SpyProxy: Execution-based Detection of Malicious Web Content

Alexander Moshchuk, Tanya Bragin, Damien Deville,
Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington
{anm, tbragin, damien, gribble, levy}@cs.washington.edu

Abstract

This paper explores the use of **execution-based** Web content analysis to protect users from Internet-borne malware. Many anti-malware tools use signatures to identify malware infections on a user's PC. In contrast, our approach is to render and observe active Web content in a disposable virtual machine **before** it reaches the user's browser, identifying and blocking pages whose behavior is suspicious. Execution-based analysis can defend against undiscovered threats and zero-day attacks. However, our approach faces challenges, such as achieving good interactive performance, and limitations, such as defending against malicious Web content that contains non-determinism.

To evaluate the potential for our execution-based technique, we designed, implemented, and measured a new proxy-based anti-malware tool called SpyProxy. SpyProxy intercepts and evaluates Web content in transit from Web servers to the browser. We present the architecture and design of our SpyProxy prototype, focusing in particular on the optimizations we developed to make on-the-fly execution-based analysis practical. We demonstrate that with careful attention to design, an execution-based proxy such as ours can be effective at detecting and blocking many of today's attacks while adding only small amounts of latency to the browsing experience. Our evaluation shows that SpyProxy detected every malware threat to which it was exposed, while adding only 600 milliseconds of latency to the start of page rendering for typical content.

1 Introduction

Web content is undergoing a significant transformation. Early Web pages contained simple, passive content, while modern Web pages are increasingly *active*, containing embedded code such as ActiveX components, JavaScript, or Flash that executes in the user's browser. Active content enables a new class of highly interactive applications, such as integrated satellite photo/mapping systems. Unfortunately, it also leads to new security threats, such as "drive-by-downloads" that exploit browser flaws to install malware on the user's PC.

This paper explores a new *execution-based* approach to combating Web-borne malware. In this approach we render and execute Web content in a disposable, isolated execution environment before it reaches the user's browser. By observing the side-effects of the execution, we can detect malicious behavior in advance in a safe environment. This technique has significant advantages: because it is based on behavior rather than signatures, it can detect threats that have not been seen previously (e.g., zero-day attacks). However, it raises several crucial questions as well. First, can execution-based analysis successfully detect today's malware threats? Second, can the analysis be performed without harming browser responsiveness? Third, what are the limitations of this approach, in particular in the face of complex, adversarial scripts that contain non-determinism?

Our goal is to demonstrate the potential for execution-based tools that protect users from malicious content as they browse the Web. To do this, we designed, prototyped, and evaluated a new anti-malware service called SpyProxy. SpyProxy is implemented as an extended Web proxy: it intercepts users' Web requests, downloads content on their behalf, and evaluates its safety before returning it to the users. If the content is unsafe, the proxy blocks it, shielding users from the threat. Our intention is not to replace other anti-malware tools, but to add a new weapon to the user's arsenal; SpyProxy is complementary to existing anti-malware solutions.

SpyProxy combines two key techniques. First, it executes Web content on-the-fly in a disposable virtual machine, identifying and blocking malware before it reaches the user's browser. In contrast, many existing tools attempt to remove malware after it is already installed. Second, it monitors the executing Web content by looking for suspicious "trigger" events (such as registry writes or process creation) that indicate potentially malicious activity [28]. Our analysis is therefore based on behavior rather than signatures.

SpyProxy can in principle function either as a service deployed in the network infrastructure or as a client-side protection tool. While each has its merits, we focus in this paper on the network service, because it is more challenging to construct efficiently. In particular, we describe a set of performance optimizations that are necessary to meet our goals.

In experiments with clients fetching malicious Web content, SpyProxy detected every threat, some of which were missed by other anti-spyware systems. Our evaluation shows that with careful implementation, the performance impact of an execution-based malware detector can be reduced to the point where it has negligible effect on a user's browsing experience. Despite the use of a "heavyweight" Internet proxy and virtual machine techniques for content checking, we introduce an average delay of only 600 milliseconds to the start of rendering in the client browser. This is small considering the amount of work performed and relative to the many seconds required to fully render a page.

The remainder of the paper proceeds as follows. Section 2 presents the architecture and implementation of SpyProxy, our prototype proxy-based malware defense system. Section 3 describes the performance optimizations that we used to achieve acceptable latency. In section 4 we evaluate the effectiveness and performance of our SpyProxy prototype. Section 5 discusses related work and we conclude in Section 6.

2 Architecture and Implementation

This section describes SpyProxy—an execution-based proxy system that protects clients from malicious, active Web objects. We begin our discussion by placing SpyProxy in the context of existing malware defenses and outlining a set of design goals. We next describe the architecture of SpyProxy and the main challenges and limitations of our approach.

2.1 Defending Against Modern Web Threats

Over the past several years, attackers have routinely exploited vulnerabilities in today's Web browsers to infect users with malicious code such as spyware. Our crawler-based study of Web content in October 2005 found that a surprisingly large fraction of pages contained drive-by-download attacks [28]. A drive-by-download attack installs spyware when a user simply visits a malicious Web page.

Many defenses have been built to address this problem, but none are perfect. For example, many users install commercial anti-spyware or anti-virus tools, which are typically signature-based. Many of these tools look only for malware that is already installed, attempting the difficult operation of removing it after the fact. Firewall-based network detectors can filter out some well-known and popular attacks, but they typically rely on static scanning to detect exploits, limiting their effectiveness. They also require deployment of hardware devices at organizational boundaries, excluding the majority of household users. Alternatively, users can examine blacklists or public warning services such as SiteAdvisor [41] or StopBadware [43] before visiting a Web site, but this can be less reliable [5, 44].

None of these defenses can stop zero-day attacks based on previously unseen threats. Furthermore, signature databases struggle to keep up with the rising number of malware variants [9]. As a result, many of today's signature-based tools fail to protect users adequately from malicious code on the Web.

2.2 Design Goals

SpyProxy is a new defense tool that is designed for today's Web threats. It strives to keep Web browsing convenient while providing on-the-fly protection from malicious Web content, including zero-day attacks. Our SpyProxy architecture has three high-level goals:

1. **Safety.** SpyProxy should protect clients from harm by preventing malicious content from reaching client browsers.

2. **Responsiveness.** The use of SpyProxy should not impair the interactive feel and responsiveness of the user's browsing experience.

3. **Transparency.** The existence and operation of SpyProxy should be relatively invisible and compatible with existing content-delivery infrastructure (both browsers and servers).

Providing safety while maintaining responsiveness is challenging. To achieve both, SpyProxy uses several content analysis techniques and performance-enhancing optimizations that we next describe.

2.3 Proxy-based Architecture

Figure 1 shows the architecture of a simplified version of SpyProxy. Key components include the *client browser*, *SpyProxy*, and remote *Web servers*. When the client browser issues a new request to a Web server, the request first flows through SpyProxy where it is checked for safety.

When a user requests a Web page, the browser software generates an HTTP request that SpyProxy must intercept. Proxies typically use one of two methods for this: browser configuration (specifying an HTTP proxy) or network-level forwarding that transparently redirects HTTP requests to a proxy. Our prototype system currently relies on manual browser configuration.

