houses from craigslist.com and update a map with pins for each address on maps.google.com. Both sites provide APIs for reading and updating content, and an application can leverage the APIs to build and application that doesn’t

2.1.1.2 Frameworks

In addition, often developers rely on application frameworks to provide basic functionality for their application. These frameworks exist for each of the popular languages listed above, including the Zend Framework \cite{Zend} for PHP, Ruby on Rails \cite{RubyOnRails} for Ruby, Django \cite{Django} for Python, Catalyst \cite{Catalyst} for Perl, Struts \cite{Struts} for Java, and ASP.NET MVC \cite{ASPNETMVC} for ASP. However, these frameworks do not prevent problems inherent in the underlying language, such as PHP. There are many commonly accepted practices for
developing secure applications [40], which require technical understanding of the entire computing infrastructure. Newer application frameworks take advantage of new language constructs to provide better security.

2.1.2 Storage

In generating a response, a web server will often interact with backend storage mechanisms to retrieve data. For example, when serving static content, the storage mechanism is the filesystem storing the static content. For dynamic content, the server will communicate with some sort of database, either the more traditional SQL [41] databases, such as MySQL [42], PostgreSQL [43], and SQLite [44], or a new set of databases that have become popular under the name “NoSQL”, based on the fact that these databases do not rely on SQL for structuring and querying data. Some examples of new “NoSQL” databases include MongoDB [45], Redis [46], or Google’s BigTable [47].

2.1.3 Transmission

The communication between the client and server is done via the Hypertext Transfer Protocol, HTTP [48]. This specification provides a text-based protocol for transferring data between clients and servers and is the de-facto protocol for the web today. More recently, Google has developed a new protocol designed to address some limitations of the HTTP protocol. This protocol is titled SPDY, pronounced “SPeeDY” [49]. SPDY is based on ideas from several previous protocols, namely the Stream Control Transmission Protocol, SCTP [50] and the Structured Stream Transport [51], as well as work on using HTTP over these TCP [52] replacements [53].

SPDY targets one limitation of HTTP in particular, namely that any delay in a single response delays other responses that may be ready to transmit. This is due to the HTTP specification dealing with single requests at a time. One attempt to eliminate this problem is HTTP pipelining, which allows a client to issue multiple HTTP requests on a single TCP connection, without waiting for each request to complete, however, pipelining has proven difficult to deploy. SPDY works by creating different streams for each request on a single TCP connection. Each stream is associated with a different resource being downloaded, and the server is able to interleave packets from different streams as data becomes available. SPDY also provides several other features, including allowing the server to push data to the client, instead of passively waiting for requests, and HTTP header compression to reduce bandwidth usage.
Both HTTP and SPDY are insecure protocols, in that they do not provide any mechanisms for confidentiality or integrity. Instead, both rely on Secure Sockets Layer, SSL, or Transport Layer Security, TLS to provide secure communication between clients and servers.

The SSL protocol [54] was developed by Netscape in 1994, but was not initially released to the public. The initial idea behind SSL was a mechanism to support secure communication between two remote end-points, such as two systems exchanging sensitive information, e.g. credit card numbers during an e-commerce transaction. After the initial version 1.0, Netscape continued to refine the protocols. Version 2.0 was known to be buggy and vulnerable, and in 1996, Netscape released version 3.0 of the SSL protocols, the currently deployed version.

After SSL version 3.0 was released, the Internet Engineering Task Force, IETF, began developing the Transport Layer Security, TLS, protocol [55, 56, 57] that provides the same functionality as SSL. The current version of TLS is 1.2 [57], released in 2008, and is also widely deployed in web systems to provide secure communication between hosts. TLS is often described as an upgrade to SSL, with version 1.0 of TLS being mostly compatible with SSL version 3.0. However, the differences between TLS 1.0 and SSL 3.0 are significant enough that they are not compatible with each other.

2.1.4 Rendering

One cornerstone of Internet applications is HTML [33], the markup language for web pages. HTML is is derivative of Standard Generalized Markup Language, SGML [58], and defines tags that developers use to define the structure of the page. The HTML specification has gone through several revisions over the years, with the current draft being HTML5 [59]. HTML started as the brain-child of Tim Berners-Lee in 1989 as a way of describing static content. His initial work was based on SGML, a commonly accepted language for markup at the time. In 1994, HTML version 2 [60] was published with additional enhancements, followed rapidly by HTML 3 [61] in 1995, and an updated to HTML 3.2 [62] in 1997. Shortly after that HTML 4 was published [63] with an update to HTML 4.01 [64] in 1998. HTML 5 [59] is the version currently being developed, with many new features being added to support interactive web applications. While the specification is not finalized, many browsers support features present in the specification.

Meanwhile, efforts to move from SGML to XML [65] were also underway in the form of XHTML [66]. The initial version of the XHTML specification (1.0) was published in 1999, with an update (version 1.1) published in 2001. An update is currently being
worked on as part of the HTML 5 specification, known as XHTML 5.

As pages became more complex, adding images and further styling, it became clear that a mechanism was required to separate the styling information from the content. This took shape as the Cascading Style Sheet specification [67], with the initial version being released in 1996 from the World Wide Web Consortium, W3C. In 1997, two revisions were released, namely CSS version 2 [68] and version 2.1 [69]. In 1998 work began on drafting CSS 3 [70]. One major improvement in CSS 3 is the work being done to divide the specification into a set of modules, instead of being a monolithic standard.

The last part of the client-side experience is provided by JavaScript, a scripting language supported by all of the popular browsers. In 1995 Brendan Eich developed the first implementation of JavaScript as part of the Netscape browser [71]. The language was first called LiveScript, later renamed to JavaScript, and submitted to the ECMA International standards committee. The specification was published in 1997 as ECMAScript version 1. In 1998, the specification was updated to add several new features and resulted in ECMAScript version 2 [72]. In 1999, version 3 [73] was released adding support for things like regular expressions, try/catch blocks, and better error handling. Work on version 4 was started, but later abandoned due to political differences among the editors, who could not agree on a set of features to add. Proposed features included a module system, classes, and static typing of variables. Version 5 was released in 2009 [74], but is not currently supported by any of the major vendors. Version 5 added support for a strict mode to increase error checking to prevent potential problems with ambiguous constructs. At the same time that version 5 was announced, the ECMAScript Harmony project was announced [75] to consider features from the abandoned version 4 for inclusion in the next version of JavaScript.

The HTML file obtained by the client defines what resources the browser should fetch to render the full page, including images, stylesheets, and script files. Popular web browsers include Internet Explorer from Microsoft [76], Firefox from Mozilla [77], Chrome from Google [78], Safari from Apple [79], and Opera [80]. Each browser presents challenges for web developers as each implements the standards slightly differently, leading developers to test their site on many different browsers to ensure that the client-side experience is the same in each case. This happens because each developer can interpret the standard in a different way, leading to inconsistent implementations.

With different standards and implementations, a number of tests have been developed to ensure some level of consistency. These tests are known as the ACID tests [81, 82, 83], with each version testing different aspects of the browser. The original ACID test checked
compatibility with version 1.0 of the Cascading Style Sheet specification [67]. ACID 2 tests HTML rendering [63], CSS 2.1 [69] compliance, PNG support [84], and data URIs [85]. Finally, ACID 3 tests aspects of JavaScript, HTML5 [59], and other Web 2.0 capabilities for building rich Internet applications.

After being sent to the client, the browser is responsible for rendering the content. This content comes in a variety of forms, with the predominant technologies being HyperText Markup Language, HTML [33] for content, Cascading Style Sheets, CSS [86] for defining presentation, and JavaScript [87] for interactive scripting. In addition, browsers support plugins for handling other content types, with the most popular being Flash from Adobe [88], Java [31], and Silverlight [89] to provide richer interfaces for web applications. Next, we consider the attacks that a web application is vulnerable to during each phase of the web application process.

### 2.2 Attacks

Next, we consider attacks that occur on each phase of the request processing. We begin with generation attacks, such as code modification and phishing. Next we look at storage attacks, such as SQL injection and persistent cross-site scripting attacks. Next, transmission attacks, including man-in-the-middle attacks, are described. Finally, we consider attacks that occur once content arrives at the client.

#### 2.2.1 Generation

When considering the generation phase, an attacker wishing to subvert the process must first access the web server. Once access is gained, the web application code can be modified to achieve the attacker’s goal. As an example, consider the bank in Pakistan, where the attacker modified the application code to send usernames and passwords to the attacker. Such attacks offer an easy method for an attacker to attack the application, without users becoming aware of any problems.

In addition to attacking the server and modifying application code, attackers can also attack the underlying technologies. This can something simple, like a buffer overflow in the PHP functions for filtering user input [90], to vulnerabilities in Ruby that allow for denial-of-service attacks and arbitrary code execution by the attacker [91].

Another recently popular attack on web servers is named “Slowloris” [92]. This attack occurs when a client makes a partial HTTP request, and then slowly sends new request headers to the server, without ever finishing the request. In doing this, the server can
quickly become overrun with these connections, denying legitimate requests from being serviced. Such an attack is easy to accomplish with a single machine, since a very small amount of bandwidth is actually used. Next, we consider attacks against the storage mechanisms used to build web applications.

### 2.2.2 Storage

When application developers rely on backend storage to store user input, there is the potential for users to subvert the storage, leaking sensitive information, or destroying data stored in the database. In many cases, the developer will use some user input to derive an SQL query. If the developer does not properly construct the query and sanitize the user input, then a malicious user can inject their own SQL commands. Consider query begin constructed in Listing 2.1. Note that the ‘+’ denotes string concatenation. In the PHP code, the developer is reading the value of the \$_POST[USERNAME] variable. Consider a malicious user that supplies the value junk;DROP TABLE users. In this case the final value of the qry variable is shown in Listing 2.2. When the web application executes the query, the user table is dropped, breaking the application. This type of attack is known as a SQL injection attack [93].

Another type of attack on storage mechanisms is known as a persistent cross-site scripting attack [94]. Here, instead of targeting the database, the malicious user is simply injecting content that is meant to be rendered by victim browsers, leaking information. For example, consider a forum application that stores posts in a database. The posts are allowed to contain HTML tags, including <script> tags, and this input is not filtered in any way before being stored. Consider an attacker that posts the script shown in

```javascript
var img = new Image();
ing.src = "http://evil.com/pathToImage?id=" + document.cookie;
document.write(img)
```

Listing 2.3. Example of a persistent XSS attack.
Listing 2.3. When an unsuspecting client reads the post, the browser will render the content, including executing the JavaScript. This JavaScript will make a request to http://evil.com setting a parameter in the URL to the value of document.cookie, the current value of the user’s cookie. By doing this, the attacker is able to steal cookies from users, which can be used to impersonate the user.

2.2.3 Transmission

While SSL is often used to secure communication between two parties, until very recently, it was seen as something only required for sites that handled only the most sensitive information. This is due to the perceived performance penalty for deploying SSL. This overhead is often attributed to the handshake that begins any SSL session, as it involves several cryptographic computations to establish the keys for the session. As more and more personal information is being put on the web, more sites are deploying SSL, including many major service providers like Google and Facebook. These providers have worked to reduce the impact of the SSL handshake, making it feasible for servers to support more SSL connections.

In addition to performance problems with SSL\(^2\), several other problems have come to light recently, including client misunderstanding of SSL security indicators and the presented warnings, as well as malicious proxies. In [95], the authors studied user reactions to various security indicators and warnings. Each subject in the study was presented with three possible pages and asked to login using one of two sets of credentials, either the subject’s actual credentials, or a fabricated set. The first page removed any indication of an SSL-based connection to the site, while the second page also removed site-authentication images [96], and finally a warning page was shown to the user that must be clicked through to get to the login page. The results of the study indicated that users did not bother to check for SSL indicators, and few questioned the removal of the site-authentication images. When presented with the warning page, over half of the subjects ignored the warning and logged in anyway.

Another issue with SSL deployment and user understanding relates to the use of proxies. In [97], the authors show how a Pretty-Bad-Proxy, PBP, can inject content into an SSL protected page and leak sensitive information. This is due to the subtle way in which all components of the browser interoperate after decrypting SSL protected traffic, and also how errors are displayed to the user. For example, when a proxy requests a page on behalf of a user, the proxy can return an error message to the user. The user’s

\(^2\)The performance of SSL is becoming less of an issue as computers become more powerful.
browser renders this page in the context of the site being requested, i.e. if the request is for a page at the user’s bank and the proxy returns an error, that error message is rendered as if coming from the bank server. This allows an attacker to exfiltrate data from the client’s browser and issue fraudulent requests on behalf of the user.

Another problem that can arise with the use of SSL is misplaced trust in the certificate authorities that issue certificates [98]. In this case, the certificate authority system’s were compromised allowing the attacker to impersonate sites that have certificates issued by the compromised authority. Clients accessing these impersonated sites would not be able to detect anything is wrong unless the browser developers remove the root certificate for the authority from the set of accepted root certificates. Next, we discuss the attacks that the browser itself can face.

2.2.4 Rendering

With the rapid pace of changes in browsers and the underlying technologies, it is often difficult for browsers to maintain a secure environment for clients. In addition, attackers are coming up with new and creative ways to attack the end-user and the servers hosting the content. In this section we discuss several popular attacks and some of the proposed defenses being employed by developers. The fist type of attack is very popular and relies on attackers running code in the users browser.

2.2.4.1 Cross-site Scripting

Cross-site scripting, XSS [24] is an attack that occurs when an attacker causes a script to be executed on behalf of the client. Often times this occurs when user input is presented to other clients that is not properly sanitized/filtered. In [99], the authors examine several techniques for isolating content to prevent XSS attacks. These types of isolation are shown to be insufficient, and a defense-in-depth approach is suggested that uses multiple defense techniques to prevent attack.

2.2.4.2 Cross-site Request Forgery

Another attack that is becoming popular is known as Cross-site Request Forgery, CSRF or XSRF. In this attack, an attacker posts something to one site that when rendered by the client causes requests to be issued to another site. For example, if a user is logged into siteA and logout is accomplished by an HTTP GET to http://siteA.com/logout.php then the attacker can post an image link on siteB with a src attribute pointing at the
logout script on siteA, e.g. `<img src='http://siteA.com/logout.php'/>`. When the user visits siteB and the browser tries to fetch the image, the user will be logged out of siteA. As another example, consider a banking web site that uses the following URL structure for fund transfers: `http://bank.com/transfer.php?to=5678&amount=1000.00`. In this case, $1000.00 is transferred from account 1234 to account 5678. If the image `src` attribute above is replaced with the bank transfer request, then any user logged into bank.com will transfer money to the attackers account, clearly something any user wants to avoid. This type of attack is known as a *confused deputy attack* [100].

### 2.2.4.3 Other Attacks

Most browsers keep track of a user’s *browsing history*, i.e. the sites visited by the user. For attackers, this is incredibly useful, as it helps narrow down which clients are vulnerable to certain types of attacks, like the ones listed above. This type of history detection was first proposed in [101] and in [102], the authors show that such attacks are feasible in real-world scenarios. With these types of “history sniffing” attacks, the attacker generates a list of potential sites a user may have visited and then attracts potential victims to the page. When the user visits the page, a script checks the style of each link to determine if a link is highlighted as visited or not. Another detection method, proposed in [101] looks at timing to determine if content is available in the browser’s cache. If the content loads very quickly, the content is from the cache, meaning a user has visited the site recently.

### 2.3 Defenses and Countermeasures

In this section, we consider the defenses and countermeasures for each phase of the request process. We begin with the generation phase, followed by the storage and transmission phases. Finally, we consider the rendering on the client-side.

#### 2.3.1 Generation

When the application code is executed to generate a response, a number of potential problems can arise. To better understand the code and its potential vulnerabilities, a number of different analysis techniques have been proposed, including static and dynamic analysis, as well as combinations of each. In the next section, we examine several tools for analyzing web applications.
2.3.1.1 Static Analysis

Static analysis is the process of analyzing the source code of the application to determine properties of the application. This is in contrast to dynamic analysis, where the runtime behavior of the code is monitored and analyzed. Many different approaches to static analysis for web applications have been proposed.

The WebSSARI project is a set of analysis tools that perform static code analysis of web application code. In their first work, Huang et. al. [103] develop a tool that analyzes PHP applications to determine potential vulnerabilities. In places where the code accepts untrusted inputs, run-time guards are automatically inserted by WebSSARI to ensure that vulnerabilities are not exploited. In future works, they extend WebSSARI with different analysis techniques.

One such technique is bounded model checking [104], which they provide as an extension to the WebSSARI tool in [105]. Using this approach, a model of the web application is generated and assertions about the code are checked against this model. The authors argue that bounded model checking is more appropriate than binary decision diagrams [106] in this case due to the sound and complete analysis provided by bounded model checking, as well as the ability to handle a large number of variables, which can be problematic for binary decision diagrams.

In addition to directly analyzing code, others have looked at mechanisms to generate approximate representations of output from the code. This allows for reasoning about the code without having to fully parse and analyze the code directly. This is the approach taken by Minamide [107], where the goal of the analysis is to generate approximate pages. Once the approximation is generated, it is possible to reason about potential vulnerabilities, allowing developers to determine potential vulnerabilities in pages generated from scripts.

In the Pixy project [108], the authors used data flow analysis to track untrusted inputs to sensitive functions. As an example, if a web application simply echoes input from a client, with no processing, the site is vulnerable to compromise. Instead, the input should be filtered to ensure that no user input is executed by the server or other clients. Detecting these types of vulnerabilities is the goal of the Pixy project. The authors describe these types of vulnerabilities as taint-style vulnerabilities, based on the idea that untrusted client input is tainted and should not be used as input to sensitive functions, such as the ones generating output for clients. In an extension to Pixy, the authors present a novel alias analysis for PHP code that takes into account the reference operator, which allows a variable to hold a reference to another variable, much like a
pointer in C [109]. Unlike C pointers, though, references simply allow a program to refer to a memory location by multiple names, operations such as pointer arithmetic are not supported.

Another approach to static analysis of PHP applications looks at the possible output strings generated from untrusted user input, as is done in [110]. The authors use analysis techniques similar to Minamid [107], looking specifically for instances where untrusted client input can inject some client-side script code into the output. In order to handle large programs, the analysis starts with summarizing functions and will only analyze the function in detail if code within the function uses untrusted user input. This block-level analysis allows the program to scale with the size of the web application code.

Finally, in [111], the authors examine a method of detecting script injection that builds a context free grammar to represent the possible outputs from the program. This grammar is then compared to a policy to ensure that JavaScript inducing strings cannot be injected into the output variable. By preventing the injection of JavaScript, the authors effectively block attacks that rely on injecting JavaScript, a common attack vector in web applications.

In addition to analyzing server-side code, there have been efforts to analyze browser extensions. Extensions allow developers to add functionality to the browser without modifying the source of the browser. Popular uses for extensions include advertisement blockers and extensions that make some task easier for the client. A potential problem with extensions is that, unlike web page scripts, the extensions are not constrained by the same origin policy. The same origin policy states that a script can only read responses from servers with the same origin as the script. Extensions can make arbitrary requests to arbitrary domains without notifying the client. In [112], the authors present Vex, a tool for analyzing browser extensions. The tool generates a list of source-to-sink information flows that can then be analyzed to determine if the extension is behaving maliciously. All of the above efforts have focused on determining behavior of code statically, but in addition, many have looked at ways to monitor code at run-time as well.

### 2.3.1.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis tools gather information during execution of code to determine application properties. With dynamic analysis, it is more difficult to ensure that all possible code paths are analyzed, but the analysis can provide valuable insight about the execution of the code that may not be apparent from static analysis. In

---

3Here an origin is defined as a tuple of (protocol, domain, and port).
addition to tools developed in academia, several major industrial players have developed tools to scan applications for vulnerabilities, such as Google’s skipfish tool [113], and WhiteHat Security’s Sentinel services [114]. These tools are available to application developers, allowing them to evaluate their application for potential vulnerabilities.

