Detecting In-Flight Page Changes with Web Tripwires

Charles Reis
University of Washington

Steven D. Gribble
University of Washington

Tadayoshi Kohno
University of Washington

Nicholas C. Weaver
International Computer Science Institute

Abstract

While web pages sent over HTTP have no integrity guarantees, it is commonly assumed that such pages are not modified in transit. In this paper, we provide evidence of surprisingly widespread and diverse changes made to web pages between the server and client. Over 1% of web clients in our study received altered pages, and we show that these changes often have undesirable consequences for web publishers or end users. Such changes include popup blocking scripts inserted by client software, advertisements injected by ISPs, and even malicious code likely inserted by malware using ARP poisoning. Additionally, we find that changes introduced by client software can inadvertently cause harm, such as introducing cross-site scripting vulnerabilities into most pages a client visits. To help publishers understand and react appropriately to such changes, we introduce *web tripwires*—client-side JavaScript code that can detect most in-flight modifications to a web page. We discuss several web tripwire designs intended to provide basic integrity checks for web servers. We show that they are more flexible and less expensive than switching to HTTPS and do not require changes to current browsers.

1 Introduction

Most web pages are sent from servers to clients using HTTP. It is well-known that ISPs or other parties between the server and the client *could* modify this content in flight; however, the common assumption is that, barring a few types of client proxies, no such modifications take place. In this paper, we show that this assumption is false. Not only do a large number and variety of in-flight modifications occur to web pages, but they often result in significant problems for users or publishers or both.

We present the results of a measurement study to better understand what in-flight changes are made to web pages in practice, and the implications these changes have for end users and web publishers. In the study, our web server recorded any changes made to the HTML code of our web page for visitors from over 50,000 unique IP addresses.

Changes to our page were seen by 1.3% of the client IP addresses in our sample, drawn from a population of technically oriented users. We observed many types of changes caused by agents with diverse incentives. For example, ISPs seek revenue by injecting ads, end users seek to filter annoyances like ads and popups, and malware authors seek to spread worms by injecting exploits.

Many of these changes are undesirable for publishers or users. At a minimum, the injection or removal of ads by ISPs or proxies can impact the revenue stream of a web publisher, annoy the end user, or potentially expose the end user to privacy violations. Worse, we find that several types of modifications introduce bugs or even vulnerabilities into many or all of the web pages a user visits—pages that might otherwise be safe and bug-free. We demonstrate the threats these modifications pose by building successful exploits of the vulnerabilities.

These discoveries reveal a diverse ecosystem of agents that modify web pages. Because many of these modifications have negative consequences, publishers may have incentives to detect or even prevent them from occurring. Detection can help publishers notify users that a page might not appear as intended, take action against those who make unwanted changes, debug problems due to modified pages, and potentially deter some types of changes. Preventing modifications may sometimes be important, but there may also be types of page changes worth allowing. For example, some enterprise proxies modify web pages to increase client security, such as Blue Coat WebFilter [9] and BrowserShield [30].

HTTPS offers a strong, but rigid and costly, solution for these issues. HTTPS encrypts web traffic to prevent in-flight modifications, though proxies that act as HTTPS endpoints may still alter pages without any indication to the server. Encryption can prevent even beneficial page

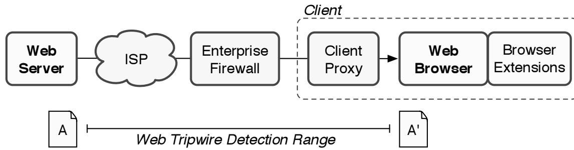

Figure 1: Web tripwires can detect any modifications to the HTML source code of a page made between the server and the browser.

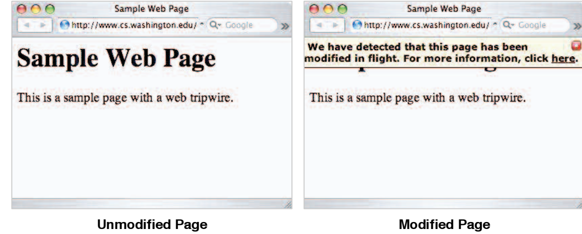

Figure 2: If a web tripwire detects a change, it displays a message to the user, as in the screenshot on the right.

changes, as well as web caching, compression, and other useful services that rely on the open nature of HTTP.

As a result, we propose that concerned web publishers adopt *web tripwires* on their pages to help understand and react to any changes made in flight. Web tripwires are client-side JavaScript code that can detect most modifications to unencrypted web pages. Web tripwires are not secure and cannot detect all changes, but they can be made robust in practice. We present several designs for web tripwires and show that they can be deployed at a lower cost than HTTPS, do not require changes to web browsers, and support various policy decisions for reacting to page modifications. They provide web servers with practical integrity checks against a variety of undesirable or dangerous modifications.