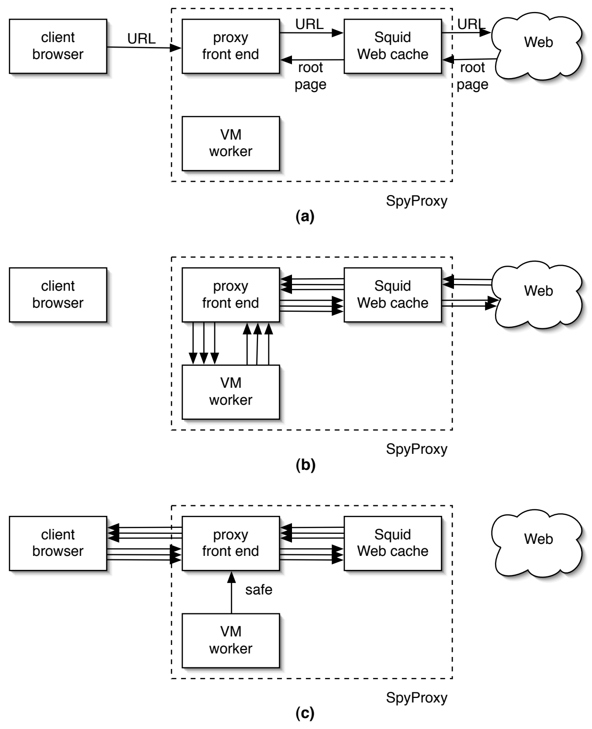

Figure 1: **SpyProxy architecture.** (a) A client browser requests a Web page; the proxy front end intercepts the request, retrieves the root page, and statically analyzes it for safety. (b) If the root page cannot be declared safe statically, the front end forwards the URL to a VM worker. A browser in the VM downloads and renders the page content. All HTTP transfers flow through the proxy front end and a Squid cache. (c) If the page is safe, the VM notifies the front end, and the page content is released to the client browser from the Squid cache. Note that if the page has been cached and was previously determined to be safe, the front end forwards it directly to the client.

The SpyProxy front end module receives clients' HTTP requests and coordinates their processing, as shown in Figure 1(a). First, it fetches the root page using a cache module (we use Squid in our prototype). If the cache misses, it fetches the data from the Web, caching it if possible and then returning it to the front end. Second, the front end statically analyzes the page (described below) to determine whether it is safe. If safe, the proxy front end releases the root page content to the client browser, and the client downloads and renders it and any associated embedded objects.

If the page cannot be declared safe statically, the front end sends the page's URL to a virtual machine (VM) worker for dynamic analysis (Figure 1(b)). The worker directs a browser running in its VM to fetch the requested URL, ultimately causing it to generate a set of HTTP requests for the root page and any embedded objects. We configure the VM's browser to route these requests first through the front end and then through the locally running Squid Web cache. Routing it through the front end facilitates optimizations that we will describe in Section 3. Routing the request through Squid lets us reduce interactions with the remote Web server.

The browser in the VM worker retrieves and renders the full Web page, including the root page and all embedded content. Once the full page has been rendered, the VM worker informs the front end as to whether it has detected suspicious activity; this is done by observing the behavior of the page during rendering, as described below. If so, the front end notifies the browser that the page is unsafe. If not, the front end releases the main Web page to the client browser, which subsequently fetches and downloads any embedded objects (Figure 1(c)).

2.3.1 Static Analysis of Web Content

On receiving content from the Internet, the SpyProxy front end first performs a rudimentary form of static analysis, as previously noted. The goal of static analysis is simple: if we can verify that a page is safe, we can pass it directly to the client without a sophisticated and costly VM-based check. If static analysis were our only checking technique, our analysis tool would need to be complex and complete. However, static analysis is just a performance optimization. Content that can be analyzed and determined to be safe is passed directly to the client; content that cannot is passed to a VM worker for additional processing.

Our static analyzer is conservative. If it cannot identify or process an object, it declares it to be potentially unsafe and submits it to a VM worker for examination. For example, our analyzer currently handles normal and chunked content encodings, but not compressed content. Future improvements to the analyzer could reduce the number of pages forwarded to the VM worker and therefore increase performance.

When the analyzer examines a Web page, it tries to determine whether the page is *active* or *passive*. Active pages include executable content, such as ActiveX, JavaScript, and other code; passive pages contain no such interpreted or executable code. Pages that contain active content must be analyzed dynamically.

It is possible for seemingly passive content to compromise the user's system if the renderer has security holes. Such flaws have occurred in the past in both the JPEG and PNG image libraries. For this reason, we consider any non-HTML content types to be unsafe and send them for dynamic processing. In principle, a browser's HTML processor could have vulnerabilities in it as well; it is possible to configure SpyProxy to disable all static checking if this is a concern.

We validated the potential benefits of static checking with a small measurement study, where we collected a

17-hour trace of Web requests generated by the user population in our department. We saw that 54.8% of HTML pages transferred contain passive content. Thus, there can be significant benefit in identifying these pages and avoiding our VM-based check for them.

2.3.2 Execution-based Analysis through VM-based Page Rendering

A drive-by download attack occurs when a Web page exploits a flaw in the victim's browser. In the worst case, an attack permits the attacker to install and run arbitrary software on the victim's computer. Our execution-based approach to detecting such attacks is adapted from a technique we developed in our earlier spyware measurement study [28], where we used virtual machines to determine whether a Web page had malicious content. We summarize this technique here.

Our detection method relies on the assumption that malicious Web content will attempt to break out of the security sandbox implemented by the browser. For example, the simple act of rendering a Web page should never cause any of the following side-effects: the creation of a new process other than known helper applications, modifications to the file system outside of safe folders such as the browser cache, registry modifications, browser or OS crashes, and so on. If we can determine that a Web page triggers any of these unacceptable conditions, we have proof that the Web page contains malicious content.

To analyze a Web page, we use a "clean" VMware [45] virtual machine configured with unnecessary services disabled. We direct an unmodified browser running in the VM to fetch and render the Web page. Because we disabled other services, any side effects we observe must be caused by the browser rendering the Web page. We monitor the guest OS and browser through "triggers" installed to look for sandbox violations, including those listed above. If a trigger fires, we declare the Web page to be unsafe. This mechanism is described in more detail in [28].

Note that this technique is behavior-based rather than signature-based. We do not attempt to characterize vulnerabilities; instead, we execute or render content to look for evidence of malicious side-effects. Accordingly, given a sufficiently comprehensive set of trigger conditions, we can detect zero-day attacks that exploit vulnerabilities that have not yet been identified.

2.4 Limitations

Our approach is effective, but has a number of challenges and limitations. First, the overhead of cloning a VM, rendering content within it, and detecting trigger conditions is potentially high. In Section 3 we describe several optimizations to eliminate or mask this overhead, and we evaluate the success of these optimizations in Section 4. Second, our trigger monitoring system should be located outside the VM rather than inside it, to prevent it from being tampered with or disabled by the malware it is attempting to detect. Though we have not done so, we believe we could modify our implementation to use techniques such as VM introspection [18] to accomplish this. Third, pre-executing Web content on-the-fly raises several correctness and completeness issues, which we discuss below.

2.4.1 Non-determinism

With SpyProxy in place, Web content is rendered twice, once in the VM's sandboxed environment and once on the client. For our technique to work, all attacks must be observed by the VM: the client must never observe an attack that the VM-based execution missed. This will be true if the Web content is deterministic and follows the same execution path in both environments. In this way, SpyProxy is ideal for deterministic Web pages that are designed to be downloaded and displayed to the user as information.