One popular technique for dynamic analysis is taint tracking. In taint tracking systems, untrusted input is marked as tainted and then the system monitors sensitive functions to ensure that they do not use tainted values. In [115], the authors examine several dynamic analysis tools to determine how well they perform in practice. The results of the evaluation indicate that on average, these dynamic analysis tools were able to detect less than 50% of known vulnerabilities, indicating that such tools should not be the sole-source of security evaluation for any project.

WAVES [116, 117, 118] is one example of a dynamic analysis tool for web applications. In [116], the authors present WAVES, focusing on specific types of application attacks. The tool crawls the web application, applying machine learning techniques to learn the correct behavior of the application. After the learning phase, WAVES injects malicious content into the application looking for potential vulnerabilities. Unlike previous tools, WAVES can cause harm to the web application by injection malicious input. In order to prevent damage, the authors extended WAVES to support non-detrimental scanning. This is done by classifying the operations of the web application into categories, and generating dummy data for operations that can potentially harm the application.

In addition to WAVES, others have also looked at various techniques for detecting vulnerable web applications. Another project that relies on dynamic analysis is the NoTamper project [119]. Here, the authors are looking at classes of attack where client-side form validation is the only mechanism for validating input to the web application. As an example, consider a shopping cart application, where clients can purchase items. If the only validation of the shopping cart checkout form is done on the client, then a malicious client can easily disable any such checks and submit incorrect values, such as negative prices, in effect getting the company to pay the attacker, while still receiving goods and services. To detect these types of attacks, NoTamper submits input that is classified as benign and input that is considered hostile, and compares the responses. If they responses are similar, then there is the potential that the parameter can be tampered with, leading to the types of attacks detailed above.

Another black-box technique for finding vulnerabilities in web applications is taint inference [120]. This inference is done by comparing input and output from the various stages of the web application to determine if any input is used directly in the output.
If so, a potential vulnerability exists and needs further exploration to determine if the application is vulnerable. The advantage of taint inference, as opposed to full-scale taint tracking, is the lower overhead, both during development of the web application infrastructure and during run-time. However, dynamic analysis techniques have been shown to find fewer vulnerabilities than static analysis.

2.3.1.3 Hybrid Approaches

In the above analysis frameworks, the authors focused on either static, or dynamic analysis. In some projects, researchers have looked at using both static and dynamic analysis, to gain a better understanding of the web application code. This allows then to identify and contain certain vulnerabilities that either approach alone could not. Below, we describe several of these types of systems.

One such hybrid tool is Saner [121], which looks for improper, or insufficient input sanitization. Saner uses static analysis to identify and model custom sanitization routines in the web application. Any routine that is flagged by the static analysis as insufficient is dynamically analyzed. The static analysis models the sanitization routines as “taint-aware” automata, allowing the tool to analyze the effect of processing tainted and untainted strings. If the routine is flagged, a static tool would simply return a warning to the user and terminate. Saner on the other hand performs a dynamic analysis to determine if the reported routine is in fact insufficient. This is done by testing the routine on known-malicious inputs and checking if the routine reduces the malicious input to a benign output. In performing this dynamic analysis, Saner is able to eliminate false positives, allowing developers to focus on fewer warnings.

Another project has looked at analyzing Flash advertisements, a popular delivery mechanism for advertisements due to the ubiquitous nature of Flash plugins. In [122], the authors developed an analysis framework for Flash advertisements. The tool, named OdoSwiff, begins by statically analyzing the Flash file, looking for specific indications of malicious behavior. As an example, the tool will look for specific sections of the Flash file marked as image data, and attempt to parse the data using standard image parsing libraries. Failure to parse the image is a good indication that the file is potentially malicious. After the static analysis, the dynamic analysis is run to determine any potentially malicious run-time behaviors. The dynamic analysis gathers information about referenced URLs, functions called, and other actions taken by the code. All of this information provides a view of the Flash application’s run-time behavior, allowing users to determine if the code is malicious.
In the above tools, the authors are examining application code for specific malicious behavior. While this works well in practice when all of the code is available for analysis, in some cases, the code is not available until run-time. As an example, consider the Facebook application platform. Applications written for the platform can leverage a restricted subset of the JavaScript language. One implementation of the Facebook application platform relied on static rewriting to analyze and rewrite unsafe JavaScript operations before sending the JavaScript to the client. In [123], the authors analyze this technique, formalizing the rewriting, and then showing why static rewriting alone is insufficient. They then propose a set of run-time checks and source-to-source transformations that allow for a more flexible enforcement mechanism that does not require all code to be present for static analysis. In addition to analysis of single applications, people have also looked at building secure mashups.

2.3.1.4 Mashups

One limitation of current mashup design is the security model. Currently, developers have two choices for building their mashups. The first is to place all content on a single page, allowing all sites access to all resources. The second is to isolate each component in an iframe making content sharing more difficult. In using isolated iframes, developers can leverage the recently introduced postMessage() function to exchange messages between components. However, a problem was discovered with postMessage() making it trivial for a malicious script to forge messages [124], for which several browser vendors have already issued the proposed fix [125].

In addition to working within the confines of existing browser architectures, others have looked at radically new designs for mashup architectures as well as building new frameworks for secure mashup development. In [126], the authors describe a vision for the browser as a platform for building mashups, examining the current behavior of browsers for content and script isolation, and how each isolation should change to support mashups. Others have proposed new abstractions, which they call MashupPOS [127], for existing browsers that allow developers to build mashups without the current workarounds. The CompoWeb project [128] is another model for mashup applications that leverages existing browsers, but adds new features to specifically support mashups.

In addition to modifying current browsers, others have looked at building frameworks for mashups that leverage existing browser features [129, 130, 131, 132]. These frameworks provide libraries for mashup developers to build secure mashup applications, often
presenting new communication abstractions for passing data between the various data sources within the mashup.

2.3.1.5 Frameworks

Many frameworks have been developed over the years to aid developers in building applications. Many of the languages used have been shown to allow insecure practices, such as using unfiltered user input. These languages often lead to developers having “checklists” of common tasks to secure their applications during the development and deployment cycles [40]. Others have proposed switching to languages that do not allow such insecure coding practices, such as the Haskell framework presented in [133]. Here, the authors argue that by using the strongly typed language Haskell, many common web application vulnerabilities can be prevented, such as injection attacks. Other proposals for switching to secure languages such as Jif [134], a language based on Java that provides information flow security. In [134], the authors present Servlet Information Flow, Sif, as a set of modifications to the Jif [135] compiler to support dynamic applications. SELinks [136] is yet another framework for building applications that leverages secure language constructs. SELinks uses the Fable [137] type system to ensure that the security policy is correctly enforced.

Another approach to building secure web applications does not require developers to switch programming languages, instead asking them to switch toolchains. Tools like Volta [138] and Ripley [139] use existing languages for application development, instead modifying the output of the compilation process to enhance security. Volta allows developers to write their application as a single program, and then the Volta compiler divides the program into modules for the client and server. For example, an application may present a form to the user to gather information. Instead of developing the form and form handling code in multiple languages, the developer simply writes the application as if everything is occurring on one system and then the compiler will generate the correct code for the client and server side applications. Ripley addresses a problem where the server side code expects some piece of client-side code to run, such as input validation. In this case, moving the input validation to the client has several benefits, including making the application more responsive to the client. However, an attacker can block the execution of the validation code and submit data that fails validation. Ripley ensures that any client side validation code is replicated on the server, ensuring that a malicious client cannot subvert the validation process. Both Volta and Ripley allow developers to use languages they are already familiar with, instead of requiring them to change to new
languages.

### 2.3.2 Storage

In each of the storage attacks, the storage is either being used to store malicious content, or being directly attacked by the attacker. The root cause in both cases was that user input was not adequately filtered by the application before being processed. In [140], Su and Wasserman define a formal model for SQL injection attacks, and then build a framework for preventing these types of attacks. They observe that when a malicious input results in a SQL injection attack, the parse tree for the SQL query contains additional nodes. By checking for these additional nodes, the system can prevent SQL injection attacks.

In addition to checking input to SQL commands, other data validation strategies are required to ensure that user input does not subvert the system. The Open Web Application Security Project, OWASP [141] provides a set of guidelines for application developers to sanitize user input and prevent injection attacks [18]. These techniques include things like white- and blacklisting known good and bad inputs, domain-specific validations, such as checking for validly constructed email addresses, phone numbers, and credit card numbers. Also discussed is methods for sanitizing input, which typically includes removing, or escaping certain characters that can lead to injection attacks.

In addition to commonly accepted practices for protecting backend storage mechanisms, several projects have looked at building systems that provide protection against untrusted user input. One such project is the DBTaint project [19], which adds taint tracking to the PostgreSQL database. The prototype includes support for Perl and Java-based web applications, relying on mechanisms for each language to track tainted information in the application [142, 143]. When the database interface is invoked, DBTaint rewrites the SQL query to propagate taint information. By tracking taint information, application developers can prevent malicious input from accessing sensitive functions in the web application.

As cloud computing becomes more popular, many developers are leveraging cloud storage mechanisms, meaning application developers no longer have to develop and maintain a costly server infrastructure for their applications. However, there are many unanswered questions for cloud computing, such as how to prevent data leaks and ensure isolation between users. In [144], the authors develop a file storage service that returns control of user data to the user, and allows the user to allow or deny applications access to the data, without the data being uploaded to each service.
2.3.3 Transmission

Google has recently started allowing most services they provide to be accessed via an SSL protected connection, including their popular search service [145]. In addition to providing services over SSL, the developers of the popular Chrome browser [146] have developed a mechanism to reduce the number of message exchanges to establish a TLS connection. This enhancement is known as a TLS False Start [147], and eliminates one full round-trip during the TLS handshake phase. In addition, Google has experimented with re-using TLS connections for a single client over longer periods of time, eliminating the handshake for return visits.

In addition to improving the performance of the protocols, others have looked at ways to off-load portions of the web transaction to other systems, such as content distribution networks, or CDNs. In [148], the authors propose a technique called SSL splitting, where the server hosting content handles the SSL traffic, but the actual content is sent to the client from a CDN. In doing this, the client can validate the integrity of the content using the SSL information. In SSL splitting, the SSL headers for each packet of content are generated by the hosting server and sent to the client, while the actual content is sent from the CDN. When all information arrives at the client, the client performs the standard validation and uses the content. One limitation of SSL splitting is that content is not encrypted, as the CDN does not have the keys to encrypt data for the server.

In addition to work being done to improve the performance of SSL/TLS in software, a number of different companies sell hardware to accelerate SSL connection processing. Many hardware vendors provide add-on cards and systems to process SSL transactions [149, 150, 151]. The hardware handles all aspects of the SSL transaction, freeing the system processors to handle other tasks. In some cases, these hardware-based systems can perform over 17,000 RSA operations per second [152]. Most of these add-on cards are available for a nominal cost, and provide support for high-throughput SSL systems.

2.3.4 Rendering

Next, we look at the defenses and countermeasures for client-side attacks, including cross-site scripting and cross-site request forgery.

2.3.4.1 Cross-site scripting

XSS defenses can be broadly broken down into two categories, namely server-side and client-side defense mechanisms. On the server, the goal is to filter user input before
sending content to other clients. XSS-GUARD [153], XSSDS [154], and Spectator [155] work by examining inputs and outputs on the server to determine if content represents an XSS attack. SWAP [156] works by rendering content on the server before sending the response to the client. If scripts are detected that are not on a whitelist of scripts, the content is flagged as an attack. By placing the defense mechanism on the server, clients are not required to deploy any additional software, increasing potential adoption, but using valuable server resources. In addition, different browsers parse and render content differently, making it difficult for servers to detect all attacks.

Browsers make an ideal location for XSS defenses, as the browser is able to detect any scripts, and also has knowledge of what information the client considers sensitive. Many approaches have been proposed for client-side XSS defenses, including taint-tracking [157] and application-layer firewalls [158, 159] that allow or deny connections to specific servers based on a client’s policy. While these techniques allow for the greatest control by the client, they also require the client to understand the details of the application and potential attacks, something that is difficult, even for trained experts.

Another potential defense is through annotating page contents in such a way that the browser can detect content that is not part of the original content. For example, ⊕JS [160] encodes legitimate JavaScript files by XORing each file with a key known only to the client and server. The client decodes the file before execution. If an attacker injects a script into the page, the decoding operation will not succeed and the script will not be executed. Another annotation method uses XHTML namespace support [161]. For each client, a unique namespace identifier is generated and added to each tag. The browser will only render correctly formatted XHTML content, so any script injected into the page that does not have the correct namespace will prevent the page from rendering, preventing attacks from occurring, but creating a potential denial-of-service attack. Blueprint [162] provides a canonical encoding of content that is decoded on the client before being rendered. By encoding potentially malicious content that must be decoded by the browser, clients can detect potential attacks in the decoded content and block execution. Finally, Document Structure Integrity [163], DSI, provides a system that ensures injected content is easily detected. DSI generates unique tags to delimit content that is untrusted. These tags are random, making it difficult for an attacker to determine the correct tag to escape the wrapped content. The browser is modified to ensure that all tags match before rendering the content, and that any content between tags should be rendered as text, and not executed as script content.

While each of the above defenses looked specifically at rendering HTML content in the
browser, another type of XSS attack looked specifically at an algorithm used by browsers to detect content types for unknown filetypes. This is known as content-sniffing, and in [164], the authors show how a maliciously crafted PDF file can cause a malicious script to be executed in the client’s browser. The authors then propose a different algorithm for content sniffing that does not allow content to be mis-detected.

2.3.4.2 Cross-site Request Forgery

The most commonly accepted method for protecting against CSRF attacks is to employ CSRF tokens. These tokens are randomly generated tokens that allow a single request to occur. To obtain a token, the client must fetch a specific page, such as a form that contains the token. Only if the token matches a token the server knows about will the request be processed. This, in conjunction with the same origin policy [165], means that an external site, such as siteB in the above example, prevents CSRF attacks. The same-origin policy states that an origin, defined as a tuple of \((domain, protocol, port)\) can only read server responses from the same origin. This policy is applied to the scripts running on the page, so any scripts running on siteB cannot read a response from siteA, preventing the scripts on siteB from stealing a CSRF token.

Another proposal, first proposed in [166], and being supported by Mozilla [167] is the Origin header. The origin header will be placed in POST requests to web applications, and will include the same origin information as detailed above, i.e. \((domain, protocol, port)\). Web applications must then be configured to service only POST requests for state-modifying requests, such as login and logout, and only from origins that are whitelisted. Unlike the Referrer header [168] that is currently sent with most requests, very little private information is sent as part of the origin header, meaning that suppression of the header by firewalls is unlikely\(^4\).

2.3.4.3 New Browsers

Many new browsers and browser architectures have been proposed that aim to mitigate many of the vulnerabilities and attacks described in previous sections. This section looks at several of these proposed browsers, and how they accomplish the task of protecting the user from malicious content. Most of the browsers described here are research prototypes, with the exception of Chrome, which is actively developed and widely used. Chrome’s

\(^4\)The Referrer header passes information about the current path in the application to other domains, and is often suppressed by firewalls as it leaks potentially sensitive information.
market share in 2010 surpassed 10%, while Microsoft’s Internet Explorer has dipped below 50% [169].

Google released the Chrome browser was first released in 2008 [78, 146], and quickly gained market share. Chrome uses the popular WebKit layout engine [170], also used by Safari [79], Mobile Safari [171], the Android Browser [172], and many others. Chrome was the first browser to support process-isolated tabs, allowing the browser to continue functioning even if a page caused a single tab to crash. In addition, many new security features are present that are slowly becoming available in other browsers as well. In [173], the authors outline the changes made by the Chrome developers to support a more modular browser architecture that is also employed by many of the research prototypes described below.

The first browsers developed featured a monolithic design. All of the parsing, rendering, and plugin execution occurred within a single process. Even when popular browsers added tabs, each tab within the window shared a single process. As more and more dynamic content was added, this became problematic, since browsers that used to simply parse HTML and display it are now executing a large amount of dynamic content. To combat the instability and potential insecurities of this dynamic content, researchers proposed a modular design. The OP browser [174] is one such modular browser, with the Illinois Browser OS [175], IBOS, building on the ideas of the OP browser. The OP browser isolates each component of the browser using processes. These processes render content and send the output to another process for final rendering. Access to other components is mediated by a browser kernel, which also handles resource allocation for the various browser processes. IBOS is a new OS designed to use the browser as the only interface, with the browser kernel and processes running on a secure kernel. This is very similar to the recently announced Chrome OS [176] from Google.

While the OP browser was designed as a fully-modular system, with each component isolated in a separate process, the Gazelle browser takes a different approach, instead providing isolation between different instances. Here, an instance corresponds to a single origin. Plugins are also isolated to separate processes. Again the Gazelle browser kernel manages resource allocation for each browser instance, and any operating system access required by a browser instance is mediated by the browser kernel. The idea of isolating content based on origin can be seen in the most recent versions of the popular browsers, where each tab is isolated as a separate OS process, with a separate process managing the global window that manages the tabs. This model is used by recent versions of

---

5Recall that an origin is defined as a tuple \((domain, port, protocol)\).
Firefox, Chrome, Safari, and Internet Explorer.

Another new browser is presented in the FlowwoLF [177] project, which developed a framework for building end-to-end secure web applications. The browser in the FlowwoLF project is based on the Lobo browser [178], a Java-based browser. The FlowwoLF browser leverages existing mandatory access control (MAC) to build a secure platform for web applications. Each layer of the system provides a MAC enforcement mechanism, such as SELinux [179] for the OS layer, Xen [180] to provide isolation at the VMM layer, labeled IPSec [181] to control network access, and Jif [135] for label enforcement at the application layer. By layering the MAC enforcement, each layer only controls access to resources it has, not worrying about other resources that are handled by other layers.

As browsers have become more complex, with ever increasing code-bases, it becomes more difficult to ensure that the browser is secure. While the recent move to modular designs aids in the verification process, the code is still large and difficult to verify. The Featherweight Firefox project [182], provides a formal model of a browser. This formal model allows developers to test different security policies in an abstract model, without requiring the developer to deal with existing legacy code for current security policies.

All of the above browsers attempt to provide a more secure environment for web content, as the content being consumed by clients becomes more dynamic, and is obtained from different systems and combined in the browser. Next, we look at other proposals that work within the confines of existing browsers, instead of developing completely new browsers.

### 2.3.4.4 Modifications to Existing Browsers

Existing browsers are widely deployed, making it difficult for new browsers to make significant gains in market share. Existing browsers also provide a wide variety of features that users become used to using, making it difficult to convince them to switch to new browsers. To deal with this, many efforts have been made to modify existing browsers to enhance the security and privacy protections provided, while providing the features desired by end-users.

Several popular features of browsers can lead to privacy leaks for end users, such as the content cache, cookie jar, and history. A malicious site can use available information to determine browsing history and track users as they interact with the site. As a simple example, consider a site that displays the message “Hello, guest”. The first time a new user clicks a link on the page, they are first asked for their name. This information is stored in a cookie in the browser. For all subsequent visits, the message changes from
“Hello, guest” to “Hello, Jim”. In addition to these simple personalization efforts, cookies can also be used to track information, such as what pages and items have been looked at, allowing sites to tailor advertisements and marketing to those items of interest.

It is also possible for sites to determine browsing history by looking at information about links on the page. For example, if a user has visited bank-a.com and then goes to bank-b.com, and the second site includes a link to bank-a.com, the link will be colored differently if the page has been visited by the user. It is a simple matter for a script to examine to color of the link to determine if the link is highlighted as visited. To prevent these types of leaks, SafeCache and SafeHistory [183] are extensions for the popular Firefox browser that isolates the cache and history for a page, based on the same origin policy already enforced by the browser for JavaScript.