However, highly interactive Web pages resemble general-purpose programs whose execution paths depend on non-deterministic factors such as randomness, time, unique system properties, or user input. An attacker could use non-determinism to evade detection. For example, a malicious script could flip a coin to decide whether to carry out an attack; this simple scheme would defeat SpyProxy 50% of the time.

As a more pertinent example, if a Web site relies on JavaScript to control ad banner rotation, it is possible that the VM worker will see a benign ad while the client will see a malicious ad. Note, however, that much of Internet advertising today is served from ad networks such as DoubleClick or Advertising.com. In these systems, a Web page makes an image request to the server, and any non-determinism in picking an ad happens on the server side. In this case, SpyProxy will return the same ad to both the VM worker and the client. In general, only client-side non-determinism could cause problems for SpyProxy.

There are some potential solutions for handling non-determinism in SpyProxy. Similar to ReVirt [12], we could log non-deterministic events in the VM and replay them on the client; this likely would require extensive browser modifications. We could rewrite the page to make it deterministic, although a precise method for doing this is an open problem, and is unlikely to generalize across content types. The results of VM-based rendering can be shipped directly to the client using a remote display protocol, avoiding client-side rendering altogether, but this would break the integration between the user's browser and the rest of their computing environment.

None of these approaches seem simple or satisfactory; as a result, we consider malicious non-determinism to be a fundamental limitation to our approach. In our prototype, we did not attempt to solve the non-determinism problem, but rather we evaluated its practical impact on SpyProxy's effectiveness. Our results in Section 4 demonstrate that our system detected all malicious Web pages that it examined, despite the fact that the majority of them contained non-determinism. We recognize that in the future, however, an adversary could introduce non-determinism in an attempt to evade detection by SpyProxy.

2.4.2 Termination

Our technique requires that the Web page rendering process terminates so that we can decide whether to block content or forward it to the user. SpyProxy uses browser interfaces to determine when a Web page has been fully rendered. Unfortunately, for some scripts termination depends on timer mechanisms or user input, and in general, determining when or whether a program will terminate is not possible.

To prevent "timebomb-based" attacks, we speed up the virtual time in the VM [28]. If the rendering times out, SpyProxy pessimistically assumes the page has caused the browser to hang and considers it unsafe. Post-rendering events, such as those that fire because of user input, are not currently handled by SpyProxy, but could be supported with additional implementation. For example, we could keep the VM worker active after rendering and intercept the events triggered because of user input to forward them to the VM for pre-checking. The interposition could be accomplished by inserting run-time checks similar to BrowserShield [33].

2.4.3 Differences Between the Proxy and Client

In theory, the execution environment in the VM and on the client should be identical, so that Web page rendering follows the same execution path and produces the same side-effects in both executions. Differing environments might lead to false positives or false negatives.

In practice, malware usually targets a broad audience and small differences between the two environments are not likely to matter. For our system, it is sufficient that harmful side-effects produced at the client are a subset of harmful side-effects produced in the VM. This implies that the VM system can be partially patched, which makes it applicable for all clients with a higher patch level. Currently, SpyProxy uses unpatched Windows XP VMs with an unpatched IE browser. As a result, SpyProxy is conservative and will block a threat even if the client is patched to defend against it.

There is a possibility that a patch could contain a bug, causing a patched client to be vulnerable to an attack to which the unpatched SpyProxy is immune [24]. We assume this is a rare occurrence, and do not attempt to defend against it.

2.5 Client-side vs. Network Deployment

As we hinted before, SpyProxy has a flexible implementation: it can be deployed in the network infrastructure, or it can serve as a client-side proxy. There are many tradeoffs involved in picking one or the other. For example, a network deployment lets clients benefit from the workloads of other clients through caching of both data and analysis results. On the other hand, a client-side approach would remove the bottleneck of a centralized service and the latency of an extra network hop. However, clients would be responsible for running virtualization software that is necessary to support SpyProxy's VM workers. Many challenges, such as latency optimizations or non-determinism issues, apply in both scenarios.

While designing our prototype and carrying out our evaluation, we decided to focus on the network-based SpyProxy. In terms of effectiveness, the two approaches are identical, but obtaining good performance with a network deployment presents more challenges.

3 Performance Optimizations

The simple proxy architecture described in section 2 will detect and block malicious Web content effectively, but it will perform poorly. For a given Web page request, the client browser will not receive or render any content until the proxy has downloaded the full page from the remote Web server, rendered it in a VM worker, and satisfied itself that no triggers have fired. Accordingly, many of the optimizations that Web browsers perform to minimize perceived latency, such as pipelining the transfer of embedded objects and the rendering of elements within the main Web page, cannot occur.

To mitigate the cost of VM-based checking in our proxy, we implemented a set of performance optimizations that either enable the browser to perform its normal optimizations or eliminate proxy overhead altogether.

3.1 Caching the Result of Page Checks

Web page popularity is known to follow a Zipf distribution [6]. Thus, a significant fraction of requests generated by a user population are repeated requests for the same Web pages. Web proxy caches take advantage of this fact to reduce Web traffic and improve response times [1, 13, 15, 21, 52, 53]. Web caching studies generally report hit rates as high as 50%.

Given this, our first optimization is caching the *result* of our security check so that repeated visits to the same page incur the overhead of our VM-based approach only once. In principle, the hit rate in our security check cache should be similar to that of Web caches.

This basic idea faces complications. The principle of complete mediation warns against caching security checks, since changes to the underlying security policy or resources could lead to caching an incorrect outcome [34]. In our case, if any component in a Web page is dynamically generated, then different clients may be exposed to different content. However, in our architecture, our use of the Squid proxy ensures that no confusion can occur: we cache the result of a security check only for objects that Squid also caches, and we invalidate pages from the security cache if any of the page's objects is invalid in the Squid cache. Thus, we generate a hit in the security cache only if all of the Web page content will be served out of the Squid proxy cache. Caching checks for non-deterministic pages is dangerous, and we take the simple step of disabling the security cache for such pages.

3.2 Prefetching Content to the Client

In the unoptimized system shown in Figure 1, the Web client will not receive any content until the entire Web page has been downloaded, rendered, and checked by SpyProxy. As a result, the network between the client and the proxy remains idle when the page is being checked. If the client has a low-bandwidth network connection, or if the Web page contains large objects, this idle time represents a wasted opportunity to begin the long process of downloading content to the client.

To rectify this, SpyProxy contains additional components and protocols that overlap several of the steps shown in Figure 1. In particular, a new client-side component acts as a SpyProxy agent. The client-side agent both prefetches content from SpyProxy and releases it to the client browser once SpyProxy informs it that the Web page is safe. This improves performance by transmitting content to the client in parallel with checking the Web page in SpyProxy. Because we do not give any Web page content to the browser before the full page has been checked, this optimization does not erode security.

In our prototype, we implemented the client-side agent as an IE plugin. The plugin communicates with the SpyProxy front end, spooling Web page content and storing it until SpyProxy grants it authorization to release the content to the browser.