To prevent cookie tracking, the Doppleganger extension [184] uses two processes to determine page differences for accepting and rejecting cookies. If significant differences are detected, the user is shown the differences and asked to choose to accept the cookie from the site or not. After accepting or rejecting a cookie, the decision is recorded by the browser extension and the user is not prompted again for that particular site. This allows the user more fine-grained control over cookies, while also allowing the client to see the effects of accepting or rejecting a cookie.

One popular theme of the new browsers proposed in the previous section is isolation of various components. Several mechanisms have been looked at for isolating content, either by using a proxy [185], or allowing native code execution within a tightly controlled sandbox as provided by the Native Client plugin developed by Google [186]. In addition techniques have been developed to isolate user-generated content within a page [187], by a process known as script accenting, where scripts are only able to access resources with the same accents, thus preventing XSS attacks.

In addition to isolation of content, as browsers are providing access to more services, several enhancements have been proposed to support secure access to resources. One proposed enhancement is W3BCrypt [188] to support encryption in the browser, allowing sensitive information to be sent via untrusted web servers, such as a web email system. This differs from the protections provided by SSL/TLS, which aim to protect content in transit.

Several large web applications have recently started offering SSL/TLS protected access to their services, but often as an option, and not the default. These services typically use SSL/TLS to login, and then redirect the user to the unencrypted version of the application to reduce load on the server, thereby increasing throughput and reducing latency.
Gmail and Facebook are two such services. After adding these options, users requested a mechanism to always enable SSL/TLS. While some sites like Google and Facebook have since added these options, other sites have not been as quick to respond, resulting in the development of ForceHTTPS [189] which ensures that the services listed in the extension are always accessed via HTTPS and not insecure HTTP.

### 2.3.4.5 New Policies

As the web has evolved from the relatively simple text-only pages to the more dynamic web applications used today, the policies enforced by browsers have evolved as well, often in an ad-hoc manner. There have been proposals for clean-slate development of new security policies, including browser-wide policies [190, 191] and policies for specific parts of the browser, such as client-side storage [192] and plugins [193]. Other work includes looking at server-side policies for content access as well as formally modeling policies.

The most common policy employed by browsers today is the *same-origin policy*, which states that a JavaScript function can only access resources from the origin of the currently loaded page, and not from other origins. Recall that an origin is defined as a tuple \((domain, port, protocol)\). The same-origin policy prevents an AJAX request from one site to another site from reading the response. Exceptions to this rule include POST requests, which can be sent to any origin, as well as loading new content, such as scripts and images. Any scripts loaded from an external site will be marked as part of the current site, i.e. the origin will be the same even if two scripts came from different servers.

While the same-origin policy is widely used, as applications become more complex and access resources from different domains, new policies are needed. ConScript [194] is one such proposal for supporting fine-grained policies for JavaScript, while another proposal looks at allowing scripts to explicitly share data [195]. In addition to new JavaScript policies, others have looked at ways to prevent script injection attacks. The BEEP project [196] aims to prevent script injection attacks by providing a security hook that is called before any script is executed. This hook is used to check a policy for each script execution, and can prevent scripts from executing if the policy does not allow that particular script. Finally, others have looked at ways to instrument current JavaScript files to enhance browser security [197] at run-time.

---

6Client-side storage is part of the HTML5 draft specification that allows applications to access storage mechanisms such as a database for storing application data.
In addition to JavaScript, others have looked at policies for other types of content. One approach is a server-side policy named SOMA [198]. Under SOMA, the origin server provides a manifest file that specifies what origins content is to be loaded from, while content providers provide a script that determines if an origin can embed content, such as images. For example, if social-site.com wants to embed an image from image-host.com, then the manifest file on social-site.com would include image-host.com in its manifest file and image-host.com would need to include social-site.com in its list of allowed sites to download content. One advantage of SOMA is the ability to deploy gradually, not requiring a single point in time where sites must all enable SOMA.

Another actively developed policy is Content Security Policy, or CSP [199]. CSP is a new policy language, developed by Mozilla, that allows site administrators to define what origins the browser is allowed to download resources from. Unlike SOMA, CSP is only deployed by the site, and not by content providers. CSP allows the administrator to define a whitelist of origins that the browser can download content from, and allows fine-grained control based on the type of the resource, such as images, scripts, and stylesheets.

In addition to new policy languages, several projects have looked at formally modeling security policies for web applications. In [200], the authors looked at methods for developing an abstract policy language for web applications. This abstract representation is then compiled into a set of validation checks, which are enforced by a security gateway deployed by the site administrator. Others have looked at ways to model browser-side policies, such as the Alhambra project [201]. Alhambra allows browser developers to model new policies and check invariants before making the effort to implement the proposed policy.

In this chapter, we have surveyed web applications, including the technologies used to build them, the problems faced by application developers, and newly proposed mechanisms and techniques for securing them. In the next chapter we examine document integrity systems, describing the properties they provide as well as example systems.
Document Integrity Systems

In this chapter, we describe document integrity systems and the guarantees they provide. Document integrity systems provide guarantees about the content they are hosting and the integrity of the system hosting the content. 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 of the server. 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:

\[
\text{IS}_s(n)
\]

Here \( n \) is a nonce, or challenge, that ensures the freshness of the generated proof. This proof satisfies guarantee \( b \). 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:
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 $\text{IS}_s(\cdot)$ 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 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.

In order to ensure the freshness of the integrity proof, we consider two distinct proof constructions. Here, $\mid$ denotes concatenation. The first is:

$$\text{IS}_s(h(d) \mid n)$$

Here, the integrity proof includes not only the hash of the page, binding the document to the integrity proof, but a client provided nonce. This ensures that the integrity proof is generated at the time the document is served to the client. However, such a construction requires the server to generate a fresh proof for each page served, even if the page is static. With the use of slow trusted hardware, this can impact the throughput and latency experienced by the client.

Consider another construction, where 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. The timestamp is written:

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

Where $\text{IS}(h(t_i))$ 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:

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

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. This construction eliminates the need for the server to generate a fresh proof for every client. Next, we discuss available hardware and software for building document integrity systems, beginning with the hardware that is used in this thesis, namely the Trusted Platform Module.

### 3.1 Trusted Computing Hardware

The Trusted Computing Group [202] has published a set of standards for trusted computing [203]. Included in this set of specifications is the specifications for the Trusted
Platform Module, TPM; a low-cost hardware device that facilitates the construction of trusted computing environments. The core idea behind the trusted computing initiative is to develop systems and software that consistently behave in specific ways, with enforcement by the hardware and software present on the system [202].

The TPM itself is a passive device that provides tamper-evident storage for cryptographic keys and measurements. Measurements in this case are cryptographic hashes, i.e. SHA-1 [204] hashes, stored in the TPM’s Platform Configuration Registers, PCRs. The registers provide an extremely limited interface. During power-on, the registers are set to a known value, e.g. zero. When a measurement is stored in a PCR, the TPM.extend operation is used. The extend operation concatenates the old value of the PCR with the new value, computes the hash and updates the register with the new hash. This hash chain allows a verifier to determine if the measured values are the expected values, by recomputing the hash chain using a list of expected values and comparing the reported and computed results.

In addition to the storage mechanisms, the TPM provides a set of functions for cryptographic operations, including key generation, signing, and encryption. The key generation functions are used to generate keys that are accessible only to the TPM that generated the key. These keys are then used by the TPM to perform various signing and encryption operations. The signing operation, denoted as TPM.quote, is used to report the values in the PCRs. This report is commonly called a quote, or attestation, as it is attesting to the current measured state of the system. Attestations are generated by the TPM when a verifier requests the attestation. The verifier provides a nonce to the TPM, as a means of ensuring the freshness of the report. The TPM reads the PCRs, and signs the data, along with the nonce.

One disadvantage to using the TPM to generate attestations, is the time that it takes to generate each response. In [205], the authors measured the latency introduced when using the TPM, and found that in most cases, the additional latency was high. The authors then propose changes to the scheduling algorithms, as well as adding additional TPMs to service requests. For this thesis, we consider typical server systems that use a single TPM, with the simplest first-come-first-served scheduling algorithm that is the default for current implementations. Under this scheme, each request that must be serviced by the TPM experiences an increased latency. For example, a client requesting a quote will experience at least 900 milliseconds of additional latency. As the number of clients increases, the latency additional latency will also increase, since the TPM can

---

1This is the minimum time required to generate a TPM.quote() on our experimental systems.
During power-on, the chain of trust starts with the BIOS measuring itself and the bootloader. The bootloader then measures the kernel and initial RAM disk (initrd). The kernel is then responsible for measuring the applications. This chain allows a verifier to validate all code loaded by the system.

However, without the support of the software, the TPM measurements include only the firmware, BIOS, and bootloader. In order to continue the measurement into the software, software that is “TPM-aware” must be used. This includes a bootloader that measures the OS and supporting data, and finally the OS must perform measurements of the running system. This chain of measurement is depicted in Figure 3.1. Several bootloaders exist that support trusted computing including TrustedGRUB [206], and Intel’s tboot [207]. Other possible hardware setups exist to support the efforts of trusted computing, as we discuss in Chapter 7.

3.1.1 IBM 4758

The IBM 4758 [208] is an add-on card available from IBM that provides a secure execution environment. Smith provided a theoretical understanding of the 4758 in [209], developing the concept of outbound authentication, where each software module measures the next module, before passing control to the next module. Unlike the TPM, the 4758 provides several defenses against physical tampering, going as far as to destroy the memory of the card should an attack be detected. One major disadvantage to the IBM 4758 is the cost of each device and license. The device alone can cost tens of thousands of dollars compared to the TPM, costing less than two dollars per chip, making widespread deployment unlikely. In contrast, the TPM is available on many systems today from
popular manufacturers like Dell, HP, and IBM.

3.2 Integrity Measurement

In order to ascertain the current state of the system, a verifier must know more that what hardware and firmware was responsible for booting the system. The verifier must also be able to determine what software is running, and possibly what configuration options have been set by the administrator. In [210], the authors survey various methods for “bootstrapping” trust in systems relying on commodity hardware using various integrity measurement mechanisms. The goal of integrity measurement systems is to gather information about the current state of the system to report to a verifier. In one use case, the verifier is a remote client wishing to access resources of the integrity measured system. A remote attestation is generated by the system and sent to the verifier. If the verifier accepts the attestation as valid, he can then access the resources of the system. In order to support these processes, the OS and software must be able to gather integrity relevant information. Below, we detail several solutions that aim to provide integrity information about the system.

3.2.1 Software Solutions

Before the TPM became commonplace, there was still a need to ensure the integrity of systems. Without hardware support, the only option available was to rely on software to perform the measurement and reporting tasks. One method for determining the code to be executed is to report a “checksum” of the code. Several proposals exist for generating secure checksums of code to be executed. For example Pioneer [211] aims to provide highly optimized checksum computation code that is sensitive to modification. When the code is run, timing information is gathered and reported. If the time to execute the checksum code falls outside of a set of tight constraints, then the system is considered suspect.

Other systems have looked at using software checksums over the memory of the process. In order to thwart adversaries, the memory included in the checksum is selected pseudorandomly, meaning an adversary cannot anticipate what regions will be queried and pre-compute valid answers. This is the approach taken by projects such as SWATT [212], Genuinity [213] and [214]. However, these systems have been shown to be insufficient, as attacks have been discovered that allow an attacker to subvert the system without detection [215, 216]. Given these attacks, other efforts to provide
software-based integrity measurement have explored self-modifying code as a mechanism for preventing attack [217]. The authors show that such attacks can be prevented when the checksumming code is able to modify itself during execution.

Other software based solutions have looked at building the measurement into the kernel and shell [218]. In the CASS project, the authors provide prototype systems that compare cryptographic signatures of executables to lists of known values before executing any code. The project resulted in two different prototype kernels, as well as a portable shell that validates signatures before executing any code. In all of the above solutions, the limiting factor has been access to lower layers of the system. In each case, access to a lower layer by an adversary means that the system can be compromised without being detected. Next, we look at solutions that base their measurements in hardware, such as the TPM described previously.

### 3.2.2 Hardware Solutions

Many different projects have examined methods of using trusted hardware to provide high-integrity computing environments. One such effort is the Linux Integrity Measurement Architecture, IMA [219]. IMA started as a set of patches to the Linux kernel that modified the Linux Security Module framework to measure any code that is loaded before the code is executed. IMA interfaces with the TPM, extending each measurement into a PCR on the TPM, as well as storing the list of measurements in kernel memory. Given a list of loaded code and the PCR value, a verifier can determine what code has been loaded by the system, checking the list for known bad code. Recently, the IMA system has been made an official part of the Linux kernel, taking advantage of recently added Linux Integrity Measurement, LIM, hooks [220] added to support various integrity measurement systems.

One downside to IMA is that the measurement list can grow quite long, making verification slower. An extension of IMA, the Policy Reduced Integrity Measurement Architecture, PRIMA [221], aims to solve this by analyzing the systems mandatory access control policy, specifically the systems SELinux [179] policy. The authors show that the policy can be setup to enforce an integrity model known as Clark-Wilson lite, an approximation of the Clark-Wilson model [222]. With PRIMA, only code that can directly impact the integrity of the system is measured, instead of all code that is loaded, reducing the number of measurements taken. This is done by analyzing the policy to determine what code can impact the high-integrity processes, and what code is prevented from interacting with the high-integrity processes by the system’s MAC policy. In those
cases where a low-integrity process must pass data to a high-integrity process, the data must be passed via a filtering interface that ensures the integrity of the data before allowing the high-integrity process to access the data.

While IMA, and PRIMA, aim to provide system-wide integrity measurement, other projects have looked at how to ensure the integrity of a specific part of the system at run-time. By limiting the scope of measurement, it is possible to perform deeper validation, using available context to evaluate the integrity of the system. This is the approach taken by LKIM [223], which examines the integrity of a running Linux kernel. In this case, the kernel uses some well-defined data structures that should not change during run-time, except in very specific ways that can be easily tracked. Using this knowledge, LKIM can monitor the kernel and ensure that the data structures are not modified maliciously.

Other fine-grained measurement approaches have also been proposed that aim to provide hardware-supported attestation of just integrity-relevant code, instead of all running code. BIND [224] uses this to ensure that the code expected to be executed is the code that is actually executed. This is done by hashing the process code and input data. This measurement is signed and returned with the result, allowing a verifier to determine if the correct code was used to process the input.

BIND made several assumptions about the underlying hardware and software, such as the presence of a secure kernel. However, such assumptions are difficult to satisfy in practice, leaving the system vulnerable. One method of protecting the system from potentially compromised software is to rely on the hardware’s ability to isolate code during execution. Both AMD and Intel provide mechanisms for this, known as Secure Virtual Machine, SVM [225] or Trusted eXecution Technology, TXT [226] for AMD and Intel, respectively.

These technologies allow the system CPU to be reset and then execute a specific piece of code. This is used by the Flicker project [227]. When a high-integrity piece of code needs executed, the system CPU is reset, and the code is measured and executed. During execution, interrupts and other external inputs are disabled, as well as all but a single CPU core. This ensures that the code being executed is not tampered with during execution. After execution, the TPM stores the measurements, for later examination by a verifier.

In addition to the TPM, several projects have looked at building secure co-processors to support high-integrity systems. CoPilot [228] is one such co-processor that monitors the integrity of the executing kernel. Unlike the TPM, the CoPilot processor is an active device with its own memory and processor. This has the distinct advantage that
the host system requires no modification beyond installing the co-processor. The co-
processor then monitors the execution of the kernel on the host and detects malicious
modifications.

While CoPilot examined techniques for using commodity hardware, AEGIS [229]
looked at building an entirely new processor architecture that provides a secure comput-
ing environment. Unlike other proposals, AEGIS provides several modes of operation,
a normal mode, a secure mode, and a private mode. In secure mode, any tampering is
detected, and in private mode, computations are done with encrypted data to ensure
privacy. The processor alone is not enough though, and the TrustedBox project [230]
built a secure computing environment based on a secure kernel that measures code
before execution. This is similar to the approach taken by IMA and PRIMA, but on
different hardware.

3.2.3 Privacy Concerns

One of the major concerns with remote attestation protocols is that they leak informa-
tion about the system. This information can be used to focus the attacks of an adversary.
Several proposals exist that aim to solve this privacy problem. One proposal is for prop-
erty attestations, where the system is able to prove properties of itself instead of sending
a list of running code. This has the advantage that an adversary cannot determine if the
system is running vulnerable code.

In addition, the TCG has adopted two different protocols for protecting the privacy
of systems generating attestations. The first calls for a Privacy CA, a trusted third
party that manages certificates for attesting servers. When a client wishes to validate a
system, they contact the Privacy CA to obtain a certificate for a key used by the TPM.
This prevents the verifier from determining exactly which TPM generated an attestation,
while still being assured that the platform is high integrity. This third party requires
trust from all parties involved, and as such has not seen wide-spread adoption.

Another protocol adopted by the TCG is known as Direct Anonymous Attesta-
tion [231], or DAA. With DAA, the verifier still talks with the host, but instead of
providing an attestation of the system, the verifier and host use zero-knowledge proofs
to validate the integrity of the host.
3.3 Other Document Integrity Systems

Others have looked at the problem of ensuring content integrity. We look at several of these systems here.

The WebALPS project [232, 233] examines the problem of ensuring that a compromised web server cannot subvert client requests. The implementation relies on the IBM 4758 to provide a secure co-processing environment for the web server. When a request arrives at the web server, it is encrypted, such that only the secure co-processor can decrypt the request. Once the request is decrypted by the co-processor, it is stripped of sensitive information and passed back to the web server. The web server processes the request, obtaining encrypted data to be sent back to the client. When the client obtains the response it is decrypted.

The WebALPS project differs from the systems in this thesis in several ways. First, the use of the IBM 4758 is ruled out for our systems, as we are developing solutions that are lower cost, and more readily available. Currently, the IBM 4758 cost exceeds $10,000 per card, making it impractical for large scale deployments. Second, the WebALPS system does not provide evidence to the client that the server is operating as expected. Instead, the WebALPS project ensures that a misbehaving server cannot read sensitive information, protecting the user from leaking important data.

Several other projects have examined the problem of protecting content as it is transmitted from the server to the client. In addition to SSL splitting [148], described in the previous chapter, the SINE project also provides a system that allows content to be cached, while still allowing clients to verify the integrity of the content. SINE works by creating an Authentication Tag that is stored with the content. When the client obtains the content, even from a proxy, the Authentication Tag allows the client to ensure that the content has not been modified since being sent from the server. Web tripwires [234] are another approach for ensuring unmodified content during transmission. The tripwires work by sending a canonical representation of the content to the client that is compared to the obtained content. If the two differ, the server and client are both notified.

Both the SINE and tripwire projects assume that the server itself is trustworthy, instead addressing the problem of ensuring that content served over insecure HTTP connections are not modified in transit to the client. This differs from the systems in this thesis, which looks at the problem of providing evidence to clients that the server is trustworthy.

Unlike the previous, client-side solutions, DSSA [235] provides content checking on the server. The DSSA system runs a daemon on the server that continuously verifies
the integrity of the content before sending it to the client. This allows the web server to ensure that any malicious modifications to the system are detected and prevents the content from being sent to the client. To prevent unauthorized modification of content, the integrity verification information used to ensure the content is unmodified is stored in a write-protect location, ideally on another server to prevent modification.

In addition to web content integrity, others have looked at proving system integrity in other types of networks, such as peer-to-peer networks [236]. In these systems, there are no central servers, and each client connects directly to other clients to exchange information. When a client connects to obtain some data, he trusts that the other client is trustworthy, but has no way of verifying that the remote client is indeed trustworthy. In [236], the authors propose the Trusted Reference Monitor, TRM, to ensure that all clients in the P2P network are high integrity. The TRM architecture relies on the TPM and a secure kernel. In their prototype, the authors propose the use of Microsoft’s Next Generation Secure Computing Base, NGSCB, to provide the secure kernel required to implement the TRM.