3.3 The Staged Release of Content

Although prefetching allows content to be spooled to the client while SpyProxy is performing its security check, the user's browser cannot begin rendering any of that content until the full Web page has been rendered and checked in the VM worker. This degrades responsiveness, since the client browser cannot take advantage

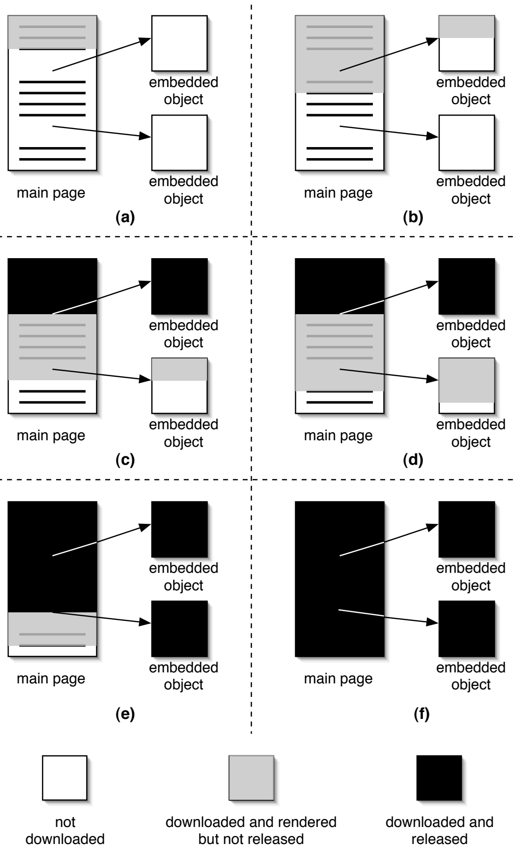

Figure 2: **Staged release optimization.** The progression of events in the VM worker's browser shows how staged release operates on a Web page with two embedded objects. As embedded objects become fully downloaded and rendered by the VM worker's browser, more of the Web page is released to the client-side browser.

of its performance optimizations that render content well before the full page has arrived.

We therefore implemented a "staged release" optimization. The goal of staged release is to present content considered safe for rendering to the client browser in pieces; as soon as the proxy believes that a slice of content (e.g., an object or portion of an HTML page) is safe, it simultaneously releases and begins transmitting that content to the client.

Figure 2 depicts the process of staged release. A page consists of a root page (typically containing HTML) and a set of embedded objects referred to from within the root page. As a Web browser downloads and renders more and more of the root page, it learns about embedded objects and begins downloading and rendering them.

Without staged release, the proxy releases no content until the full Web page and its embedded objects have been rendered in the VM. With staged release, once the VM has rendered an embedded object, it releases that object and all of the root page content that *precedes* its reference. If the client browser evaluates the root page in the same order as the VM browser, this is safe to do; our results in Section 4 confirm this optimization is safe in practice. Thus, a pipeline is established in which content is transmitted and released incrementally to the client browser.

In Figures 2(a) and 2(b), only part of the main Web page has been downloaded and rendered by the VM browser. In Figure 2(c), all of the first embedded object has been rendered by the VM, which causes that object and some of the main Web page content (shown in black) to be released and transmitted to the client browser. More of the main Web page and the second embedded object is downloaded and rendered in Figure 2(d), until finally, in Figures 2(e) and 2(f), the full Web page is released.

Many Web pages contain dozens of embedded images. For example, CNN's Web page contains over 32 embedded objects. Faced with such a Web page, our staged release optimization quickly starts feeding the client browser more and more of the root page and associated embedded objects. As a result, the user does not observe expensive Web access delay.

Note that staged release is independent from prefetching. With prefetching, content is pushed to the client-side agent before SpyProxy releases it to the client browser; however, no content is released until the full page is checked. With staged release, content is released incrementally, but released content is not prefetched. Staged release can be combined with prefetching, but since it does not require a client-side agent to function, it may be advantageous to implement staged release without prefetching. We evaluate each of these optimizations independently and in combination in Section 4.

3.4 Additional optimizations

SpyProxy contains a few additional optimizations. First, the VM worker is configured to have a browser process already running inside, ready to accept a URL to retrieve. This avoids any start-up time associated with booting the guest OS or launching the browser. Second, the virtual disk backing the VM worker is stored in a RAM-disk file system in the host OS, eliminating the disk traffic associated with storing cookies or files in the VM browser. Finally, instead of cloning a new VM worker for every client request, we re-use VM workers across requests, garbage collecting them only after a trigger fires or a configurable number of requests has occurred. Currently, we garbage collect a worker after 50 requests.

4 Evaluation

This section evaluates the effectiveness and performance of our SpyProxy architecture and prototype. The prototype includes the performance optimizations we described previously. Our results address three key questions: how effective is our system at detecting and blocking malicious Web content, how well do our performance optimizations mask latency from the user, and how well does our system perform given a realistic workload?

malicious pages visited	browser exploits	27
	spontaneous downloads	73
	total	*100*
# sites containing the malicious pages		45
malicious pages blocked by SpyProxy		**100%**
malicious domains identified by SiteAdvisor		80%
malicious pages containing non-determinism		96%

Table 1: **Effectiveness of SpyProxy.** The effectiveness of SpyProxy at detecting and blocking malicious Web content. SpyProxy was successful at detecting and blocking 100% of the malicious Web pages we visited, in spite of the fact that most of them contained non-determinism. In comparison, the SiteAdvisor service incorrectly classified 20% of the malicious Web domains as benign.

4.1 Effectiveness at Blocking Malicious Code

We first consider the ability of SpyProxy to successfully block malicious content. To quantify this, we manually gathered a list of 100 malicious Web pages on 45 distinct sites. Each of these pages performs an attack of some kind. We found these pages using a combination of techniques, including: (1) searching Google for popular Web categories such as music or games, (2) mining public blacklists of known attack sites, and (3) examining public warning services such as SiteAdvisor.

Some of the Web pages we found exploit browser vulnerabilities to install spyware. Others try to "push" malicious software at clients spontaneously, requiring user consent to install it; we have configured SpyProxy to automatically accept such prompts to evaluate its effectiveness at blocking these threats. The pages include a diversity of attack methods, such as the WMF exploit, ActiveX controls, applet-based attacks, JavaScript, and pop-up windows. A successful attack inundates the victim with adware, dialer, and Trojan downloader software.

Table 1 quantifies the effectiveness of our system. SpyProxy detected and blocked 100% of the attack pages, despite the diversity of attack methods to which it was exposed. Further, most of these attack pages contained some form of non-deterministic content; in practice, none of the attacks we found attempted to evade detection by "hiding" inside non-deterministic code.

The table also shows the advantage of our on-the-fly approach compared to a system like SiteAdvisor, which provides static recommendations based on historical evidence. SiteAdvisor misclassified 20% of the malicious sites as benign. While we cannot explain why SiteAdvisor failed on these sites, we suspect it is due to a combination of incomplete Web coverage (i.e., not having examined some pages) and stale information (i.e., a page that was benign when examined has since become malicious). SpyProxy's on-the-fly approach examines Web page content as it flows towards the user, resulting in a more complete and effective defense.

For an interesting example of how SpyProxy works in practice, consider *www.crackz.ws*, one of our 100 malicious pages. This page contains a specially crafted image that exploits a vulnerability in the Windows graphics rendering engine. The exploit runs code that silently downloads and installs a variety of malware, including several Trojan downloaders. Many signature-based anti-malware tools would not prevent this attack from succeeding; they would instead attempt to remove the malware after the exploit installs it.

In contrast, when SpyProxy renders a page from *www.crackz.ws* in a VM, it detects the exploit when the page starts performing unacceptable activity. In this case, as the image is rendered in the browser, SpyProxy detects an unauthorized creation of ten helper processes. SpyProxy subsequently blocks the page before the client renders it. Note that SpyProxy does not need to know any details of the exploit to stop it. Equally important, in spite of the fact that the exploit attacks a non-browser flaw that is buried deep in the software stack, SpyProxy's behavior-based detection allowed it to discover and prevent the attack.