All of the above solutions help to ensure that content obtained by the client is not maliciously modified. Some assume a trustworthy server, while others take steps to protect the server from being compromised, and in case of compromise allow early detection and mitigation. However, none of the above solutions provide evidence to the client that the server is operating as expected, or if such evidence is made available, it is not bound to the content being served.

In the next several chapters, we describe solutions for building document integrity systems using the TPM and IMA that satisfy the guarantees detailed at the beginning of this chapter. These systems will satisfy different performance requirements, such as high throughput, provided by Spork in Chapter 4, or low latency, provided by Sporf in Chapter 5. Finally, Splayd, described in Chapter 6 presents a framework for building generalized document integrity systems that are not specific to web servers.
In this chapter, 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.

4.1 System Overview

An overview of the basic system architecture is shown in Figure 4.1. The core elements of the system are a) a web server that generates static or dynamic web content and provides clients with content integrity proofs, b) a time server that supplies the web server with an attestation of the current time, providing bounds on when the web server’s attestations were generated and c) a web browser 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
Figure 4.1. 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.

\[
\text{Web Server} \quad \text{Client} \quad \text{Time Service} \\
\text{Apache} \quad \text{Spork Daemon} \quad \\
\text{TPM} \quad \text{TPM} \\
\text{Request Page} \quad \text{Request Time} \quad \text{Request Time}
\]

Figure 4.2. A content proof construction that ties content to both the originating host and the time.

\[
\text{web server quote (content proof + time server quote)} \quad \text{time server quote} \quad \text{time}
\]

with a cryptographic proof system similar to an authenticated dictionary [237]. 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.

Next, we describe 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.

### 4.2 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 came from 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 c) that the integrity state of the systems is reported at the time the page was generated/served. For ease of exposition, we begin with a simple proof and build toward more semantically rich and efficient constructions that provide these properties.

First, let us introduce the notation used throughout. The function $h(d)$ denotes a cryptographic hash over some data $d$, and concatenation of different data elements is denoted as $|$. The quoting hosts are denoted $H_w$ for the web server and $H_{TS}$ for the time server. $pcr_i$ denotes the integrity state of host $i$. A TPM quote is denoted $Quote(H, s, c)$, where $H$ is the host identity performing the quote, $s$ is the PCR state, and $c$ is the quote challenge.\footnote{In practice, the quote mechanism uses \textit{attestation identity key} (or simply the \textit{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.} 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, $\text{CPS}_r$ 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 \(a\) (web server identity) and \(b\) (integrity state) from above are provided. What is missing from the simple proof is \(c\) (the element of time), 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.

Figure 4.2 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 [238]), 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 4.4.2.

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 [239]. 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 \(\approx 1\) request per second. What is 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 [237, 240], we concentrate on a conceptually simple proof system based on Merkle hash trees [241]. 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
### Figure 4.4

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

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{web server quote (content proof + time server quote)} & \text{proof} & \text{time server quote} & \text{sys. root} & \text{page} & \text{time} & \text{proof} \\
\text{Quote}(H_w, pcr_{H_w}, h(CPS_p)) & \text{Quote}(H_T S, pcr_{H_T S}, h(t_i))) & CPS_p & \text{Quote}(H_T S, pcr_{H_T S}, h(t_i)) & Pf(p_i) & t_i
\end{array}
\]
the next section). To create the proof system for these static documents, all of the documents are arranged as an ordered sequence of pages $p_1 \ldots p_n$. As shown in Figure 4.3, 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 Figure 4.3 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 $((\log_2 n) + 1) \cdot H + S$, where $H$ and $S$ are the sizes of the hash and signature respectively.

The proof system used to generate an extended content proof for page $p_i$ is shown in Figure 4.4. 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 [242]. 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.

4.2.1 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
Figure 4.5. 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 ($CPS_r$).

Figure 4.6. 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.

design and operation of the Firefox extension in Section 4.4.4.

Determining what pages should be included in a proof system is essential to supporting the browsing community. Static web pages represent the simplest case. As illustrated in Figure 4.5, 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

Figure 4.7. 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 ($Q_1$).
page, the succinct proof is extracted from the most recent proof system completed and returned to the browser, as shown in Figure 4.6. 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.

Dynamic content presents other challenges. Centrally, the page content only becomes available after the request arrives from a client. For example, consider a PHP [27] 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 Figure 4.7, 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 operation (which is the case in our test environment) and uniform distribution of arrivals, the expected latency would be about 1350 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 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

---

2TPMs from different manufacturers have different optimizations, leading to different times of operation.
Figure 4.8. Full content proof sent to the client, including quotes from the database and time server, and the succinct page proof.

Figure 4.9. 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.

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.

4.3 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 Figure 4.9. As such the proof must cover both the web server and the database. One property still missing from the proof in Figure 4.4 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 Figure 4.10. 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. Figure 4.8 shows the full attestation that the client receives.

4.4 Spork Implementation

We have developed a version of the architecture detailed in the preceding sections that supports static, dynamic, and mixed content. Figure 4.11 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.

4.4.1 Proof-Generating Web Server

As directed by the requested URL, the Apache web server supporting Spork directs all client requests (1 in Figure 4.11) 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 Figure 4.11). Dynamic requests occur in substantially the same way except that the content is generated using the appropriate content generation code, e.g., ASP [32], 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 Figure 4.11) (i.e. sockets). The processing thread then sleeps waiting for a “proof ready” event. When the requested proof (5 in Figure 4.11) 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 Figure 4.11).

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 Figure 4.11). The remaining threads update the internal state from which the proof systems are constructed. A TPM thread schedules and executes quote operations (4 in Figure 4.11) as governed by the algorithms defined in Section 4.2.1, and separate threads similarly retrieve time attestations (3 in Figure 4.11) and database attestations (12 in Figure 4.12), 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.
Figure 4.12. 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.

Client browsers receive the content proof from the web server (6 in Figure 4.11) and acquire time attestations from the time server (7 in Figure 4.11). 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 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 4.4.4.

4.4.2 Time Server

The time service uses a hash of the current hardware timestamp as a challenge to the TPM (8 in Figure 4.11). This time attestation is provided to clients 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 [238]. This approach provides a smaller base of components that need to be trusted: the BIOS core root of trust measurement (CRTM), the TPM, and the clock.

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

### 4.4.3 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. Figure 4.12 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 Figure 4.12). 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 Figure 4.12) and hashes this as the challenge for the database system’s TPM (11 in Figure 4.12). This quote is cached to service later requests (12 in Figure 4.12).

### 4.4.4 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 [243], or SSL.
Figure 4.13. Dialog notifying user of an invalid content proof

The Firefox interface is modified as shown in Figure 4.13. In Figure 4.13, 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 [95]. This is not surprising given that \( \approx 73\% \) of SSL-protected sites generates a validation warning [17]. 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 [189] 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 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.

4.5 Evaluation

Next, 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\(^3\). We propose a number of optimizations and evaluate the performance impact.

All tests were performed on Dell PowerEdge M605 blades with 8-core 2.3GHz Dual Quad-core AMD Opteron processors, 16GB RAM, and 2x73GB SAS Drives (RAID 1). Six blades running Ubuntu 8.04.1 LTS Linux kernel version 2.6.24 were connected over a gigabit Ethernet switch on a quiescent network. One blade ran Apache web servers (one normal install and one running the integrity proof system described in the preceding sections). One blade ran the time server, and four were used for simulated clients. All experiments use the Apache 2.2.8 server with mod_python 3.3.1 modules for dynamic content generation. The Spork daemon is written in Python 2.5.2 and uses a custom TPM integration library written in C. The server and client browser extension exceeds 5,000 lines of code. All load tests were performed using the Apache JMeter benchmarking tool [244].

A recent study of web pages indicated that the average web page size is about 130KB total, with an average HTML source size of 25KB and the average non-flash object being just under 10KB [245]. More focused studies of popular websites indicate somewhat larger total sizes (≈ 300KB) [246]. The sizes of the component objects (e.g., images) in popular websites is essentially the same as reported in the broader study, with the increases in the number of embedded objects accounting for the larger total page size. Thus, we use 10KB and 25KB file sizes in all experiments.

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\(^4\), but

\(^3\)Here we use a single client, and not the full client load determined below.

\(^4\)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
necessitates significantly more clients to sustain maximal throughput. After experiment-
ing 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.

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.

Figure 4.14 shows throughput of an unaltered web server measured in requests per
second (RPS). The throughput of the 10KB static content (average 10,770 RPS) has
about 29% higher throughput than the dynamic case (average 7,600 RPS) for 10KB
retransmissions. Randomized arrivals of client proof requests will dampen oscillation.
web pages. Such throughput disparities are not atypical in web systems. The additional overheads are due to forking and using a `mod_python` interpreter. This disparity is further amplified by the static content being delivered from in-memory caches in all tests, i.e., the web server can easily hold all experimental static content in memory. The throughput of the web server serving non-integrity measured 25KB pages for dynamic content are 4,486 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 10KB and 25KB pages is essentially the same:

\[
10,770 \times 10 = 107,700 \text{KB/s} \approx 4,485 \times 25 = 112,125 \text{KB/s},
\]

where 5% more “bytes on the wire” are delivered by serving larger web pages. This slight advantage can be accounted for by overheads of processing individual requests (there is 2.5 times more per-byte HTTP protocol overhead in 10KB web pages). This indicates that the bottleneck in the static case is bandwidth. 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 Figure 4.15, the average throughput of the integrity-measured web server hovers around 1,000 RPS, a significant drop in throughput compared to the un-
altered 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 4.6 and 4.7.

Integrity-measured dynamic content shows an average throughput of 1,100 RPS in both the 10KB and 25KB cases, similar to the non-integrity 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 4.1 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 $\mu$sec 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 4.2.1), 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 4.2 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, and are not on the critical path of any content delivery. Nearly 99% of the latency involves the acquisition of the time quote and the local quote operation. The remaining operations are external to the web server processing. 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

---

5Recall 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).
<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 KB Pages</td>
<td>25 KB Pages</td>
</tr>
<tr>
<td></td>
<td>RPS</td>
<td>Min. Lat.</td>
</tr>
<tr>
<td>Base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10769</td>
<td>0.49</td>
<td>4485.5</td>
</tr>
<tr>
<td>IMA</td>
<td>1108.6</td>
<td>3.1</td>
</tr>
<tr>
<td>PRIMA</td>
<td>1232.6</td>
<td>2.9</td>
</tr>
<tr>
<td>Compressed IMA</td>
<td>1504.9</td>
<td>2.6</td>
</tr>
<tr>
<td>Compressed PRIMA</td>
<td>1557.7</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Table 4.1. Static and dynamic system measurements. Latencies are measured in milliseconds. The various forms of integrity measurement used are discussed in Section 3.2. Uncompressed and compressed versions of each system are measured.
<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Merkle Hash Tree</td>
<td>0.716 (0.08%)</td>
<td>1.9 (0.19%)</td>
</tr>
<tr>
<td>Obtain TS Quote</td>
<td>35.9 (3.68%)</td>
<td>34.9 (3.58%)</td>
</tr>
<tr>
<td>Generate Quote</td>
<td>938.4 (96.24%)</td>
<td>938.8 (96.23%)</td>
</tr>
</tbody>
</table>

**Table 4.2.** Proof creation latency measurements – latency of proof system generation measured in milliseconds. For the static content, a pool of 125 files was used.
computation bottlenecks within the content delivery process itself.

### 4.6 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) [221] 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\(^6\). 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 107KB proof and the full PRIMA reduces to 82KB. 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 25KB, respectively.

Returning to Table 4.1, 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% increased throughput, with compressed PRIMA proofs seeing a 57% 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% overhead for 10KB page and 65% in 25KB pages. This is largely due to every integrity-measured static page

\(^{6}\)Additional information about the XML structure and PRIMA can be found in the Appendices.
requiring the processing and delivery of one static and one dynamic page: one for the content and one for the proof. While compression techniques mitigate the delivery of the dynamic page, it does nothing to mitigate the computational costs of its creation. Thus, our next best hope is to alter the relationship between the number of requested pages and requested proofs.

4.7 Proof Amortization

Recall that prior studies of web pages show that an average page has one root HTML page and just over 10 static 10KB 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 $P$ can be computed in pages as:

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

where $\mu$ is the service time for a web server serving a 10KB static object and $\epsilon$ is the service time for the web server serving static (dynamic) 25KB 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 4.3 shows the expected and experimentally-measured “real” throughput of the amortized proofs. We show the parameters in terms of throughput (i.e., the inverse of the service time) for clarity, with the expected throughput computed using the measurements presented in Table 4.1. Interestingly, the model underestimates throughput.
Table 4.3. Proof Amortization Performance – the expected and measured performance of the amortized proof serving.
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% of the maximum web server.

4.8 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 4.5 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.
Table 4.4. Throughput, measured in requests per second (RPS), with the database on a separate host system. Uncompressed and compressed versions of each system are measured.

<table>
<thead>
<tr>
<th></th>
<th>10 KB Pages (RPS)</th>
<th>25 KB Pages (RPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>5887.5</td>
<td>3549.4</td>
</tr>
<tr>
<td>IMA</td>
<td>439.2</td>
<td>421.2</td>
</tr>
<tr>
<td>PRIMA</td>
<td>418.5</td>
<td>450.9</td>
</tr>
<tr>
<td>Compressed IMA</td>
<td>544.4</td>
<td>541.6</td>
</tr>
<tr>
<td>Compressed PRIMA</td>
<td>549.7</td>
<td>488.8</td>
</tr>
</tbody>
</table>

Figure 4.17. 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.

Figure 4.16 shows the throughput of the unaltered web system. With 10KB dynamic pages, the throughput averages 5,887.5 RPS, while the 25KB pages averages a throughput of 3,549.4 RPS. The reduction in throughput (≈ 20%) from previous dynamic experiments is due to the additional network overhead of contacting the database system to retrieve data to service requests.

When adding integrity proofs to each request, the throughput of the web system drops dramatically. Figure 4.17 shows the throughput of the web system using the uncompressed IMA proof. The throughput averages 430 RPS in both cases, an approximately 90% 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 180KB. By leveraging PRIMA, the proof is reduced to 148KB. Compressing the proofs leads to sizes of 54KB and 44KB, respectively. With these smaller proofs, Table 4.4 shows the throughput of the web application. Even with the smallest proofs, the overhead induced by Spork is approximately 90%.

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 4.5 shows the estimated and actual throughputs using the simple model from Section 4.7. 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%.
<table>
<thead>
<tr>
<th></th>
<th>$\mu$</th>
<th>$\epsilon$</th>
<th>Expected $\mathcal{P}$</th>
<th>Web Objects</th>
<th>Actual $\mathcal{P}$</th>
<th>Web Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline with Dynamic Root Page</td>
<td>10769</td>
<td>3549.4</td>
<td>826.2</td>
<td>9088.4</td>
<td>726.3</td>
<td>7989.1</td>
</tr>
<tr>
<td>Integ. Measured Dynamic Root (Full IMA)</td>
<td>10769</td>
<td>421.2</td>
<td>302.8</td>
<td>3330.5</td>
<td>329.7</td>
<td>3627.1</td>
</tr>
<tr>
<td>Integ. Measured Dynamic Root (Comp. PRIMA)</td>
<td>10769</td>
<td>488.8</td>
<td>336.2</td>
<td>3698.2</td>
<td>596.7</td>
<td>6563.2</td>
</tr>
</tbody>
</table>

Table 4.5. Amortized proof performance when the database is on a separate host system.
In the previous chapter, we examined a document integrity system that supported high throughput web systems, at the expense of high latency for dynamic content. In some cases, this is an acceptable trade-off. For systems where low latency responses are desired, a system like Spork is not ideal. In this chapter, we examine another document integrity proof construction that allows for low latency dynamic content responses.

5.1 Overview

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 [247] and oft-invalid certificates [17]. 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 [248, 235, 232, 233, 249] 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 [9]. The user will simply stop interacting with that site, and therefore no additional damage can be done.

In the previous chapter, we 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 chapter 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 [250], 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 the cryptographic constructions we explore to support low-latency responses.

5.2 Background

In this section, we describe AJAX web applications and how they differ from traditional web applications. We also describe off-line/on-line signature schemes, and the advantages they provide for document integrity systems.

5.2.1 AJAX Applications

Asynchronous JavaScript and XML or AJAX [251] is a new method of building interactive web applications. In the traditional web application, the user requests a page from the server. The page is sent from the server, and the client renders and consumes the
content. When the user is ready to process more data, a new request for a new page is sent, repeating the process. For AJAX applications, the user loads the initial page, and then input from the user is sent to the server *asynchronously*. When the server sends an update, the client receives the update in the background and updates the currently loaded page accordingly. By only changing the current content, and not re-rendering the entire page, the user is able to continue acting on currently loaded data, even while new data is being loaded from the server, reducing the user perceived delay significantly.

### 5.2.2 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 [252] and Rabin [253] are too slow for these types of operations. In [254], 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 constructions for off-line/on-line signature schemes proposed in [255] can be found in Appendix B.1.

### 5.3 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.

#### 5.3.1 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 $VK^{ot}$. $\sigma$ and $\pi$ represent signatures.
Figure 5.1. A document proof construction that binds the document proof to the system integrity state and a recent timestamp.
We begin with the system integrity proof, showing how to bind a single document to the system integrity state:

\[
\text{Quote}(H_s, pcr_s, h(h(VK) | h(VK^{ot}))) | VK^{ot} | \sigma | \pi | ML_s
\]

where \(\sigma\) is the signature of the one-time key generated with the many-times key, and \(\pi\) 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 Chapter 3), 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 \(c\)).

In order to show that the binding between the one-time key and the integrity state happened at a verifiably known time, we include a timestamp from a trusted time server. Figure 5.1 shows the construction that includes the timestamp from the time server. Here, in addition to the verification key, the server includes a recent timestamp from the trusted time server. This construction shows that the verification key was bound to the integrity state of the server at a verifiably known time. One limitation of binding a single key to the TPM quote, is that a single TPM quote takes approximately 900 milliseconds to generate. Next, we show how to bind multiple verification keys to a single TPM quote.

We rely on cryptographic proof systems, namely a Merkle hash tree, to bind multiple 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 5.2 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.
Figure 5.3. An example of a hash tree with four keys. The circled values are a succinct proof for key $k_1$. The client computes the hash of the provided key and then computes the values up to the root of the tree. This computed value is then compared to the provided value.

5.3.2 Latency Improvement

The construction in Figure 5.1 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.

The construction in Figure 5.2 provides a statement of the system integrity at the time the keys are generated, which occurs before the content has been generated. Figure 5.4 shows the request timeline for a single client requesting content. At time $t_{quote}$, 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 5.2, is generated at time $t_{bind}$, and sent to the
client. The proof in Figure 5.2, 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 Chapter 3, 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 $t_{quote_1}$, 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 dynamic content. 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.

5.3.3 Window of Uncertainty and Countermeasures

We define the time between $t_{quote_1}$ and $t_{bind}$, in Figure 5.4 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 (Quote2 in Figure 5.4, i.e. a Spork-integrity proof. The proof obtained is the proof described in Chapter 4, obtained by the client at t_quote2 in Figure 5.4. 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_quote2), 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_quote2. 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.

5.3.3.1 Example Applications

Below, we consider two popular applications, and how the developers would integrate Sporf into the application. We first consider the popular Gmail [256] application and also a framework for building AJAX-based instant messaging clients [257]. 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\(^1\). 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.

AJAX IM [257] 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

\(^1\)A Gmail label corresponds to an IMAP folder, except that a message can have more than one label.
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 queuing 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.

5.3.4 Optimizations

In the Spork project, a number of optimizations were applied to the system to sustain higher throughput under heavy load. Some of the same optimizations can be applied to the Sporf system, but additional optimizations can be applied, which we describe in the next section.

As in the Spork project, we use IMA to provide system integrity proofs. The use of IMA leads to very large measurement lists, as every single binary is measured and reported in the attestations. In order to compensate for the large proof size, proofs can easily be compressed before being sent to the client. Mechanisms for compressing responses already exist in the popular web servers. Sporf can leverage these existing mechanisms to provide compression of data before being sent to the clients.

With Spork, each client is assumed to be unique, meaning each client must obtain the full system integrity proof for each request. With AJAX applications, after the initial page download, requests for updates come from existing clients. Taking advantage of this fact, Sporf can tailor the system integrity proof to the client making the request, by removing information that has not changed, much like AJAX does for web pages. For example if the system has taken 140 measurements when a client connects for the first time, then the system integrity proof will contain all 140 measurements, which the client can cache. However, the next time the client makes a request, the server does not have to send the first 140 measurements, drastically reducing the amount of data being sent each time.

In addition to sending only updated information, we can leverage the pre-computation of verification keys with each client by having each client pre-fetch a proof with a pre-
determined number of verification keys. Essentially, when the client connects, instead of sending a single verification key, each client receives a set number of verification keys that are then cached by the client. When the client makes a request for updates, the server uses a signing key that corresponds to a verification key that the client already has. In doing this, the server can send the signature to the client as part of the update, instead of sending it as a separate request. This eliminates one of the round-trips for a client obtaining data, further reducing load on the server and the network.

Each of the above optimizations can be applied separately, or in combination. In the evaluation, Section 5.5, we examine the combination of optimizations to identify those optimizations that provide the lowest latency and highest throughput.

5.4 Implementation

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 4.4.2. 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 sends these to the main Sporf daemon (3 in Figure 5.5). The Sporf daemon handles signing dynamic content generated by Apache (2 in Figure 5.5), 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 5.5), 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 5.5), 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 5.5). 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.

5.5 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 trade-offs. 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).
<table>
<thead>
<tr>
<th>Scheme</th>
<th>Off-line Thpt.</th>
<th>Off-line Key Gen</th>
<th>On-line Thpt.</th>
<th>Verify Per Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHR-OTS</td>
<td>3289.474</td>
<td>2282.200</td>
<td>525.210</td>
<td>762.195</td>
</tr>
<tr>
<td>GHR-DL</td>
<td>5000.000</td>
<td>2348.200</td>
<td>525.210</td>
<td>632.911</td>
</tr>
<tr>
<td>GHR-RSA</td>
<td>1982.600</td>
<td>1982.600</td>
<td>510.204</td>
<td>634.518</td>
</tr>
<tr>
<td>GHR-DL2</td>
<td>2182.400</td>
<td>2182.400</td>
<td>512.295</td>
<td>617.284</td>
</tr>
<tr>
<td>CS-OTS</td>
<td>2282.200</td>
<td>2282.200</td>
<td>312.500</td>
<td>1336.898</td>
</tr>
<tr>
<td>CS-DL</td>
<td>2099.600</td>
<td>2099.600</td>
<td>305.250</td>
<td>925.926</td>
</tr>
<tr>
<td>CS-RSA</td>
<td>2126.000</td>
<td>2126.000</td>
<td>304.507</td>
<td>968.992</td>
</tr>
<tr>
<td>CS-DL2</td>
<td>2191.000</td>
<td>2191.000</td>
<td>309.023</td>
<td>961.538</td>
</tr>
<tr>
<td>CS-RSA2</td>
<td>1517.200</td>
<td>1517.200</td>
<td>326.371</td>
<td>1086.957</td>
</tr>
</tbody>
</table>

Table 5.1. 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 Appendix B.3.

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 [244] benchmarking tool, version 2.4.

### 5.5.1 Microbenchmarks

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 [255], and were compiled for the machines in our test environment. Table 5.1 shows throughput measurements for the off-line/on-line selected schemes from [255]. In this table, we consider only parameter combinations that give the highest throughput for on-line signing. A full table is presented in Appendix B.3. In looking at Table 5.1, we see that the on-line signing phase for some schemes is able to achieve very high throughput, 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 throughput for on-line signing. This includes the following signature schemes from Table 5.1: 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.
### Table 5.2. Macrobenchmarks of the four selected off-line/on-line signature schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Content Thpt.</th>
<th>Content Lat.</th>
<th>Proof Thpt.</th>
<th>Proof Lat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>6134.4</td>
<td>80.8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GHR-DL</td>
<td>384.2</td>
<td>358.9</td>
<td>381.1</td>
<td>316.1</td>
</tr>
<tr>
<td>GHR-DL2</td>
<td>390.7</td>
<td>558.6</td>
<td>387.6</td>
<td>256.2</td>
</tr>
<tr>
<td>CS-DL</td>
<td>270.8</td>
<td>984.1</td>
<td>266.8</td>
<td>531.7</td>
</tr>
<tr>
<td>CS-DL2</td>
<td>274.5</td>
<td>713.6</td>
<td>270.9</td>
<td>415.1</td>
</tr>
</tbody>
</table>

Table 5.2. 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 Appendix B.2.

#### 5.5.2 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 [258], 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 5.2 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 5.2 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 5.1. 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 5.7 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.

Figure 5.6 shows the distribution of latency measurements for three of the client population sizes, namely 16, 128 and 512 clients. In each case, the experienced latency is relatively stable for a given population size, showing the stability of the server over time to support varying client loads. As the number of clients increases, we see more variance in the latency. Next, we will examine the latency experienced with a Sporf-enabled server.

Figure 5.9 shows the average latency under varying client population sizes for a Sporf-enabled web server. The content-only plot (Figure 5.8) looks at the effect of Sporf on just the content retrieval. Here, the minimum latency, i.e. a single client making requests, is between 1.89 and 1.92 milliseconds, an increase of only 260 microseconds. 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
Figure 5.6. Distribution of latency measurements for varying client population sizes, including 16 clients, 128 clients, and 512 clients. Here, we see that in each case, the latency for each client is relatively stable.
Figure 5.7. 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.

Figure 5.8. Latency for a varying number of clients using the four different signature schemes to obtain content. This chart reflects only the latency for obtaining content, and not the latency for obtaining the indicated proof.
Figure 5.9. 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.
Table 5.3. 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.

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 5.2, 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 [250]. 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.

5.5.3 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 5.3 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\textsuperscript{2}. 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 [250]. The bottleneck in this case is computation, as each client is waiting for a signature of their content.

5.5.4 Adding Hardware

In supporting large client populations, the system cannot generate keys fast enough to sustain the throughput 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 [152], 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

\textsuperscript{2}The off-line/on-line signature implementations are provided by the authors of [255]. We have made no efforts to optimize the implementation of these signature schemes.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Keysize</th>
<th>Key Gen.</th>
<th>Sign</th>
<th>Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>1024</td>
<td>16.98</td>
<td>1841.62</td>
<td>32258.06</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>3.57</td>
<td>290.78</td>
<td>10204.08</td>
</tr>
<tr>
<td>DSA</td>
<td>1024</td>
<td>4.19</td>
<td>3378.38</td>
<td>2994.01</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>0.57</td>
<td>1055.97</td>
<td>922.51</td>
</tr>
</tbody>
</table>

**Table 5.4.** A benchmark of key-generation, signing and verification for RSA and DSA signature schemes. These measurements were performed using the OpenSSL library.

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 throughput for an unmodified server, or just over 6,000 requests per second.

### 5.6 Discussion

#### 5.6.1 Traditional Signature Schemes

In addition to the off-line/on-line signature schemes presented in Section 5.3, it is possible to generate a single RSA/DSA keypair that is signed by the TPM. This key is bound to the integrity state of the system, and then used to sign dynamic content as it is generated by the web server. This key could potentially be the same key as the key for SSL transactions. In Appendix B.3, Table 5.4 shows the benchmarks for RSA/DSA signature schemes, generated on the same system as described in Section 5.3. Even with signature schemes show in the table, the throughput for a single processor signing content is only 1,055 requests per second, much slower than the off-line/on-line schemes. However, this does provide additional capabilities for the web server.

#### 5.6.2 Hybrid Scheme

One limitation of the off-line/on-line signature schemes is the throughput for the off-line signing phase. When the system exhausts the key cache, requests for dynamic content see an increase in latency as they wait for new keys to be signed by the TPM. In this case, we can consider a construction that allows the system to fall back to a traditional RSA/DSA key, with lower throughput, than the off-line/on-line schemes, but higher than waiting for the TPM to sign fresh keys.
Figure 5.10. Quote construction using an RSA key to sign dynamic content. This construction binds the system’s integrity state to a recent timestamp and a freshly generated RSA key. The cryptographic proof system root is still included for static content.
Web server quote
(content proof + time quote)

Proof Dynamic Time IMA
sys. content server measurement
root proof quote lists

where:

\[ Q_{ts} = Q(H_{ts}, pcr_{ts}, h(t_i)) | t_i \]

\[ CP = \begin{cases} Pf(\pi) | \pi | \sigma | VK | VK^{ot} & \text{if signing key is off-line/on-line key} \\ VK^{RSA} | \sigma & \text{if signing key is RSA/DSA key} \\ Pf(p_i) & \text{if the proof is a post-facto verification} \end{cases} \]

Figure 5.11. Quote construction using either an off-line/on-line scheme, or a traditional RSA/DSA public-private key-pair to sign dynamic content. This construction binds the system’s integrity state to a recent timestamp, a set of off-line/on-line keys, and an RSA/DSA key-pair. The cryptographic proof system includes all of the static content, the one-time keys generated during the TPM quote window, and all dynamic content that has been generated since the previous TPM quote.
With this approach, different types of keys are included in the TPM quote, namely the single RSA/DSA key, and a batch of off-line/on-line keys. Figure 5.11 shows the quote construction for including both the RSA/DWA key and the off-line/on-line keys.

With the construction shown in Figure 5.11, the Sporf daemon can use the off-line/on-line keys when they are available, and the traditional RSA/DWA key when they are not. This will allow Sporf to continue serving content, even after the supply of off-line/on-line keys is depleted, without causing clients to be delayed by key generation and TPM quote operations. One area of future work is to examine the performance of a system that uses both off-line/on-line keys, as well as traditional keys for signing content.

5.6.3 Dynamic Tuning of Sporf

Currently, the Sporf daemon spawns a fixed number of key generators at initialization, and these process run at all times. In cases where the system is heavily loaded all the time, this is ideal, but in other cases, the server may only need to deal with “flash-crowds”, where traffic loads spike for short periods and at other times, the overall load is low. In our current system, Sporf would need to be restarted before being able to handle large demands.

In addition, it is possible to eliminate waiting for keys, by falling back to a Spork-style content signature, where the hash of the content is signed by the TPM directly. Such a system would eliminate the latency for content, but would increase the overall latency, unless this fallback was intelligently applied, i.e. the content signing process would examine the additional latency introduced based on current system conditions, and choose the lowest possible latency for each request. One area of future work is to examine techniques and algorithms for intelligently scaling the number of processes based on current loads, as well as historical load information, and current system state.

Finally, looking at the types of requests being made, the daemon can choose to use off-line/on-line signatures for requests that must be low-latency, and use Spork-style signatures for the other requests. An example of this would be using the Spork-style signature for initial page requests, saving the off-line/on-line keys for the AJAX requests. Looking at expanded workloads is an area of future work.
Splayd

In the previous chapters, we have examined the construction of two document integrity systems, namely Spork and Sporf. Spork provided high-throughput web document integrity, trading off low latency for dynamic content. In contrast, Sporf provides low latency for dynamic content, but the throughput is lower than Spork. In this chapter, we present Splayd, a framework for building document integrity systems. The Splayd framework provides the following:

1. A module that interfaces with the TPM to generate system integrity proofs from available data.

2. Integration with the Linux Integrity Measurement Architecture to gather system integrity data, which is bound to the TPM proof.

3. A module that gathers data to be directly bound to the TPM proof (i.e. a Spork-style attestation), as previously described in Chapter 4.

4. A module that indirectly binds content to the system integrity state via an asymmetric keypair (i.e. a Sporf-style attestation), as previously described in Chapter 5.

5. An HTTP interface to service client requests for proofs, to ease integration with other systems.

In the next section, we examine the construction of the framework, and what a developer must provide to support the creation of document integrity proofs.

After outlining the construction of the framework, we examine a specific use case, namely binding data provenance to system integrity. Data provenance [20] tracks the
creation and modification of data as it moves through and between systems. This record of information, which is sometimes called a *provenance chain* \[21, 22\] allows a client to ascertain the trustworthiness of the data being accessed. Provenance has been studied in a number of different systems, including databases \[259, 260\], filesystems \[261, 262, 263, 264\] and also in full end-to-end systems \[265\]. This is becoming increasingly important for web applications as more and more sensitive information is being accessed via web applications.

In the next section, we examine the construction of the framework, and how an application developer can use the framework to construct a document integrity system. Next, we present our case study which uses a provenance-aware relational database to track data integrity over time. In the case study, we examine what additional effort a developer must exert to provide a full document integrity system for their application. Finally, using the provenance-aware database, we evaluate the performance of the framework, showing that it can sustain the throughput and latency requirements of a provenance-aware relational database system.

### 6.1 Framework Design and Implementation

In this section, we describe the Splayd framework, including what a developer must provide to integrate the framework into their application. We begin by detailing the different modules and their functions. Next, we describe the code a developer must write to leverage the services of the framework, and finally, we look at the framework implementation before moving on to a case study in using the framework to build an application.

The Splayd framework allows the developer to choose the style of attestation used in their application. Specifically, a developer can provide code to generate Spork-style attestations, where the data being obtained is bound directly to the system integrity proof, as described in Chapter 4. The developer can also generate Sporf-style attestations, described in Chapter 5, where the data is bound to the system integrity state via an asymmetric keypair. Depending on the application requirements, the developer can choose the style of attestation.

Recall that a Spork-style attestation provided high throughput, but exhibited high latency for dynamic content. Spork-style attestations are suited to situations where the developer can amortize the cost of proof generation over many requests, such as fetching web pages from a web server. Sporf-style attestations, on the other hand,
Figure 6.1. Splayd framework modular design. The two interface modules are provided by an application developer. The request service module presents an HTTP interface, making integration simple for most systems.

provide low latency for single requests, such as AJAX-based web applications, where individual requests are small and frequent. Next, we look at what is provided by the Splayd framework.

TPM and Integrity Measurement Integration Figure 6.1 presents a high-level overview of the modules present in the framework. The TPM interface module gathers data from the other modules and generates TPM quotes, i.e. the system integrity state described in previous chapters. This module is designed to account for any data provided by the other two interface modules, and does not require any developer code to be fully functional. All TPM operations are carried out in this module, including gathering the most recent measurement list from kernel memory for IMA.

Spork-style attestations The Spork interface module is responsible for gathering and tracking relevant data for Spork-style attestations. This includes the set of files and data being served by the system, such as static web objects and dynamic content. In order to use the Spork interface module, the developer must write code that registers files and data to be bound to attestations. As an example, consider the Spork system in
Chapter 4, where static and dynamic content are included in the attestations. In order to track static content, the framework provides a function, `registerStaticData()`, which ensures that any registered static content is included in each attestation. A developer can write a daemon that monitors a set of files and handles the registration of files, either once at startup, or continuously during system operation.

In order to support dynamic content, which appears only in the attestation after being generated, the developer must use the `registerDynamicData()`. Unlike the static file function above, the data provided to this function will only appear in the next attestation, in order to keep the size of the hash tree from growing too large. Currently, a developer uses this function in their application code, and the framework returns a unique URL for the proof that will be generated. This URL must then be passed to the client, so the client can retrieve the proof. If the application is web based, this can be done by inserting a new HTTP header. For more details about this, refer to Chapter 4.

**Sporf-style attestations** As with the Spork-style attestations described above, a developer must provide code to generate Sporf-style attestations. This code accepts a client request, and returns the response. This response is then signed by the daemon in the framework and a unique URL returned that points to the integrity proof described in Chapter 5. In the current prototype, we have explored two methods supporting this binding operation. The first is to provide an XML-RPC based interface to the binding operation, and the second is as a Python function decorator. With the XML-RPC interface, the developer must ensure that their application code makes the correct call to the XML-RPC interface, and returns the result, i.e. the proof URL, to the client. By using the Python decorator, this happens automatically when the client places the correct decorator at the top of the function that services client requests. We describe this in more detail in the implementation description, below.

**HTTP Interface** The Spork and Sporf modules are responsible for generating proofs for client requests. In order to make integration with other systems as simple as possible, the proofs are indicated to clients via URLs. When the client requests the proof, by making a standard HTTP GET request, the request service interface sends the request to the appropriate module and returns the result to the client as a standard XML document. By using commonly available technologies, and not custom protocols, integration into other systems becomes easier.
6.1.1 Framework Implementation

The Splayd framework is currently implemented in Python, as a set of daemons handling the various tasks to generate document integrity proofs. To facilitate communication between the modules, each module presents an XML-RPC interface. This choice was made to ease implementation, and also to allow the various modules to be replaced by different implementations, without having to rewrite each module. Below, we describe each module, along with the interface that each must support.

The TPM interface module is responsible for gathering data from the other modules and communicating with the TPM. It does not have any external interface, instead it expects the other modules to present a specific interface. When the TPM is available to generate a new quote, the module communicates with the Spork and Sporf modules to gather data that will be bound to the quote. This data is then combined, i.e. by concatenating and hashing, with the timestamp from the trusted time server and a new quote is requested. When the TPM finishes generating a quote, the module sends the updated quote to the other modules.

The request interface module is the external interface to the Splayd framework. When a request for a proof arrives, this module examines the request to determine which module should process the request. By providing a routing interface, new attestation constructions can be easily added. In order to distinguish different modules, the first part of the URL will indicate which module should process the request, e.g. http://example.com/sporf/proofID would indicate that the Sporf module should process the request. This module uses HTTP as a communication mechanism, making it simple to integrate into a larger system. The communication between the request interface module and the attestation modules is done via XML-RPC.

The Spork interface module handles generation of hash trees for data that is registered with the module. When the TPM module requests the current data, it generates the dynamic tree, combines the static and dynamic tree and sends the root of the hash tree to the TPM module. When the TPM module sends a new quote, it is cached by the Spork module to service requests for proofs. Much of the code to handle caching and proof creation is described in detail in Chapter 4. The basic interface that any Spork module implements involves two functions, namely \texttt{getData()} which should return the root of the current hash tree, and \texttt{updateQuoteCache()} which accepts a TPM quote and returns nothing. The Spork interface module also includes a function to obtain a URL for a proof for static content. When dynamic content is sent to the Spork module, a URL is returned for the proof.
The Sporf interface module is responsible for generating keys that are used to sign data. This signing operation effectively binds the data to the current attestation. When the TPM module is ready to generate a quote, the module’s `getData()` function is called to obtain the hash of the key being used to sign content. In Chapter 5, a hash tree of off-line/on-line keys is used, but for Splayd, we use a single asymmetric key, e.g. an RSA key, as described in the discussion at the end of Chapter 5. The `updateQuoteCache()` function is called by the TPM module when a new quote is ready. When data needs signed, the developer calls the `signContent()` function, which signs the content, returning the signature and a URL to the full proof.

In addition to the modules described above, the Splayd framework also includes an implementation of a trusted time service, used in Chapters 4 and 5. This daemon can be used to implement a trusted time server as described in previous chapters. If the developer wishes to provide a different mechanism for timestamps, the `getTimestamp()` function in the TPM interface module must be written to fetch a trusted timestamp that is tailored to the application developer’s needs. The current implementation works with the provided trusted time service daemon. In the next section, we will look at a specific case study, where we build a system that provides integrity proofs for data in a database.