4.2 Performance of the Unoptimized System

This section measures the performance of the basic *unoptimized* SpyProxy architecture we described in Section 2.3. These measurements highlight the limitations of the basic approach; namely, unoptimized SpyProxy interferes with the normal browser rendering pipeline by delaying transmission until an entire page is rendered and checked. They also suggest opportunities for optimization and provide a baseline for evaluating the effectiveness of those optimizations.

We ran a series of controlled measurements, testing SpyProxy under twelve configurations that varied across the following three dimensions:

- **Proxy configuration.** We compared a regular browser configured to communicate directly with Web servers with a browser that routes its requests through the SpyProxy checker.
- **Client-side network.** We compared a browser running behind an emulated broadband connection with a browser running on the same gigabit Ethernet LAN as SpyProxy. We used the client-side NetLimiter tool and capped the upload and download client bandwidth at 1.5 Mb/s to emulate the broadband connection.
- **Web page requested.** We measured three different Web pages: the Google home page, the front page of the *New York Times*, and the "MSN shopping insider" blog, which contains several large, embedded images. The Google page is small: just 3,166 bytes of HTML and a single 8,558 byte embedded GIF. The *New York Times* front page is larger and more complex: 92KB of HTML, 74 embedded images, 4 stylesheets, 3 XML objects, 1 flash animation, and 10 embedded JavaScript objects. This represents 844KB of data. The MSN blog consists of a 79KB root HTML page, 18 embedded images (the largest of which is 176KB), 2 stylesheets, and 1 embedded JavaScript object, for a total of 1.4MB of data.

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
direct	0.21s	0.64s	0.41s	4.8s	0.40s	10.2s
unoptimized SpyProxy	0.79s	1.2s	3.4s	7.3s	2.7s	12.4s

(a) broadband

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
direct	0.20s	0.63s	0.41s	3.3s	0.36s	2.3s
unoptimized SpyProxy	0.79s	1.2s	3.4s	5.3s	2.7s	3.9s

(b) gigabit

Table 2: **Performance of the unoptimized SpyProxy.** These tables compare the latency of an unprotected browser that downloads content directly from Web servers to that of a protected browser downloading through the SpyProxy service. We show the latency until the page begins to render on the client and the latency until the page finishes rendering. The data are shown for three Web pages as well the client on (a) an emulated broadband access link, and (b) the same LAN as SpyProxy.

For each of the twelve configurations, we created a timeline showing the latency of each step from the client's Web page request to the final page rendering in the client. We broke the end-to-end latency into several components, including WAN transfer delays, the overhead of rendering content in the VM worker before releasing it to the client, and internal communication overhead in the SpyProxy system itself. We cleared all caches in the system to ensure that content was retrieved from the original Web servers in all cases.

For each configuration, Table 2 shows the time until content first begins to render on the user's screen and the time until the Web page finishes rendering. In all cases,

time (ms)	event
0	user requests URL, browser generates HTTP request
169	SpyProxy FE receives request, requests root page from Squid
538	SpyProxy FE finishes static check, forwards URL to VM
560	VM browser generates HTTP request
561	first byte of root page arrives at VM browser
3055	last byte of last page component arrives at VM browser
3363	VM browser finishes rendering, checking triggers
3374	first byte of root page arrives at client browser
7334	last byte of last page component arrives at client browser
7347	client browser finishes rendering content

client browser transfer and render time:	4.5s
overhead introduced by VM browser:	2.8s
other SpyProxy system overhead:	0.05s

Table 3: **Detailed breakdown of the unoptimized SpyProxy.** Events occurring when fetching the New York Times page over broadband through SpyProxy. Most SpyProxy overhead is due to serializing the VM browser download and trigger checks before transferring or releasing content to the client browser.

the unoptimized SpyProxy implementation added less than three seconds to the total page download time. However, the time until rendering began was much higher on the unoptimized system, growing in some cases by a factor of ten. This confirms that our system can perform well, but, without optimizations, it interferes with the browser's ability to reduce perceived latency by pipelining the transfer and rendering of content.

Table 3 provides a more detailed timeline of events when fetching the *New York Times* page from a broadband client using the unoptimized SpyProxy. Downloading and rendering the page in the VM browser introduced 2.8 seconds of overhead. Since no data flows to the client browser until SpyProxy finishes rendering and checking content, this VM rendering latency is responsible for delay experienced by the user.

4.3 Performance Optimizations

To reduce the overhead introduced by the unoptimized SpyProxy system, we previously described three optimization techniques: prefetching content to a client-side agent, the staged release of content to the client browser, and caching the results of security checks. We now present the results of a set of microbenchmarks that evaluate the impact of each optimization.

Figure 3 summarizes the benchmark results. Both figures show the latency to download three different pages to a client on the emulated broadband connection. For each page, we show latency for five cases: (1) the unoptimized SpyProxy, (2) SpyProxy with only prefetching enabled, (3) SpyProxy with only staged release enabled, (4) SpyProxy with a hit in the enabled security cache, and (5) the base case of a client fetching content directly from Web servers. Figure 3(a) shows the latency before page rendering begins in the client browser, while Figure 3(b) shows the latency until page rendering ends.

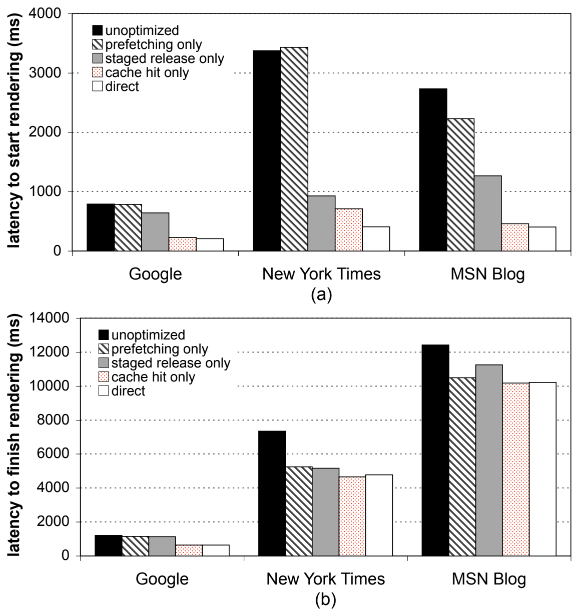

Figure 3: **Performance of optimizations (broadband).** The latency until the client browser (a) begins rendering the page, and (b) finishes rendering the page. Each graph shows the latency for three different pages for five configurations.

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
prefetching only	.78s (-0.01s)	1.15s (-0.06s)	3.43s (+0.06s)	5.2s (-2.1s)	2.2s (-0.5s)	11.3s (-1.1s)

Table 4: **Prefetching (broadband).** Latency improvements gained by the prefetching optimization in the broadband environment. Prefetching alone did not yield significant benefits.

In combination, the optimizations serve to reduce the latency before the *start* of rendering in the client. With all of the the optimizations in place, the page load "feels" nearly as responsive through SpyProxy as it does without SpyProxy. In either case, the page begins rendering about a second after the request is generated. The optimizations did somewhat improve the total rendering latency relative to the unoptimized implementation (Figure 3(b)), but this was not nearly as dramatic. Page completion time is dominated by transfer time over the broadband network, and our optimizations do nothing to reduce this.