### 6.2 Case Study: Provenance

Next, we will look at a specific use of the Splayd framework. We will build a system that binds database provenance data to system integrity state. We begin with an overview of database provenance, and then detail what was done to implement the system. Finally, we evaluate the overhead of using the Splayd framework on the database system.

#### 6.2.1 Background

When considering the integrity of data in a database, which is inherently not static, a client must consider what operations have occurred on the data, leading to the data’s current state. This notion is known as provenance [20, 21]. Provenance systems are being actively studied, with provenance-aware databases [259, 260] and filesystems [261, 262, 263, 264, 22] being the most common targets for tracking provenance. Provenance is not a fundamentally new concept. Valuable art often comes with a provenance record indicating the transfer of ownership, as well as any alterations that have been done, such as cleanings and restorations. This allows a collector to be certain that the art is
legitimate, and has not been altered in such a way as to ruin the value.

Clients accessing data in a provenance-aware system can query the system to determine the provenance of the accessed data, allowing the client to determine if the data is currently in a good state. However, to date, such systems have not included evidence of the underlying systems integrity state. Without taking this into account, clients cannot be assured that the provenance information being gathered is accurate. In [266] the authors propose a system to leverage trusted computing in tracking provenance information, but fail to account for the underlying system integrity, meaning clients must blindly trust that the systems working with the data are high integrity.

For our purposes, we consider provenance to be the database queries that have altered records in some way. This includes things like SELECT, INSERT, and UPDATE queries. This information is tracked by the provenance log, along with the time the operation occurred. Other information can be tracked, depending on the needs of the system and its users. The question of exactly what to track is actively being studied.

### 6.2.2 Implementation

In order to understand the process of building a document integrity system using Splayd, we have setup a prototype provenance-aware database system using MySQL. This choice was made to ease implementation and integration with Splayd, as none of the currently available provenance-aware databases provide an API that can be accessed via Python, the language used to implement the framework. Our prototype is based on the popular MySQL database. We leverage built-in features of the database to gather relevant provenance information, and report this information to the end user.

MySQL [42], as well as other popular relational database management systems, including PostgreSQL [43] and MariaDB [267], support “triggers” for various operations. These triggers allow the system to modify queries on-the-fly, or introduce new queries that use data from the triggering query. In the case of PostgreSQL, the trigger can even use a scripting language to run some arbitrary operations. For our prototype, we use the triggers to track what operations occur for each record in the database. To do this, we setup triggers for the INSERT, DELETE, and UPDATE operations, with other operations supported based on the implementation of each database. For our purposes, these are the operations used by applications to modify and create data. Below we show an example of how we setup MySQL triggers to record operations that occur on data.

We start by presenting a very simple schema for a database table. This is shown in Listing 6.1. For this table, the last_login field is updated every time the user logs in.
The three operations listed above, **INSERT**, **DELETE**, and **UPDATE** can all be done. **INSERTs** occur when new users are added, **DELETEs** occur when users are removed, and **UPDATEs** occur when the user changes their password or logs in. The first part of the provenance tracking system is a table to store the operations that occur on each record of the **users** table. For our prototype, we mirror the schema of the table, but add two new columns.

The provenance log table schema for the table in Figure 6.1 is shown in Figure 6.2. The table has the same columns as the table it is logging queries for, but adds several new columns. The first column is the **txid** which assigns a unique ID to each query logged. The second column tracks what operation was done, such as **INSERT**, **UPDATE**, or **DELETE**. Finally, the last column tracks the time that operation occurred. This table is updated with new records each time an operation occurs, as determined by the triggers that are installed.

The triggers functionality in MySQL allows for the execution of arbitrary queries based on certain criteria. Figure 6.3 shows an example trigger for the **INSERT** operation for the **users** table. Every time an **INSERT** query is done by the database, this additional query is used to track the fact that an insertion occurred. For each operation, a trigger must be created for each table, however, this process is automate-able, allowing a database administrator to provide the schema, and get back a set of tables and triggers.
CREATE TRIGGER users_insert_trigger AFTER INSERT ON users
  FOR EACH ROW
  BEGIN
    INSERT INTO users_prov VALUES(NULL, 'INSERT', NOW(),
      NEW.userid, NEW.username);
  END

Listing 6.3. Example trigger command for the users table. Other operations are also tracked, as well as other tables. The trigger simply generates an INSERT statement that adds a record to the table of transactions when data is inserted into the users table.

```python
@sporfhook
def getProvProof(table, ids, pkey_col_name):
    conn = engine.connect()
    id_set = ' (' + ', '.join(map(str, ids)) + ' )'
    s = sqlalchemy.sql.select(' * ', '%s IN %s' % (pkey_col_name, id_set), '%s_prov' % table)
    results = conn.execute(s)
    ret = list()
    for result in results:
        ret.append(result.values())
    return ret
```

Listing 6.4. Code to integrate the provenance-aware database with Splayd. Here, the only requirement is that the code return data from the function. This data is signed by the key generated by the Splayd framework for Sporf-style attestations. The decorator at the top of the function definition, i.e. `@sporfhook` ensures that the data is signed by the key and a URL returned to the client, without requiring the developer to insert any code into their function.

that are then installed in the database. With this basic provenance system, users can obtain a report of the sequence of operations that resulted in a record existing in its current state. Other systems have looked at other ways to gather this information, and also gathering a multitude of different data about the operations that occur on a given system resulting in the current state of the data.

To integrate our provenance-aware database with Splayd, we wrote a single function that would fetch provenance data from the database. This function is shown in Listing 6.4. In order to integrate with MySQL, we use the popular SQLAlchemy [268] ORM to access the database from Python. The function accepts a list of primary keys and a table name. This information is then used to construct a query to fetch the provenance data from the database. This data is then signed by the Splayd framework and returned to the client, along with the URL for the document integrity proof.

Figure 6.2 shows the communications between the framework and the database. Request processing begins with the client requesting content from the database (1 in Fig-
Figure 6.2. A system built using the Splayd framework to bind provenance logs to system integrity proofs. The framework handles the communication with the TPM and the time server, while the developer provides the code to fetch provenance logs, i.e. the data to be bound to the proof.

Figure 6.2). Included in the response is a link to the proof, in the form of a URL. The client then requests the proof indicated in the URL, which is routed to the Sporf interface (2 in Figure 6.2). Upon receiving the request, the Sporf interface will fetch the provenance log from the database (3 in Figure 6.2). The Sporf interface sign the provenance log, binding it to the quote (4 and 5 in Figure 6.2). The full document integrity proof is then sent to the client.

While the process of generating a document integrity proof is occurring, the TPM interface is also generating new quotes. The TPM interface requests an updated timestamp from the time server (6 in Figure 6.2). The TPM Interface then requests a fresh quote from the TPM (7 in Figure 6.2). The TPM interface then updates the cached quote (8 in Figure 6.2). The time server is also generating fresh quotes by requesting quotes from the TPM (9 in Figure 6.2), and caching the most recent quote (10 in Figure 6.2).

In the next section, we examine the overhead our document integrity framework has on this simple provenance system.

6.2.3 Evaluation

In this section, we present an empirical evaluation of the database system that provides document integrity proofs using the Splayd framework. All of the tests were performed on Dell PowerEdge M605 blade servers with 2.3GHz Dual Quad-Core AMD Opteron processors. Each system had 16GB of RAM and 2x73GB SAS drives configured in RAID 0. One blade served as the trusted time service, another as the database system, and the third was used to simulate clients, using the Apache JMeter benchmarking tool [244]. The choice of a single machine for the clients is based on the design of simple web applications that move a database to a separate server in order to scale the application.
In this case, all of the connections would come from a single machine, the web server. Since one popular use of databases is to house data for web applications, we chose this setup to test our system.

We begin by measuring the throughput, in queries per second, of a MySQL database. The database was setup with four tables of varying size and complexity, taken from a prototype web application for collaborative task tracking. All tables had the triggers setup to track the \texttt{INSERT}, \texttt{UPDATE}, and \texttt{DELETE} SQL queries. The full schema can be found in Appendix C.1, including the code to setup the triggers. We begin our evaluation by determining how many concurrent connections the database can support, based on the capability of the hardware. This is done by tuning the parameters of the configuration to maximize the number of concurrent connections.

To determine the number of concurrent connections our database server could sustain without error, we configured JMeter to run \texttt{SELECT}, \texttt{INSERT} and \texttt{UPDATE} queries with random data for each table. We also tuned the MySQL configuration to take advantage of the available machine resources [269]. In each test, a variable number of records was selected, ranging from one record to 16 records. For example, five tests were run for the \texttt{SELECT} query, with 1, 2 4, 8, and 16 records selected from the database. The query was repeated as often as possible by each client for two minutes. The number of clients was varied to determine the maximal throughput without experiencing any errors.

Figure 6.3 shows the average throughput for a varying number of clients for different numbers of record for \texttt{SELECT} queries. In each test, we varied to number of clients from 10 to 2,560 clients. Here, a query consisted of retrieving the data, and the provenance records for the data. In each test, we recorded the number of queries successfully completed as well as the number of errors. In all tests, with 320 or more concurrent connections, clients start to see errors, so we selected 160 as our number of concurrent connections to use for the remaining tests. Figures 6.4 and 6.5 show the throughput for \texttt{INSERT} and \texttt{UPDATE} queries respectively.

Now that we understand the performance characteristics of the database system, we examine the effect of fetching document integrity proofs for each \texttt{SELECT} query. For these queries, clients will retrieve not only the data and the provenance proof, but will also fetch a data integrity proof, showing the current integrity of the database server, bound to the provenance proof. Since we issuing single requests to the database, we have chosen to leverage the Sporf-style attestations. In systems where multiple objects are sent to a given client, we could instead leverage Spork-style attestations to achieve even higher throughput, at the expense of longer latencies.
Figure 6.3. Sustained throughput for SELECT query with a varying number of records.
Figure 6.4. Sustained throughput for INSERT query with a varying number of records.
Figure 6.5. Sustained throughput for UPDATE query with a varying number of records.
Figure 6.6. Average steady-state throughput for continuous \texttt{SELECT} queries from each of 160 clients over a two-minute experiment. Here, each query is a \texttt{SELECT} to obtain the records from the tables, followed by a second \texttt{SELECT} to obtain the provenance log for the records.
Figure 6.7. Average steady-state throughput for continuous SELECT queries from each of 160 clients over a two-minute experiment. Here, each query is the initial SELECT to obtain the data, followed by an HTTP request to obtain the data integrity proof.

Figure 6.6 shows the average throughput over a two-minute experiment for 160 concurrent connections, with SELECT queries. The average throughput is approximately 148.8 queries per second, with a maximum of 154.7 queries per second. Here, recall that each query is fetching not only the data, but the associated provenance records as well.

Figure 6.7 shows the throughput of our system. In the tests, each query is a combination of a SQL SELECT to obtain the data, followed by the HTTP request to get the data integrity proof from the Splayd framework. Here, the average throughput for all tests was 144.6 queries per second, an average of 6.5% overhead.

The overhead experienced by the database can be viewed as a lower-bound on the experienced overhead. Many popular databases are optimized for fast reads, i.e. SELECTs. In introducing other queries, it is expected that the perceived overhead of serving SELECT queries will increase, as the system becomes loaded servicing write requests. Determining the right mix of queries is application and workload specific. In addition, complex queries and more optimized queries will drastically change the load on the server.

6.3 Discussion

In this section, we discuss some additional considerations for our system, and also indicate some areas of future work.
6.3.1 Provenance Database

Several projects [259, 260] have looked at building efficient, provenance-aware databases. In the future, we would like to explore integration with these databases, as they provide a much richer provenance proof than our prototype system. These systems are targeted at specific audiences, such as scientists storing experimental results in a database. The integration of provenance data and integrity measurement would allow the consumers of the data to validate that the provenance data they are receiving is not tampered with.

6.3.2 Different signature schemes

In this work, we rely on conceptually simple RSA signatures. In the future, we would like to explore other methods of signatures that can be computed as data is entering the database, and then bathed to provide a single signature for the client to verify. Fiat proposed batch RSA [270] which allows a single signature to be generated for multiple messages, using only a single full exponentiation, the most computationally expensive step in creating a digital signature.
Conclusions and Future Work

The web has become the *de-facto* mechanism for exchanging information today. What began as an academic project to make public information easily accessible has morphed into a huge network of services that combine and share public and private information. More and more, we are seeing private information sent to remote systems, with users blindly trusting that these systems will “do the right thing”. Even systems that don’t readily appear to be web-based are, and suffer from the vulnerabilities present in traditional web systems [271].

As end-users of web systems become aware of the risks of uploading their data, the demand to ensure the integrity of these systems will increase. Currently, users are trained to look for visual cues in the browser, and even if missing, will upload their data. Even dire warnings have proven ineffective, as users have been trained to “just click through the dialogs” to get to their information. However, as more sensitive information, such as banking and medical records become exposed, users will understand that the visual feedback they see is not enough.

In this thesis, we have laid the groundwork for developing high-integrity web infrastructure. These high-integrity web systems can provide strong evidence to end-users that the system is in a known, good, state, and that the client can trust the system. Initially with the Spork system, we explored the use of commodity hardware to support high-throughput web systems. We have shown that it is feasible to rely on slower trusted hardware, and still provide acceptable throughput, at the expense of high latency for dynamic content.

Next, we examined the use of cryptographic constructions to reduce the perceived latency for end-users with the Sporf system, allowing developers to construct highly-
responsive, high-integrity web applications that leverage popular technologies for building applications. While our software only prototype traded throughput for latency, we analytically show that a nominal hardware investment would allow such a system to scale to much higher throughput demands, while still maintaining the overall lower-latency.

Next, we examined one of the limitations of these previous systems, namely the database used to store dynamic content. Unlike static web content, the integrity of this dynamic content is not easily summarized by a hash value. Instead, clients must evaluate and understand the set of operations that resulted in the current data, much as an art collector must understand the events that have affected a piece of art they are considering buying. Previous systems that track this provenance information have made the implicit assumption that the underlying system is trustworthy. The Splayd framework allows application developers to integrate an integrity measurement solution into their application, simply by providing code to access the data being hosted by their system. We show how to integrate into one such system, namely a database that track provenance information.

While this thesis explores how to construct high-integrity web systems, there is still more work to be done. Next, we examine several recent developments that change the landscape of high-integrity web systems, as well as some challenges posed by current technologies.

7.1 Upcoming Technology

Secure Microcontrollers Secure microcontrollers, such as the ones developed by Infineon [272], allow for a higher degree of programmability by the system developer, and can be used as a “replacement” for the TPM. These controllers can be used to hold measurements and perform cryptographic operations much like the TPM, but due to the programmable nature of the controllers, they can also be used to actively gather measurements, and enforce given policies. This differs from the TPM, a passive device, that cannot interfere with the normal operation of the system\(^1\). Secure microcontrollers have been proposed as a way of building high-integrity systems, such as CoPilot [228]. The basic principles of using a secure coprocessor have also been developed [209]. Using this newly developed hardware and the previously developed solutions [209], Splayd could leverage newer secure microcontrollers as they become available.

In order for Splayd to leverage secure microcontrollers, the microcontroller would

\(^1\)Unless the normal operation of the system requires data encrypted by the TPM that cannot be decrypted.
need a mechanism to generate a “proof” of the current integrity of the system, much like the TPM quote operation. The details of what the proof must provide were outlined in Chapter 3. Briefly, the controller should accept a challenge from the verifier and produce an integrity proof that can be evaluated by the verifier to ensure the integrity of the system being queried. Using this primitive, Splayd can bind data to the proof, much like is currently done with the TPM.

TPM.next Part of the effort for the Trusted Computing Group is devoted to TPM.next [273], the “next generation” TPM specification. The current specification, version 1.2, has a number of limitations, including the algorithms and key sizes used, as well as speed and robustness of the operations being carried out. In addition, the working group is looking at how best to handle virtualized systems, much like the vTPM [274], allowing the TPM to be utilized by multiple virtual machines on a given host. Others have examined other methods for multiplexing the TPM to support virtual machine environments [275, 276]. They are also attempting to simplify management of the TPM, as much of the current nomenclature has become confusing, with a number of special purpose keys being used in very limited ways.

In addition to new specifications to handle upcoming technologies, there have been efforts to examine a more modular design to TPMs [277]. These modular designs allow the TPM to be extended with additional functionality, as well as reducing the amount of code required by the TPM, to only the code required for a specific application. Another approach to modular, extensible, TPMs is presented in [278], where computations are carried out by an attached smart card, using the TPM to ensure the integrity of the computation. Such architectures can be supported by Splayd, as they rely on similar Trusted Computing concepts, and use the same underlying technology.

7.2 Challenges

Application-level Integrity In this thesis, we have used the Linux Integrity Measurement Architecture, and its extension, the Policy-Reduced Integrity Measurement Architecture, to gather integrity information about the running system. Unfortunately, web applications provide a challenge for these types of integrity measurement systems, specifically, a web application is designed to be iterated on rapidly during development and even in production. As an example, Facebook is said to deploy new code every week, with potential updates every day if a change is deemed important enough [279].
Given the rapid change, tracking “known-good code” is not feasible. What is needed is a different approach to tracking integrity in application code. One area of future work is to examine what makes web applications different from desktop applications, and how these differences can be leveraged to track integrity in web applications.

**Privacy in Integrity Measured Systems** In addition to tracking integrity in web applications, current integrity measurement solutions gather potentially sensitive information about the current state of the system. For example, consider the Apache web server software running on our system. If the current “known-good version” is running on our system, users will be able to verify that the software is the expected software. Now, let’s assume that some vulnerability has been discovered in the current version of Apache. With systems like IMA and PRIMA, it becomes trivial for an attacker to scan systems and verify if the version of Apache running is the vulnerable version. It is not clear how current systems can achieve the ability to prove their integrity, and still prevent attackers from easily scanning for vulnerable software. This is something that requires a deep investigation of not only integrity reporting, but also integrity measurement gathering.

**Scaling Horizontally** In this thesis, scaling to multiple servers was achieved by including integrity proofs for each system in the document integrity proof. While this works well in practice for one and two server systems, as the number of servers grows, i.e. *scaling horizontally*, the size of the integrity proof will easily overshadow any content being sent. Currently, several active research projects are looking at mechanisms for verifying system integrity via a third-party verifier [280, 281]. While their target has been in verifying cloud computing environments, it is conceivable that this can be applied to large, distributed web applications running on disparate servers.

**Cloud Computing Challenges** Another assumption made by the systems presented in this thesis is access to physical hardware, specifically the Trusted Platform Module. As cloud computing becomes increasingly popular, companies are relinquishing control of physical resources to third-party services. This alters the assumptions being made, and also the ability to gather and generate integrity proofs for systems running “in the cloud”. There is a huge amount of research going on, investigating secure cloud-computing platforms, and how to gather, report, and maintain integrity in such systems.
Non-web Systems  Finally, while the web has exploded in popularity, many other systems exist for exchanging information that do not rely on web systems. In developing the Splayd framework, we focused on a database system, but many other types of systems exist, and a deeper case-study is required to understand the implications of gathering and reporting integrity information in such systems. As an example, consider peer-to-peer networks, which exchange information without a centralized server. Such systems present challenges for current integrity measurement systems, as a single unit of information, i.e. a file, may no longer come from a single host, but is comprised of chunks of data from many different hosts. In such an environment, it is infeasible to verify the integrity of each system to ensure that the data a client receives is trustworthy.

In this thesis, we have only begun to explore the construction and use of document integrity systems. We show how to construct a document integrity proof that allows a client to validate the integrity of the content, and the integrity of the system hosting the content. Initially, we focused on traditional web services, developing a system that provides high-throughput using slow, commonly available trusted hardware. Next, we explored a more dynamic environment, where low-latency is crucial. Through the use of new cryptographic constructions, we developed a system that reduces the end-user experienced latency. Finally, we expand our document integrity system to account for more than just web content, building a framework that developers can leverage in their applications to provide document integrity proofs.

As users become consciously aware of the need for system integrity in remote systems, document integrity systems such as Spork and Sporf provide solutions that developers can leverage to provide proofs of system and content integrity to end-users.
Appendix A

Spork

A.1 Attestation XML Example

```xml
<?xml version="1.0"?>
<ACA>
  <Attest_List>
    <Attestation>
      <MeasurementList>
        <Measurement num="0">
          <SHA1>43c2d4b9a0eb364cb937bdd64b4a9d8fe74e781</SHA1>
          <Name>boot_aggregate</Name>
          ...
        </Measurement>
      </MeasurementList>
      <PCRList>
        <PCR num="0">01A3AFEB0A11FB7897B01F742D343C0BD9A5B807</PCR>
        ...
      </PCRList>
      <QuoteList>
        <Quote type="sCore">
          <Nonce>CCCDA67FA1213DB748781D74BD7BEFAAD145B5D02</Nonce>
          <Composite>003FFFFFF000001E001A3AFECE1A1197897B01F742D343C0BD9A5B8075B93</Composite>
          ...
          FFFFFFF00000000000000000000000000000000000000000000
          <Signature>5A9D7D0EBE8E15C563DD64DBF06C20C128BD79158D0C5BE493DD72E316F9</Signature>
          ...
          C35CABA587A4A45</Signature>
        </Quote>
      </QuoteList>
    </Attestation>
  </Attest_List>
</ACA>
```
Listing A.1. Example XML Attestation as seen by the client

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

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

A.2 Reducing Attestation Sizes

As described in Section 4.6, measurement attestations account for most of the size of the proof returned to the client, and can be as large as the document being served to the client. Reducing the size of these attestations can dramatically improve server throughput. Using the Policy-Reduced Integrity Measurement Architecture (PRIMA) [221] extensions to the Linux IMA can provide for smaller attestations by reducing the size of the measurement list to include only the specific applications of interest, under the assumption that the rest of the system is running in accordance to a mandatory policy. Central to the operation of PRIMA is the concept that information flow policies are in place to reduce the size of the measurement lists to only elements relied upon by the
Using PRIMA requires additional mechanisms above those necessary for the Linux IMA. Mandatory access control (MAC) mechanisms must be in place on the system to be attested (the attesting system). The system collects a list of trusted subjects, which must be trusted by the party verifying the system to ensure system integrity. These trusted subjects are collected in a measurement list. The code and data used by trusted subjects is also collected and measured in a list. The MAC policy itself is also measured by PRIMA to ensure correctness of the system and its information flow properties. Finally, programs acting as trusted subjects must support filtering interfaces, which can discard low-integrity inputs or upgrade them to a higher level, and the remote party must be aware of the filtering subject, which are the only entities capable of receiving low-integrity input. We have defined a MAC policy for server deployment on SELinux, with the types used for our PRIMA policy listed in Table A.1. We started with the list of trusted subjects discovered with the Gokyo policy analysis tool [282], and by manual inspection added types to the list in order to ensure that the code needed by the web server and time server was measured.

<table>
<thead>
<tr>
<th>anaconda_t</th>
<th>auditctl_t</th>
<th>dpkg_script_t</th>
<th>dpkg_t</th>
<th>depmod_t</th>
<th>firstboot_t</th>
<th>getty_t</th>
<th>ifconfig_t</th>
<th>inetd_t</th>
<th>initrc_t</th>
<th>init_t</th>
<th>insmod_t</th>
<th>kernel_t</th>
<th>ldconfig_t</th>
<th>load_policy_t</th>
<th>local_login_t</th>
<th>logrotate_t</th>
<th>mount_t</th>
<th>postfix_master_t</th>
<th>process_unconfined_exempt</th>
<th>restorecond_t</th>
<th>rpm_script_t</th>
<th>rpm_t</th>
<th>unconfined_crontab_t</th>
<th>setfiles_t</th>
<th>sshd_t</th>
<th>staff_t</th>
<th>unconfined_execmem_t</th>
<th>semanage_t</th>
<th>sysadm_t</th>
<th>unconfined_t</th>
<th>unconfined_mount_t</th>
</tr>
</thead>
</table>

Table A.1. The set of trusted subjects
Appendix B

Sporf

B.1 Off-line/on-line Signature Constructions

Note that this information is found in [255] and is here simply as reference for the reader.

In [255], the authors present ten different signature schemes. These signature schemes are based on two different many-times signature schemes, and three different one-time signature schemes. Below, we present the five constructions, as well as how to construct full off-line/on-line schemes from these building blocks.

B.1.1 Many-times Signature Schemes

The many-times schemes are due to Gennaro et al. [1] and Cramer and Shoup [2]. Both schemes are based on the Strong RSA Assumption. Below we describe the three algorithms for each construction.

B.1.1.1 GHR Signature

Key generation Let \( N = pq \) be an RSA modulus with \( p \) and \( q \) safe primes of identical size. Select a random element \( s \) in \( \mathbb{Z}_N^* \) and key \( k \) for the hash function \( h_{tdi}(\cdot, \cdot) \). The public key is \((N, s, k)\) and the secret key is \( \phi(N) = (p - 1)(q - 1) \).

Signature algorithm To sign a message, \( m \), compute \( e = h_{tdi}(k, m) \) and \( d = e^{-1} \) mod \( \phi(N) \). Output signature \( \sigma = s^d \) mod \( N \).

Verification algorithm To verify the signature, \( \sigma \), of message, \( m \), using the public key \((N, s, k)\), compute \( e = h_{tdi}(k, m) \) and check if \( \sigma^e = s \) mod \( N \).

\(^1\)For a complete definition of the function \( h_{tdi}(\cdot, \cdot) \) used in the implementations see [255].
B.1.1.2 CS Signature

Key generation Let \( N = pq \) be an RSA modulus with \( p \) and \( q \) safe primes of identical size. Select two random elements \( s, t \) from \( \mathbb{Z}_N^* \) and a random key, \( k \) for the TCR hash function\(^2\). The public key is \((N, s, t, k)\) and the secret key is \( \phi(N) = (p - 1)(q - 1) \).

Signature algorithm To sign a message, \( m \), compute a random 161-bit prime \( e \) and compute \( d = e^{-1} \mod \phi(N) \) and \( \sigma = (st^{h_{tcr}(k,m)})^d \mod N \). The signature is \((e, \sigma)\).

Verification algorithm To verify the signature \((e, \sigma)\) of message, \( m \) using public key \((N, s, t, k)\), check if \( \sigma^e = st^{h_{tcr}(k,m)} \mod N \).

B.1.2 One-time Signature Schemes

In [255], the authors present previous constructions for one-time signatures due to Lamport [283], and also present constructions for one-time signatures based on chameleon hash functions. The use of the Lamport signature in the off-line/on-line signature scheme is due to Even et al. [254], while the use of chameleon hashes to construct such schemes is due to [284]. Below, we provide details for the Lamport signature scheme as well as the chameleon hashes based on the discrete log and RSA problems.

B.1.2.1 Lamport Signature

Let \( f: \{0, 1\}^l \leftarrow \{0, 1\}^l \) be a one-way hash function, and \( M \) be an \( m \)-bit message to be signed. The signer randomly chooses \( 2m \) \( l \)-bit strings, \( x_0^1, x_1^1, x_0^2, x_1^2, \ldots, x_0^m, x_1^m \). These strings are the signing key for a single message. The verification key is generated by applying \( f \) to each of the \( x_i^0, x_i^1 \) for \( i = 1 \ldots m \), i.e. \( f(x_0^0), f(x_1^0), \ldots, f(x_0^m), f(x_1^m) \). To sign the message, \( M = b_1 \ldots b_m \), the signer reveals \( x_1^{b_1} \ldots x_m^{b_m} \). The verifier applies \( f \) to the signature, \( s = s_1, s_2, \ldots, s_m \) and compares these to the images in the verification key.

\(^2\)The TCR hash function used in the implementation is defined as \( h_{tcr}(k, x) = Trunc_l(SHA-1(x \oplus k')) \) where \( k' \) is the concatenation of copies of \( k \) such that \( k' \) and \( x \) have the same length. The \( Trunc_l \) function returns the first \( l \) bits of its input. For a full definition and properties of TCR hash functions, see [255].
B.1.2.2 Chameleon Hash Functions

Chameleon hash functions are sets of algorithms (KG, C_{pk}, Coll), where KG is the key generation function, C_{pk} is the commitment algorithm, and Coll is the collision finding algorithm.

B.1.2.3 Discrete Log Based Chameleon Hash Function

Key generation Choose g at random from group G, where G is a group of prime order \( q \). Compute \( h = g^x \mod p \), with \( x \) chosen at random from \( \mathbb{Z}_q \). The public key is \( pk = (g, h) \) and the secret key is \( x \).

Commitment Given a message, \( m \) and nonce \( r \) in \( \mathbb{Z}_q \), compute \( C_{pk}(m, r) = g^{mh^r} \).

Collision Finding Given the commitment, \( c = g^{mh^r} \), the message \( m \), nonce, \( r \), trapdoor key \( x \), and new message, \( m' \), compute \( r' = r + (m - m')x^{-1} \mod q \).

B.1.2.4 RSA Based Chameleon Hash Function

Key generation Choose a prime, \( e \), relatively prime to \( \phi(N) = (p - 1)(q - 1) \), where \( N = pq \), the product of two large primes, \( p \) and \( q \), and a random element \( s \) from \( \mathbb{Z}_N^* \). Compute \( d = e^{-1} \mod \phi(N) \). The public key, \( pk = (N, s, e) \) and the secret key, \( sk = \sigma = sd \mod N \).

Commitment Given a message, \( m \in [1 \ldots e - 1] \), and a random element, \( r \) in \( \mathbb{Z}_N^* \), compute \( C_{pk}(m, r) = s^{M_r e} \mod N \).

Collision Finding Given a commitment \( c = s^{M_r e} \), message \( m \), nonce, \( r \), key \( \sigma \), and new message, \( m' \), compute \( r' = r\sigma^{m-m'} \mod N \).

To see how to build one-time signature schemes from the chameleon hash functions readers are referred to [255].

B.1.3 Off-line/on-line Signature Schemes

As stated above, in [254], the authors present a construction for off-line/on-line signature schemes using one-time signatures, while [284] presents a construction based on chameleon hash functions. Below we provide details for both constructions. Each scheme has four algorithms, instead of the traditional three, due to the two-phase signing.
B.1.3.1 Using One-time Signatures

Key generation Key generation is done by running the key generation algorithm, Gen, to generate a key-pair, $(SK, VK)$.

Off-line signature In the off-line signing phase, the one-time key generation algorithm is run to generate a fresh key-pair, $(SK^{ot}, VK^{ot})$. Then, the signing algorithm, $Sign$, is run, with the one-time verification key, $VK^{ot}$, as the message to be signed to generate a signature, $\sigma$, i.e. $\sigma = Sign(SK, VK)$. This signature is then stored with the one-time key-pair to be used in the on-line step at a later time.

On-line signature When a message, $M$, needs signed, one of the signed one-time keys is retrieved, and the one-time signing algorithm, $Sign^{ot}$, is run, signing the message, and generating the signature, $\pi$, i.e. $\pi = Sign^{ot}(SK^{ot}, M)$.

Verification To verify the signature, the verifier runs the one-time verification algorithm, $Ver^{ot}(VK^{ot}, \pi, M)$, to verify $\pi$ and then runs the verification algorithm, $Ver(VK, \sigma, VK^{ot})$ to validate $\sigma$. The message is considered valid if both verification algorithms return true.

B.1.3.2 Using Chameleon Hash Functions

Key generation Key generation is done by generating a key pair using the signature scheme, and also running the trapdoor key generation algorithm. The public key is $(VK, pk)$ and the secret key is $(SK, tk)$.

Off-line Signature An arbitrary message, $m'$, and random nonce, $r'$ are chosen and the commitment algorithm is run to compute $\delta = C_{pk}(m', r')$. This value is then signed using $SK$, i.e. $\sigma = Sign(SK, \delta)$. The tuple $(m', r', \delta)$ is then stored until a message needs signed in the on-line phase.

On-line Signature To sign a message, $m$, a tuple is retrieved, $(m', r', \delta)$, and the value, $r$ is computed using the collision finding algorithm, i.e. $C_{pk}(m, r) = \delta = C_{pk}(m', r')$. The signature for message, $m$, is $(r, \sigma)$.

Verification To verify the signature $(r, \sigma)$, for message $m$, and verification key, $VK$, compute $\delta = C_{pk}(m, r)$ and then use the verification algorithm, i.e. $Ver(VK, C_{pk}(m, r))$. 
B.2 Chameleon Hashing

In [284], the authors propose the use of *chameleon hashes*, or *trapdoor commitment schemes*, as one-time signatures. They show how to construct off-line/on-line signatures using these trapdoor commitment schemes to perform the on-line signing. In [285], a trapdoor commitment scheme was proposed that is based on the discrete log problem. In the off-line phase, a random message, \( m \), and nonce, \( r \), are chosen and the commitment phase is run. This commitment is computed with the following formula:

\[
C_{pk}(m, r) = g^m h^r
\]

where the public key \( pk \) is \((g, h)\) and the secret key is \( x \). When the actual message, \( m' \), needs signed, in the on-line phase, the signer computes a new nonce, \( r' \), such that \( C_{pk}(m, r) = C_{pk}(m', r') \). This is done with the following computation:

\[
r' = r + (m - m')x^{-1} \mod q
\]

In the off-line phase, the value for \( x^{-1} \mod q \) is computed and stored with the key, meaning that a single multiplication is required to compute \( r' \), making the discrete log-based functions fast in the on-line phase.

B.3 Off-line/on-line Signature Benchmarks
<table>
<thead>
<tr>
<th>Scheme</th>
<th>On-line Key Parameters</th>
<th>Off-line Key Gen Time</th>
<th>Off-line Sign Time</th>
<th>On-line Sign Time</th>
<th>Verify Time</th>
<th>Sig. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHR-OTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>l=80, a=4, p=1</td>
<td>1772.400</td>
<td>1.904</td>
<td>525.210</td>
<td>0.304</td>
<td>3289.474</td>
</tr>
<tr>
<td></td>
<td>l=80, a=8, p=1</td>
<td>1957.400</td>
<td>3.724</td>
<td>267.653</td>
<td>0.624</td>
<td>1602.364</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=1</td>
<td>2138.600</td>
<td>8.272</td>
<td>120.890</td>
<td>3.388</td>
<td>295.159</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=8</td>
<td>2077.000</td>
<td>8.276</td>
<td>120.831</td>
<td>2.740</td>
<td>364.364</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=10</td>
<td>2264.400</td>
<td>8.276</td>
<td>120.831</td>
<td>4.396</td>
<td>227.480</td>
</tr>
<tr>
<td></td>
<td>l=96, a=4, p=1</td>
<td>2163.600</td>
<td>1.972</td>
<td>507.099</td>
<td>0.328</td>
<td>3048.780</td>
</tr>
<tr>
<td></td>
<td>l=96, a=8, p=1</td>
<td>2068.800</td>
<td>4.144</td>
<td>241.313</td>
<td>1.396</td>
<td>716.332</td>
</tr>
<tr>
<td></td>
<td>l=96, a=8, p=8</td>
<td>2055.000</td>
<td>4.144</td>
<td>241.313</td>
<td>0.728</td>
<td>1373.626</td>
</tr>
<tr>
<td></td>
<td>l=96, a=12, p=1</td>
<td>1737.400</td>
<td>28.068</td>
<td>35.628</td>
<td>13.024</td>
<td>76.781</td>
</tr>
<tr>
<td></td>
<td>l=96, a=12, p=6</td>
<td>1585.400</td>
<td>28.052</td>
<td>35.648</td>
<td>10.386</td>
<td>96.191</td>
</tr>
<tr>
<td></td>
<td>l=96, a=12, p=12</td>
<td>1920.300</td>
<td>28.006</td>
<td>35.592</td>
<td>5.540</td>
<td>180.505</td>
</tr>
<tr>
<td></td>
<td>l=112, a=4, p=1</td>
<td>2049.000</td>
<td>2.024</td>
<td>494.071</td>
<td>0.356</td>
<td>2808.989</td>
</tr>
<tr>
<td></td>
<td>l=112, a=8, p=1</td>
<td>1832.000</td>
<td>4.564</td>
<td>219.106</td>
<td>1.596</td>
<td>626.566</td>
</tr>
<tr>
<td></td>
<td>l=112, a=8, p=8</td>
<td>2038.400</td>
<td>4.592</td>
<td>217.770</td>
<td>0.808</td>
<td>1237.624</td>
</tr>
<tr>
<td></td>
<td>l=112, a=16, p=8</td>
<td>1996.600</td>
<td>372.224</td>
<td>2.687</td>
<td>143.416</td>
<td>6.973</td>
</tr>
<tr>
<td></td>
<td>l=112, a=16, p=16</td>
<td>2175.600</td>
<td>372.316</td>
<td>2.686</td>
<td>74.672</td>
<td>13.392</td>
</tr>
<tr>
<td>GHR-DL</td>
<td>p=1024, q=160</td>
<td>2348.200</td>
<td>1.964</td>
<td>500.165</td>
<td>0.020</td>
<td>50000.000</td>
</tr>
<tr>
<td>GHR-RSA</td>
<td>n=1024, e=160</td>
<td>1982.600</td>
<td>1.960</td>
<td>510.204</td>
<td>1.592</td>
<td>628.141</td>
</tr>
<tr>
<td>GHR-DL2</td>
<td>p=1024, q=160, tcr=80</td>
<td>2091.600</td>
<td>1.968</td>
<td>508.130</td>
<td>0.172</td>
<td>5813.953</td>
</tr>
<tr>
<td></td>
<td>p=1024, q=160, tcr=96</td>
<td>2182.400</td>
<td>1.962</td>
<td>512.295</td>
<td>0.172</td>
<td>5813.953</td>
</tr>
<tr>
<td></td>
<td>n=1024, e=160, tcr=96</td>
<td>2263.400</td>
<td>1.860</td>
<td>537.634</td>
<td>1.704</td>
<td>585.480</td>
</tr>
</tbody>
</table>