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
staged release only	0.64s (-0.15s)	1.13s (-0.08s)	0.92s (-2.45s)	5.2s (-2.1s)	1.3s (-1.4s)	11.3s (-1.1s)

Table 5: **Staged release (broadband).** Latency improvements from staged release in the broadband environment. Staged release significantly improved the latency until rendering starts. It yielded improvements similar to prefetching in the latency until full page rendering ends.

4.3.1 Prefetching

Prefetching by itself does not yield significant benefits. As shown in Table 4, it did not reduce rendering start-time latency. With prefetching alone, the client browser effectively stalls while the VM browser downloads and renders the page fully in the proxy. That is, SpyProxy does not release content to the client's browser until the VM-based check ends.

However, we did observe some improvement in finish-time measurements. For example, the time to fully render the *New York Times* page dropped by 2.1 seconds, from 7.3 seconds in the unoptimized SpyProxy to 5.2 seconds with prefetching enabled. Prefetching successfully overlaps some transmission of content to the client-side agent with SpyProxy's security check, slightly lowering overall page load time.

4.3.2 Staged Release

Staged release very successfully reduced initial latency before rendering started; this time period has the largest impact on perceived responsiveness. As shown in Table 5, staged release reduced this latency by several seconds for both the *New York Times* and MSN blog pages. In fact, from the perspective of a user, the *New York Times* page began rendering nearly four times more quickly with staged release enabled. For all three pages, initial rendering latency was near the one-second mark, implying good responsiveness.

The staged release optimization also reduced the latency of rendering the full Web page to nearly the same point as prefetching. Even though content does not start flowing to the client until it is released, this optimization releases some content quickly, causing an overlap of transmission with checking that is similar to prefetching.

Staged release outperforms prefetching in the case that matters—initial time to rendering. It also has the advantage of not requiring a client-side agent. Once SpyProxy decides to release content, it can simply begin uploading it directly to the client browser. Prefetching requires the installation of a client-side software component, and it provides benefits above staged release only in a narrow set of circumstances (namely, pages that contain very large embedded objects).

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
security cache hit	0.23s (-0.56s)	0.64s (-0.57s)	0.71s (-2.66s)	4.6s (-2.7s)	0.5s (-2.2s)	10.2s (-2.2s)

Table 6: **Security cache hit (broadband).** This table shows the latency improvements gained when the security cache optimization is enabled and the Web page hits in the cache.

To better visualize the impact of staged release, Figure 4 depicts the sequence of Web object completion events that occur during the download and rendering of a page. Figure 4(a) shows completion events for the *New York Times* page. The unoptimized SpyProxy (top) does not transmit or release events to the client browser until the full page has rendered in the VM. With staged release (4(a) bottom), as objects are rendered and checked by the SpyProxy VM, they are released and transmitted to the client browser and then rendered. Accordingly, the sequence of completion events is pipelined between the two browsers. This leads to much more responsive rendering and an overall lower page load time.

Figure 4(b) shows a similar set of events for the MSN blog page. Since this page consists of few large embedded images, the dominant cost in both the unoptimized and staged-release-enabled SpyProxy implementations is the time to transmit the images to the client over broadband. Accordingly, though staged release permits the client browser to begin rendering more quickly, most objects queue up for transmission over the broadband link after being released by SpyProxy.

4.3.3 Caching

When a client retrieves a Web page using the optimized SpyProxy, both the outcome of the security check and the page's content are cached in the proxy. When a subsequent request arrives for the same page, if any of its components are cached and still valid, our system avoids communicating with the origin Web server. In addition, if all components of the page are cached and still valid, the system uses the previous security check results instead of incurring the cost of a VM-based evaluation.

In Table 6, we show the latency improvement of hitting in the security cache compared with the unoptimized SpyProxy. As with the other optimizations, the primary benefit of the security cache is to improve the latency until the page begins rendering. Though the full page load time improves slightly, the transfer time over the broad-

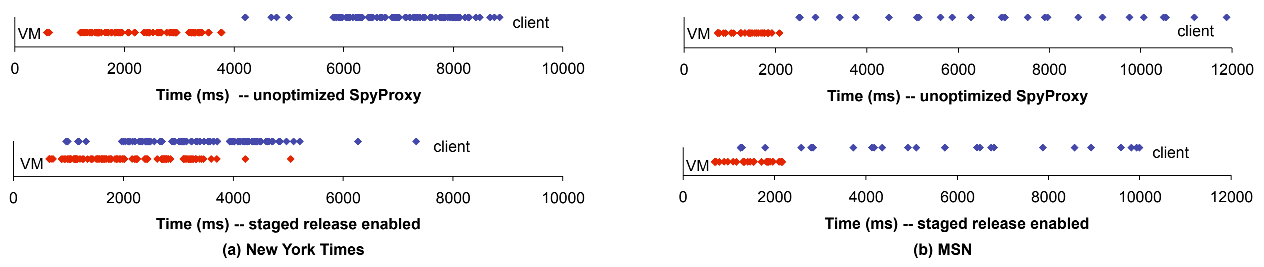

Figure 4: **Timeline of events with staged release (broadband).** The sequence of object rendering completion events that occur over time for (a) the New York Times, and (b) MSN blog pages. The top figures show the sequence of events for the unoptimized SpyProxy, while the bottom figures show what happens with staged release. In each figure, the top series of dots represents completions in the client browser and the bottom series in SpyProxy's VM browser. Staged release is effective at the early release of objects to the browser.

band connection still dominates. However, on a security cache hit, the caching optimization is extremely effective, since it eliminates the need to evaluate content in a VM.

4.4 Performance on a Realistic Workload

Previous sections examined the individual impact of each of our optimizations. In the end, however, the question remains: how does SpyProxy perform for a "typical" user Web-browsing workload? A more realistic workload will cause the performance optimizations — caching, static analysis, and staged release — to be exercised together in response to a stream of requests.

To study the behavior of SpyProxy when confronted with a realistic request stream, we measured the response latencies of 1,909 Web page requests issued by our broadband Web client. These requests were generated with a Zipf popularity distribution drawn from a list of 703 different safe URLs from 124 different sites. We chose the URLs by selecting a range of popular and unpopular sites ranked by the Alexa ranking service. By selecting real sites, we exercised our system with the different varieties and complexities of Web page content to which users are typically exposed. By generating our workload with a Zipf popularity distribution, we gave our caching optimization the opportunity to work in a realistic scenario. None of the sites we visited contained attacks; our goal was simply to evaluate the performance impact of SpyProxy on browsing.

Figure 5(a) presents a cumulative distribution function for the time to start page rendering in the client browser. This is the delay the user sees before the browser responds to a request. Figure 5(b) shows the CDF for full-page-load latencies. Each figure depicts distributions for three cases: (1) directly connecting to the Web site without SpyProxy, (2) using the optimized SpyProxy implementation, and (3) using SpyProxy with optimizations disabled. We flushed all caches before gathering the data for each distribution.