Table B.1. Off-line/on-line signature benchmarks based on the Gennaro et. al. [1] many-times signature scheme, measuring key generation, off-line signing, on-line signing and verification. The second column of the table shows the settings for various tunable parameters.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>On-line Key Parameters</th>
<th>Off-line Key Gen Time (Sec)</th>
<th>Off-line Sign Time (Sec)</th>
<th>Off-line Sign Size (Bytes)</th>
<th>On-line Sign Time (Sec)</th>
<th>On-line Sign Size (Bytes)</th>
<th>Verify Time (Sec)</th>
<th>Sign. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS-OTS</td>
<td>l=80, a=4, p=1</td>
<td>2282.200</td>
<td>3.200</td>
<td>312.500</td>
<td>0.304</td>
<td>3289.474</td>
<td>0.748</td>
<td>1336.898</td>
</tr>
<tr>
<td></td>
<td>l=80, a=8, p=1</td>
<td>2143.600</td>
<td>5.024</td>
<td>199.045</td>
<td>1.188</td>
<td>612.745</td>
<td>1.632</td>
<td>612.745</td>
</tr>
<tr>
<td></td>
<td>l=80, a=8, p=8</td>
<td>1930.800</td>
<td>5.060</td>
<td>197.628</td>
<td>0.628</td>
<td>592.937</td>
<td>1.620</td>
<td>612.745</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=1</td>
<td>2127.200</td>
<td>9.504</td>
<td>105.219</td>
<td>3.392</td>
<td>294.811</td>
<td>3.824</td>
<td>261.506</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=5</td>
<td>1962.800</td>
<td>9.552</td>
<td>104.969</td>
<td>2.744</td>
<td>644.131</td>
<td>3.824</td>
<td>261.506</td>
</tr>
<tr>
<td></td>
<td>l=80, a=10, p=10</td>
<td>2170.800</td>
<td>9.564</td>
<td>104.559</td>
<td>1.536</td>
<td>651.042</td>
<td>3.824</td>
<td>261.506</td>
</tr>
<tr>
<td></td>
<td>l=80, a=4, p=8</td>
<td>1796.600</td>
<td>3.256</td>
<td>307.125</td>
<td>0.328</td>
<td>3048.780</td>
<td>0.788</td>
<td>1269.036</td>
</tr>
<tr>
<td></td>
<td>l=80, a=8, p=8</td>
<td>1556.400</td>
<td>5.484</td>
<td>183.016</td>
<td>1.384</td>
<td>722.543</td>
<td>1.848</td>
<td>541.126</td>
</tr>
<tr>
<td></td>
<td>l=80, a=12, p=1</td>
<td>2367.600</td>
<td>29.444</td>
<td>33.963</td>
<td>13.028</td>
<td>2840.909</td>
<td>1.840</td>
<td>543.748</td>
</tr>
<tr>
<td></td>
<td>l=80, a=12, p=6</td>
<td>1820.800</td>
<td>29.392</td>
<td>34.023</td>
<td>10.444</td>
<td>1697.237</td>
<td>1.840</td>
<td>313.748</td>
</tr>
<tr>
<td></td>
<td>l=80, a=12, p=12</td>
<td>1788.000</td>
<td>29.476</td>
<td>33.926</td>
<td>5.564</td>
<td>179.727</td>
<td>1.840</td>
<td>541.126</td>
</tr>
<tr>
<td></td>
<td>l=112, a=4, p=1</td>
<td>2102.200</td>
<td>3.372</td>
<td>296.560</td>
<td>0.352</td>
<td>2840.909</td>
<td>1.840</td>
<td>543.748</td>
</tr>
<tr>
<td></td>
<td>l=112, a=8, p=1</td>
<td>1674.400</td>
<td>5.876</td>
<td>170.184</td>
<td>1.588</td>
<td>629.723</td>
<td>2.072</td>
<td>482.625</td>
</tr>
<tr>
<td></td>
<td>l=112, a=8, p=8</td>
<td>2001.000</td>
<td>5.980</td>
<td>167.224</td>
<td>0.804</td>
<td>1243.781</td>
<td>2.064</td>
<td>484.966</td>
</tr>
<tr>
<td></td>
<td>l=112, a=16, p=8</td>
<td>1739.600</td>
<td>3.372</td>
<td>396.560</td>
<td>1.588</td>
<td>629.723</td>
<td>2.072</td>
<td>482.625</td>
</tr>
<tr>
<td></td>
<td>l=112, a=16, p=16</td>
<td>1998.600</td>
<td>3.372</td>
<td>396.560</td>
<td>1.588</td>
<td>629.723</td>
<td>2.072</td>
<td>482.625</td>
</tr>
<tr>
<td>CS-DL</td>
<td>p=1024, q=160</td>
<td>2099.600</td>
<td>3.276</td>
<td>305.250</td>
<td>0.016</td>
<td>62500.000</td>
<td>1.080</td>
<td>925.926</td>
</tr>
<tr>
<td>CS-RSA</td>
<td>n=1024, e=160</td>
<td>2126.000</td>
<td>3.284</td>
<td>304.507</td>
<td>1.691</td>
<td>623.441</td>
<td>1.032</td>
<td>968.992</td>
</tr>
<tr>
<td>CS-DL2</td>
<td>p=1024, q=160, tcr=80</td>
<td>2191.000</td>
<td>3.236</td>
<td>309.023</td>
<td>0.172</td>
<td>583.953</td>
<td>1.040</td>
<td>961.538</td>
</tr>
<tr>
<td></td>
<td>p=1024, q=160, tcr=96</td>
<td>2139.600</td>
<td>3.248</td>
<td>307.882</td>
<td>0.172</td>
<td>583.953</td>
<td>1.040</td>
<td>919.118</td>
</tr>
<tr>
<td>CS-RSA2</td>
<td>n=1024, e=160, tcr=80</td>
<td>1517.200</td>
<td>3.064</td>
<td>326.371</td>
<td>1.708</td>
<td>585.480</td>
<td>0.992</td>
<td>1068.957</td>
</tr>
<tr>
<td></td>
<td>n=1024, e=160, tcr=96</td>
<td>1617.400</td>
<td>3.152</td>
<td>317.259</td>
<td>1.704</td>
<td>586.854</td>
<td>0.992</td>
<td>1068.957</td>
</tr>
</tbody>
</table>

Appendix C

Splayd

C.1 Database Schema

CREATE TABLE IF NOT EXISTS wd_users (
  userid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  username VARCHAR(128),
  password VARCHAR(40),
  email VARCHAR(128),
  firstname VARCHAR(128),
  lastname VARCHAR(128),
  ts_created INTEGER UNSIGNED,
  ts_lastlogin INTEGER UNSIGNED,
  userprivs TEXT,
  new_password VARCHAR(40),
  new_password_ts BIGINT UNSIGNED,
  new_password_key VARCHAR(40),
  defaultproject BIGINT UNSIGNED,
  active_project BIGINT UNSIGNED,
  active_list BIGINT UNSIGNED,
  event_filter VARCHAR(8),
  view_state VARCHAR(16),
  show_completed BOOLEAN DEFAULT TRUE NOT NULL,
  daily_lists BOOLEAN DEFAULT FALSE NOT NULL,
  migrate_daily BOOLEAN DEFAULT FALSE NOT NULL,
  --
  PRIMARY KEY (userid)
) ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS wd_projects (
  projectid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  projecttitle VARCHAR(128),
  userid BIGINT UNSIGNED,
  private BOOLEAN,
  --
CREATE TABLE IF NOT EXISTS `wd_lists`  
(
    listid    BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
    listtitle    VARCHAR(128),
    userid    BIGINT UNSIGNED,
    projectid    BIGINT UNSIGNED,
    deleted    BOOLEAN DEFAULT FALSE NOT NULL,
    isDaily    BOOLEAN DEFAULT FALSE,
    changeStamp    BIGINT UNSIGNED DEFAULT 0,
    
    PRIMARY KEY (listid)
) ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `wd_list_items`  
(
    itemid    BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
    listid    BIGINT UNSIGNED,
    shorttext    VARCHAR(128),
    description    TEXT,
    progress    FLOAT,
    complete    BOOLEAN,
    reminders    TEXT,
    userid    BIGINT UNSIGNED,
    priority    VARCHAR(16),
    user_assignee    BIGINT UNSIGNED,
    due_date    DATE,
    deleted    BOOLEAN DEFAULT FALSE NOT NULL,
    gcal_event_id    VARCHAR(1024) DEFAULT NULL,
    changeStamp    BIGINT UNSIGNED DEFAULT 0,
    date_completed    DATE,
    
    PRIMARY KEY (itemid)
) ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `wd_users_prov`  
(
    txid    BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
    userid    BIGINT UNSIGNED NOT NULL,
    username    VARCHAR(128),
    password    VARCHAR(40),
    email    VARCHAR(128),
    firstname    VARCHAR(128),
    lastname    VARCHAR(128),
    ts_created    INTEGER UNSIGNED,
    ts_lastlogin    INTEGER UNSIGNED,
    userprivs    TEXT,
    new_password    VARCHAR(40),
    
    PRIMARY KEY (userid)
) ENGINE = InnoDB;
```
CREATE TABLE IF NOT EXISTS wd_projects_prov (  
    txid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    projectid BIGINT UNSIGNED NOT NULL,  
    projecttitle VARCHAR(128),  
    userid BIGINT UNSIGNED,  
    private BOOLEAN,  
    deleted BOOLEAN DEFAULT FALSE NOT NULL,  
    changeStamp BIGINT UNSIGNED DEFAULT 0,  
    operation VARCHAR(20),  
    --  
    PRIMARY KEY (txid,projectid)  
) ENGINE = InnoDB;
```

```
CREATE TABLE IF NOT EXISTS wd_lists_prov (  
    txid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    listid BIGINT UNSIGNED NOT NULL,  
    listtitle VARCHAR(128),  
    userid BIGINT UNSIGNED,  
    projectid BIGINT UNSIGNED,  
    deleted BOOLEAN DEFAULT FALSE NOT NULL,  
    isDaily BOOLEAN DEFAULT FALSE,  
    changeStamp BIGINT UNSIGNED DEFAULT 0,  
    operation VARCHAR(20),  
    --  
    PRIMARY KEY (txid,listid)  
) ENGINE = InnoDB;
```

```
CREATE TABLE IF NOT EXISTS wd_list_items_prov (  
    txid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    itemid BIGINT UNSIGNED NOT NULL,  
    listid BIGINT UNSIGNED,  
    shorttext VARCHAR(128),  
    description TEXT,  
    progress FLOAT,  
    complete BOOLEAN,  
    operation VARCHAR(20),  
    --  
    PRIMARY KEY (txid,listid)  
) ENGINE = InnoDB;
```
CREATE TRIGGER wd_users_insert AFTER INSERT ON wd_users
FOR EACH ROW BEGIN
    INSERT INTO wd_users_prov VALUES (NULL, NEW.userid, NEW.username,
    NEW.password, NEW.email, NEW.firstname, NEW.lastname, NEW.ts_created,
    NEW.ts_lastlogin, NEW.userprivs, NEW.new_password, NEW.new_password_ts,
    NEW.new_password_key, NEW.default_project, NEW.active_project,
    NEW.active_list, NEW.event_filter, NEW.view_state, NEW.show_completed,
    NEW.daily_lists, NEW.migrate_daily, "INSERT");
END |

CREATE TRIGGER wd_users_update AFTER UPDATE ON wd_users
FOR EACH ROW BEGIN
    INSERT INTO wd_users_prov VALUES (NULL, NEW.userid, NEW.username,
    NEW.password, NEW.email, NEW.firstname, NEW.lastname, NEW.ts_created,
    NEW.ts_lastlogin, NEW.userprivs, NEW.new_password, NEW.new_password_ts,
    NEW.new_password_key, NEW.default_project, NEW.active_project,
    NEW.active_list, NEW.event_filter, NEW.view_state, NEW.show_completed,
    NEW.daily_lists, NEW.migrate_daily, "UPDATE");
END |

CREATE TRIGGER wd_users_delete AFTER DELETE ON wd_users
FOR EACH ROW BEGIN
    INSERT INTO wd_users_prov VALUES (NULL, OLD.userid, OLD.username,
    OLD.password, OLD.email, OLD.firstname, OLD.lastname, OLD.ts_created,
    OLD.ts_lastlogin, OLD.userprivs, OLD.new_password, OLD.new_password_ts,
    OLD.new_password_key, OLD.default_project, OLD.active_project,
    OLD.active_list, OLD.event_filter, OLD.view_state, OLD.show_completed,
    OLD.daily_lists, OLD.migrate_daily, "DELETE");
END |

CREATE TRIGGER wd_projects_insert AFTER INSERT ON wd_projects
FOR EACH ROW BEGIN
    INSERT INTO wd_projects_prov VALUES (NULL, NEW.projectid,
    NEW.projecttitle, NEW.userid, NEW.private, NEW.deleted, NEW.changeStamp,
CREATE TRIGGER wd_projects_update AFTER UPDATE ON wd_projects
    FOR EACH ROW BEGIN
        INSERT INTO wd_projects_prov VALUES (NULL, NEW.projectid,
            NEW.projecttitle, NEW.userid, NEW.private, NEW.deleted, NEW.changeStamp,
            "UPDATE");
    END

CREATE TRIGGER wd_projects_delete AFTER DELETE ON wd_projects
    FOR EACH ROW BEGIN
        INSERT INTO wd_projects_prov VALUES (NULL, OLD.projectid,
            OLD.projecttitle, OLD.userid, OLD.private, OLD.deleted, OLD.changeStamp,
            "DELETE");
    END

CREATE TRIGGER wd_lists_insert AFTER INSERT ON wd_lists
    FOR EACH ROW BEGIN
        INSERT INTO wd_lists_prov VALUES (NULL, NEW.listid, NEW.listtitle,
            NEW.userid, NEW.projectid, NEW.deleted, NEW.isDaily, NEW.changeStamp,
            "INSERT");
    END

CREATE TRIGGER wd_lists_update AFTER UPDATE ON wd_lists
    FOR EACH ROW BEGIN
        INSERT INTO wd_lists_prov VALUES (NULL, NEW.listid, NEW.listtitle,
            NEW.userid, NEW.projectid, NEW.deleted, NEW.isDaily, NEW.changeStamp,
            "UPDATE");
    END

CREATE TRIGGER wd_lists_delete AFTER DELETE ON wd_lists
    FOR EACH ROW BEGIN
        INSERT INTO wd_lists_prov VALUES (NULL, OLD.listid, OLD.listtitle,
            OLD.userid, OLD.projectid, OLD.deleted, OLD.isDaily, OLD.changeStamp,
            "DELETE");
    END

CREATE TRIGGER wd_list_items_insert AFTER INSERT ON wd_list_items
    FOR EACH ROW BEGIN
        INSERT INTO wd_list_items_prov VALUES (NULL, NEW.itemid, NEW.listid,
            NEW.shorttext, NEW.description, NEW.progress, NEW.complete,
            NEW.reminders, NEW.userid, NEW.priority, NEW.user_assignee,
            NEW.due_date, NEW.deleted, NEW.gcal_event_id, NEW.changeStamp,
            NEW.date_completed, "INSERT");
    END

CREATE TRIGGER wd_list_items_update AFTER UPDATE ON wd_list_items
    FOR EACH ROW BEGIN
        INSERT INTO wd_list_items_prov VALUES (NULL, OLD.itemid, OLD.listid,
            OLD.shorttext, OLD.description, OLD.progress, OLD.complete,
            OLD.reminders, OLD.userid, OLD.priority, OLD.user_assignee,
            OLD.due_date, OLD.deleted, OLD.gcal_event_id, OLD.changeStamp,
            OLD.date_completed, "UPDATE");
    END
FOR EACH ROW BEGIN
    INSERT INTO wd_list_items_prov VALUES(NULL, NEW.itemid, NEW.listid, NEW.shorttext, NEW.description, NEW.progress, NEW.complete,
    NEW.reminders, NEW.userid, NEW.priority, NEW.user_assignee, NEW.due_date, NEW.deleted, NEW.gcal_event_id, NEW.changeStamp,
    NEW.date_completed, "UPDATE");
END
|
CREATE TRIGGER wd_list_items_delete AFTER DELETE ON wd_list_items
FOR EACH ROW BEGIN
    INSERT INTO wd_list_items_prov VALUES(NULL, OLD.itemid, OLD.listid, OLD.shorttext, OLD.description, OLD.progress, OLD.complete,
    OLD.reminders, OLD.userid, OLD.priority, OLD.user_assignee, OLD.due_date, OLD.deleted, OLD.gcal_event_id, OLD.changeStamp,
    OLD.date_completed, "DELETE");
END
|
delimiter ;
Bibliography


[14] COMPUTERWORLD, “TJX data breach: At 45.6M card numbers, it’s the biggest ever,” http://www.computerworld.com/s/article/9014782/TJX_data_breach_At_45.6M_card_numbers_it_s_the_biggest_ever.


[16] MEMPHIS BUSINESS JOURNAL, “Mall retail co.: Attack may have compromised customer card data Read more: Mall retail co.: Attack may have compromised customer card data,” http://www.bizjournals.com/memphis/news/2010/12/10/mall-retail-co-attack-may-have.html.


“Overview of SGML Resources,” http://www.w3.org/MarkUp/SGML/.

“HTML5,” http://www.w3.org/TR/html5/.


[66] “XHTML 1.0 The Extensible HyperText Markup language,” http://www.w3.org/TR/xhtml1/.

[67] “Cascading Style Sheets, level 1,” http://www.w3.org/TR/CSS1/.


[69] “Cascading Style Sheets, Level 2 Revision 1 (CSS 2.1),” http://www.w3.org/TR/CSS21/.


Symposium on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, pp. 1–16.


142


URL http://portal.acm.org/citation.cfm?id=1863166.1863179


URL http://portal.acm.org/citation.cfm?id=1924943.1924945


URL http://doi.acm.org/10.1145/1772690.1772786


[220] “integrity: Linux Integrity Module(LIM),” http://lwn.net/Articles/287790/.


URL http://doi.acm.org/10.1145/1352592.1352625


webperformanceinc.com/library/reports/AjaxBandwidth/.

Proceedings of the 2nd conference on Theory and practice of provenance, TAPP’10,
USENIX Association, Berkeley, CA, USA, pp. 5–5.
URL http://portal.acm.org/citation.cfm?id=1855795.1855800

[260] SPILLANE, R., R. SEARS, C. YALAMANCHILI, S. GAIKWAD, M. CHINNI, and
First workshop on on Theory and practice of provenance, USENIX Association,
Berkeley, CA, USA, pp. 11:1–11:10.
URL http://portal.acm.org/citation.cfm?id=1525932.1525943

Proceedings of the 3rd conference on Hot topics in security, USENIX Association,
Berkeley, CA, USA, pp. 4:1–4:5.
URL http://portal.acm.org/citation.cfm?id=1496671.1496675

on on Theory and practice of provenance, USENIX Association, Berkeley, CA,
USA, pp. 2:1–2:5.
URL http://portal.acm.org/citation.cfm?id=1525932.1525934

preventing history forgery with secure provenance,” in Proceedings of the 7th con-
ference on File and storage technologies, USENIX Association, Berkeley, CA, USA,
pp. 1–14.
URL http://portal.acm.org/citation.cfm?id=1525908.1525909

URL http://doi.acm.org/10.1145/1629080.1629082

[265] McDANIEL, P., K. BUTLER, S. McLAUGHLIN, R. SION, E. ZADOK, and
M. WINSLETT (2010) “Towards a secure and efficient system for end-to-end prove-
nance,” in Proceedings of the 2nd conference on Theory and practice of provenance,
TAPP’10, USENIX Association, Berkeley, CA, USA, pp. 2–2.
URL http://portal.acm.org/citation.cfm?id=1855795.1855797

[266] LYLE, J. and A. MARTIN (2010) “Trusted computing and provenance: better to-
gether,” in Proceedings of the 2nd conference on Theory and practice of provenance,
TAPP’10, USENIX Association, Berkeley, CA, USA, pp. 1–1.
URL http://portal.acm.org/citation.cfm?id=1855795.1855796


Vita

Thomas Moyer

Education

The Pennsylvania State University, University Park, PA, Ph.D in Computer Science and Engineering, Fall 2011

The Pennsylvania State University, University Park, PA, M.S. in Computer Science and Engineering, Spring 2009

The Pennsylvania State University, University Park, PA, B.S. in Computer Science and Engineering, Spring 2006

Professional Experience

• Research Assistant, The Pennsylvania State University, University Park, PA, 2007 to 2011
  Developed high integrity systems for web applications, cloud computing, and storage devices.

• Instructor, The Pennsylvania State University, University Park, PA, Spring 2007
  Instructor for Introduction to Algorithmic Processes (CMPSC 101), an introductory course in computer programming.

• Summer Research Intern AT&T, Internet and Networking Systems Research Center, AT&T Labs Research, Florham Park, NJ, Summer 2007
  Assisted in developing internal tool for building and managing large-scale router configurations.

• Systems Administrator, Geodynamics Research Group, The Pennsylvania State University, University Park, PA, 2004 to 2007
  Responsible for maintaining and upgrading systems and infrastructure used for academic research projects. Built an application to assist with large-scale data processing for research projects.

Affiliations

• USENIX Advanced Computing Systems Association (USENIX)

Professional Activities

• Reviewer
  2011: IEEE Symposium on Security and Privacy (Oakland)