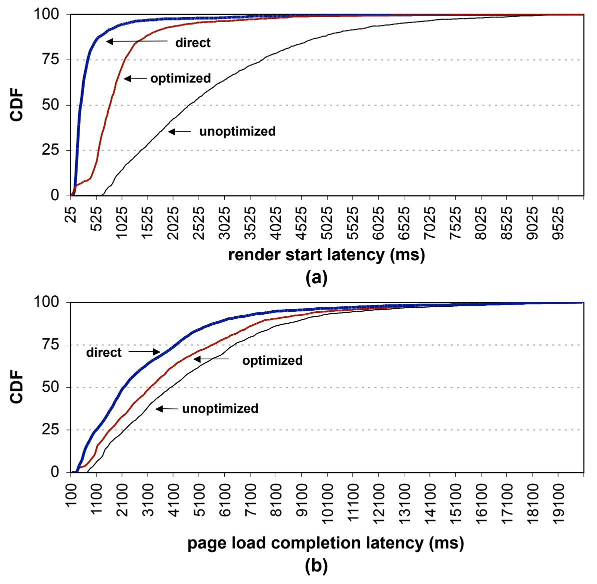

Figure 5: **Overall performance (broadband).** These graphs show the distributions of (a) render start latencies and (b) full page load latencies for a workload consisting of 1,909 requests issued to 703 pages from 102 Web sites. Each graph compares the response time for a direct client, the unoptimized system, and the fully optimized system. The artifact visible at low latencies on the optimized line in (a) corresponds to hits in our security cache.

Our results demonstrate that the optimized SpyProxy system delivers content to browsers very quickly. The median time until rendering began was 0.8 seconds in the optimized system compared to 2.4 seconds in the unoptimized system. There is still room to improve; the median start time for the direct connection was 0.2 seconds. However, the optimized system feels acceptably fast to a user. In contrast, the unoptimized system seems noticeably sluggish compared to the optimized system and direct connections.

A typical request flowing through the optimized system involves several potential sources of overhead, including interacting with the Squid proxy cache and pre-executing content in a virtual machine. In spite of this, the optimized SpyProxy effectively masks latency, resulting in an interactive, responsive system. In addition, our system generated very few false positives: only 4 of the 1,909 Web page requests resulted in an alarm being raised. Even though the offending pages were benign, they did in fact attempt to install software on the user's computer, albeit by requesting permission from the user first. For example, one of the pages prompted the user to install a browser plug-in for the QuickTime media player. We chose not to deal with such opt-in installers, as SpyProxy is primarily intended for zero-day attacks that never ask for permission before installing malware. However, we do reduce the number of false positives by including the most common browser plug-ins, such as Flash, in the base VM image.

4.5 Scalability

SpyProxy is designed to service many concurrent users in an organizational setting. Our implementation runs on a cluster of workstations, achieving incremental scalability by executing VM workers on additional nodes. We now provide some back-of-the-envelope estimations of SpyProxy's scalability. We have not performed an explicit scaling benchmark, but our calculations do provide an approximate indication of how many CPUs would be necessary to support user population of a given size.

Our estimate is based on the assumption that the CPU is likely to be the bottleneck of a deployed system; for this to be true, the system must be configured with an adequate amount of memory and network bandwidth to support the required concurrent virtual machines and Web traffic. While performing the evaluation in section 4.4, we measured the amount of CPU time required to process a Web page in SpyProxy. On a 2.8GHz Pentium 4 machine with 4GB of RAM and a single 80GB 7200 RPM disk, we found the average CPU time consumed per page was 0.35 seconds.

There is little published data on the number of Web pages users view per day. In a study of Internet content-delivery systems [38], users requested 930 HTTP objects per day on average, and another study found that an average Web page contains about 15 objects [27]. Combining these, we conservatively estimate that a typical user browses through 100 pages per day. Assuming this browsing activity is uniformly distributed over an 8-hour workday, one CPU can process 82286 Web pages per day, implying a single-CPU SpyProxy could support approximately 822 users. A single quad-core machine should be able to handle the load from an organization containing a few thousand people.

4.6 Summary

This section evaluated the effectiveness and performance of our SpyProxy prototype. Our measurements demonstrated that SpyProxy effectively detects malicious content. In our experiments, SpyProxy correctly detected and blocked every threat, including several that SiteAdvisor failed to identify. Our experiments with fully optimized SpyProxy show that a proxy-based spyware checker can be implemented with only minimal performance impact on the user. On average, the use of SpyProxy added only 600 milliseconds to the user-visible latency before rendering starts. In our experience using the system, this small additional overhead does not noticeably degrade the system's responsiveness.

5 Related Work

We now discuss related research on spyware detection and prevention, intrusion detection and firewall systems, and network proxies.

5.1 Spyware and Malware Detection

In previous work, we used passive network monitoring to measure adware propagation on the University of Washington campus [37]. In a follow-on study, we used Web crawling to find and analyze executable programs and Web pages that lead to spyware infections [28]; the trigger-based VM analysis technique in that work forms the foundation for SpyProxy's detection mechanism.

Strider HoneyMonkey [49] and the commercial SiteAdvisor service [41] both use a VM-based technique similar to ours to characterize malicious Web sites and pages. Our work differs in two main ways: we show that our VM-based technique can be used to build a transparent defense system rather than a measurement tool, and we examine optimizations that enable our system to perform efficiently and in real time.

Our system detects malicious Web content by executing it and looking for evidence of malicious side-effects. Other systems have attempted to detect malware by examining side-effects, including Gatekeeper [51], which monitors Windows extensibility hooks for evidence of spyware installation. Another recent detector identifies spyware by monitoring API calls invoked when sensitive information is stolen and transmitted [20]. However, these systems only look for malware that is already installed. In contrast, SpyProxy uses behavioral analysis to *prevent* malware installation.

Other works have looked at addressing limitations of signature-based detection. Semantics-aware malware detection [8] uses an instruction-level analysis of programs to match their behavior against signature templates. This technique improves malware detection, but not prevention. Several projects explore automatic generation of signatures for detection of unknown malware variants [7, 29, 40, 46, 48]. These typically need attack traffic and time to generate signatures, leaving some clients vulnerable when a new threat first appears.

Some commercial client-side security tools have begun to incorporate behavioral techniques, and two recent products, Prevx1 [32] and Primary Response Safe-Connect [35], use purely behavioral detection. However, these tools must run on systems packed with client-installed programs, which limits their behavioral analysis. In contrast, SpyProxy pre-executes content in a clean sandbox, where it can apply a much stricter set of behavioral rules.

Other approaches prevent Web-based malware infestations by protecting the user's system from the Web browser using VM isolation [10], OS-level sandboxing [16, 36], or logging/rollback [17]. Fundamentally, this *containment* approach is orthogonal to our *prevention* approach. Although these tools provide strong isolation, they have different challenges, such as data sharing and client-side performance overhead.

Remote playgrounds move some of the browser functionality (namely, execution of untrusted Java applets) away from the client desktop and onto dedicated machines [26]; the client browser becomes an I/O terminal to the actual browser running elsewhere. Our architecture is different — SpyProxy *pre-executes* Web pages using an unmodified browser and handles any form of active code, allowing it to capture a wider range of attacks. Nevertheless, SpyProxy could benefit from this technique in the future, for example by forwarding user input to the VM worker in AJAX sites.

Several projects tackle the detection and prevention of other classes of malware, including worms, viruses, and rootkits [19, 39, 50]. SpyProxy complements these defenses with protection against Web-borne attacks, resulting in better overall desktop security.

5.2 Intrusion Detection and Firewalls

Intrusion detection systems (e.g., Bro [31] and snort [42]) protect networks from attack by searching through incoming packets for known attack signatures. These systems are typically passive, monitoring traffic as it flows into a network and alerting a system administrator when an attack is suspected. More sophisticated intrusion detection systems attempt to identify suspicious traffic using anomaly detection [3, 4, 22, 23]. A related approach uses protocol-level analysis to look for attacks that exploit specific vulnerabilities, such as Shield [47]. The same idea has been applied at the HTML level in client-side firewalls and proxies [25, 30, 33].

These systems typically look for attack signatures for well-established protocols and services. As a result, they cannot detect new or otherwise undiscovered attacks. Since they are traditionally run in a passive manner, attacks are detected but not prevented. Our system executes potentially malicious content in a sandboxed environment, using observed side-effects rather than signatures to detect attacks and protect clients.

Shadow honeypots combine network intrusion detection systems and honeypots [2]. They route risky network traffic to a heavily instrumented version of a vulnerable application, which detects certain types of attacks at run-time. In contrast, SpyProxy does not need to instrument the Web browser that it guards, and its run-time checks are more general and easier to define.

5.3 Proxies

Proxies have been used to introduce new services between Web clients and servers. For example, they have been used to provide scalable distillation services for mobile clients [14], Web caching [1, 13, 15, 21, 52, 53], and gateway services for onion-routing anonymizers [11]. SpyProxy builds on these advantages, combining active content checking with standard proxy caching. SpyBye [30] is a Web proxy that uses a combination of blacklisting, whitelisting, ClamAV-based virus scanning, and heuristics to identify potentially malicious Web content. In contrast, SpyProxy uses execution-based analysis to identify malicious content.

6 Conclusions

This paper described the design, implementation, and evaluation of SpyProxy, an execution-based malware detection system that protects clients from malicious Web pages, such as drive-by-download attacks. SpyProxy executes active Web content in a safe virtual machine *before* it reaches the browser. Because SpyProxy relies on the behavior of active content, it can block zero-day attacks and previously unseen threats. For performance, SpyProxy benefits from a set of optimizations, including the staged release of content and caching the results of security checks.

Our evaluation of SpyProxy demonstrates that it meets its goals of safety, responsiveness, and transparency:

1. SpyProxy successfully detected and blocked all of the threats it faced, including threats not identified by other detectors.
2. The SpyProxy prototype adds only 600 milliseconds of latency to the start of page rendering—an amount that is negligible in the context of browsing over a broadband connection.
3. Our prototype integrates easily into the network and its existence is transparent to users.

Execution-based analysis does have limitations. We described several of these, including issues related to non-determinism, termination, and differences in the execution environment between the client and the proxy.

There are many existing malware detection tools, and although none of them are perfect, together they contribute to a "defense in depth" security strategy. Our goal is neither to build a perfect tool nor to replace existing tools, but to add a new weapon to the Internet security arsenal. Overall, our prototype and experiments demonstrate the feasibility and value of on-the-fly, execution-based defenses against malicious Web-page content.

7 Acknowledgements

We would like to thank the Computer Science and Engineering support staff and the Computing and Communications staff at the University of Washington for enabling our research. This research was supported in part by the National Science Foundation under grants CNS-0627367, NSF-614975, and ANI-0132817, by the Torode Family Endowed Career Development Professorship, and by the Wissner-Slivka Chair.

References

[1] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS '96)*, Miami Beach, FL, December 1996.

[2] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[3] Kevin Borders and Atul Prakash. Web tap: Detecting covert Web traffic. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, New York, NY, October 2004.

[4] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.

[5] Tanya Bragin. Measurement study of the Web through a spam lens. Technical Report TR-2007-02-01, University of Washington, Computer Science and Engineering, February 2007.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of 18th Annual IEEE Conference on Computer Communications (IEEE INFOCOM '99)*, March 1999.

[7] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.

[8] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[9] Andrew Conry-Murray. Product focus: Behavior-blocking stops unknown malicious code. `http://mirage.cs.ucr.edu/mobilecode/resources_files/behavior.pdf`, June 2002.

[10] Richard Cox, Steven Gribble, Henry Levy, and Jacob Hansen. A safety-oriented platform for Web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.

[11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[12] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[13] Brian Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of World Wide Web client proxy caches. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.

[14] Armando Fox, Steven Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St.-Malo, France, October 1997.

[15] Steven Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[16] Green Border Technologies. GreenBorder desktop DMZ solutions. `http://www.greenborder.com`, November 2005.

[17] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, Washington, DC, December 2006.

[18] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[19] Darrell Kienzle and Matthew Elder. Recent worms: A survey and trends. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM '03)*, Washington, DC, October 2003.

[20] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[21] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.

[22] Christopher Kruegel and Giovanni Vigna. Anomaly detection of Web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, New York, NY, October 2003.

[23] Lancope StealthWatch. `http://www.lancope.com`.

[24] Robert Lemos. Microsoft patch opens users to attack. `http://www.securityfocus.com/news/11408`, August 2006.

[25] LinkScanner Pro. `http://www.explabs.com/products/lspro.asp`.

[26] Dahlia Malkhi and Michael K. Reiter. Secure execution of java applets using a remote playground. *IEEE Transactions on Software Engineering*, 26(12):1197–1209, 2000.

[27] Mikhail Mikhailov and Craig Wills. Embedded objects in Web pages. Technical Report WPI-CS-TR-0005, Worcester Polytechnic Institute, Worcester, MA, March 2000.

[28] Alexander Moshchuk, Tanya Bragin, Steven Gribble, and Henry Levy. A crawler-based study of spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.

[29] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, February 2005.

[30] Niels Provos. SpyBye. `http://www.spybye.org`.

[31] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[32] Prevx. `http://www.prevx.com`.

[33] Charles Reis, John Dunagan, Helen Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[34] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[35] Sana Security. `http://www.sanasecurity.com`.

[36] Sandboxie. `http://www.sandboxie.com`.

[37] Stefan Saroiu, Steven Gribble, and Henry Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[38] Stefan Saroiu, Krishna Gummadi, Richard Dunn, Steven Gribble, and Henry Levy. An analysis of internet content delivery systems. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, New York, NY, December 2002. ACM Press.

[39] Prabhat Singh and Arun Lakhotia. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices*, 37(2):29–35, February 2002.

[40] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, December 2004.

[41] SiteAdvisor, Inc. `http://www.siteadvisor.com`.

[42] Snort. The open source network intrusion detection system. `http://www.snort.org`.

[43] StopBadware. `http://www.stopbadware.org/`.

[44] StopBadware.org - Incompetence or McCarthyism 2.0? `http://www.adwarereport.com/mt/archives/stopbadwareorg.php`.

[45] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual USENIX Technical Conference*, Boston, MA, June 2001.

[46] Hao Wang, Somesh Jha, and Vinod Ganapathy. NetSpy: Automatic generation of spyware signatures for NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, Miami Beach, FL, December 2006. http://dx.doi.org/10.1109/ACSAC.2006.34.

[47] Helen Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM 2004*, Portland, OR, August 2004.

[48] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '06)*, October 2006.

[49] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated Web patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.

[50] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, Chad Verbowski, and Aaron Johnson. Detecting stealth software with Strider GhostBuster. In *Proceedings of the*

2005 International Conference on Dependable Systems and Networks (DSN '05), Yokohama, Japan, July 2005.

[51] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *Proceedings of 18th Large Installation System Administration Conference (LISA '04)*, Atlanta, GA, November 2004.

[52] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS '99)*, Boulder, CO, October 1999.

[53] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, December 1999.