The rest of this paper is organized as follows. Section 2 describes our measurement study of in-flight page changes and discusses the implications of our findings. In Section 3, we compare several web tripwire implementation strategies that allow publishers to detect changes to their own pages. We evaluate the costs of web tripwires and their robustness to adversaries in Section 4. Section 5 illustrates how our web tripwire toolkit is easy to deploy and can support a variety of policies. Finally, we present related work in Section 6 and conclude in Section 7.

2 In-Flight Modifications

Despite the lack of integrity guarantees in HTTP, most web publishers and end users expect web pages to arrive at the client as the publisher intended. Using measurements of a large client population, we find that this is not the case. ISPs, enterprises, end users, and malware authors all have incentives to modify pages, and we find evidence that each of these parties does so in practice. These changes often have undesirable consequences for publishers or users, including injected advertisements, broken pages, and exploitable vulnerabilities. These results demonstrate the precariousness of today's web, and that it can be dangerous to ignore the absence of integrity protection for web content.

To understand the scope of the problem, we designed a measurement study to test whether web pages arrive at the client unchanged. We developed a web page that could detect changes to its HTML source code made by an agent between the server and the browser, and we attracted a diverse set of clients to the page to test many paths through the network. Our study seeks to answer two key questions:

- What kinds of page modifications occur in practice, and how frequently?
- Do the changes have unforeseen consequences?

We found that clients at over 1% of 50,000 IP addresses saw some change to the page, many with negative consequences. In the rest of this section, we discuss our measurement technique and the diverse ecosystem of page modifications that we observed.

2.1 Measurement Infrastructure

Our measurement study identifies changes made to our web page between the web server and the client's browser, using code delivered by the server to the browser. This technique allows us to gather results from a large number of clients in diverse locations, although it may not detect agents that do not modify every page.

Technology. Our measurement tool consists of a web page with JavaScript code that detects page modifications. We refer to this code as a *web tripwire* because it can be unobtrusively placed on a web page and triggered if it detects a change. As shown in Figure 1, our web tripwire detects changes to HTML source code made anywhere between the server and browser, including those caused by ISPs, enterprise firewalls, and client-side proxies. We did not design the web tripwire to detect changes made by browser extensions, because extensions are effectively part of the browser, and we believe they are likely installed with the knowledge and consent of the user. In practice, browser extensions do not trigger the tripwire because they operate on the browser's inter-

nal representation of the page and not the HTML source code itself.

Our web tripwire is implemented as JavaScript code that runs when the page is loaded in the client's browser. It reports any detected changes to the server and displays a message to the user, as seen in Figure 2. Our implementation can display the difference between the actual and expected contents of the page, and it can collect additional feedback from the user about her environment. Further implementation details can be found in Section 3.

We note two caveats for this technique. First, it may have false negatives. Modifying agents may choose to only alter certain pages, excluding those with our web tripwires. We do not expect any false positives, though, so our results are a lower bound for the actual number of page modifications.[1] Second, our technique is not cryptographically secure. An adversarial agent could remove or tamper with our scripts to evade detection. For this study, we find it unlikely that such tampering would be widespread, and we discuss how to address adversarial agents in Section 4.2.

Realism. We sought to create a realistic setting for our measurement page, to increase the likelihood that agents might modify it. We included HTML tags from web authoring software, randomly generated text, and keywords with links.

We were also guided by initial reports of ISPs that injected advertisements into their clients' web traffic, using services from NebuAd [5]. These reports suggested that only pages from `.com` top-level domains (TLDs) were affected. To test this, our measurement page hosts several frames with identical web tripwires, each served from a different TLD. These frames are served from `vancouver.cs.washington.edu`, `uwsecurity.com`, `uwprivacy.org`, `uwcse.ca`, `uwsystems.net`, and 128.208.6.241.

We introduced additional frames during the experiment, to determine if any agents were attempting to "whitelist" the domains we had selected to evade detection. After our measurement page started receiving large numbers of visitors, we added frames at `www.happyblimp.com` and `www2.happyblimp.com`.

In the end, we found that most changes were made indiscriminately, although some NebuAd injections were `.com`-specific and other NebuAd injections targeted particular TLDs with an unknown pattern.

Exposure. To get a representative view of in-flight page modifications, we sought visitors from as many vantage points as possible. Similar studies such as the ANA Spoofer Project [8] attracted thousands of participants by posting to the Slashdot news web site, so we also pursued this approach.

Although our first submission to Slashdot was not successful, we were able to circulate a story among other sites via Dave Farber's "Interesting People" mailing list. This led another reader to successfully post the story to Slashdot.

Similarly, we attracted traffic from Digg, a user-driven news web site. We encouraged readers of our page to aid our experiment by voting for our story on Digg, promoting it within the site's collaborative filter. Within a day, our story reached the front page of Digg.

2.2 Results Overview

On July 24, 2007, our measurement tool went live at `http://vancouver.cs.washington.edu`, and it appeared on the front pages of Slashdot and Digg (among other technology news sites) the following day. The tool remains online, but our analysis covers data collected for the first 20 days, which encompasses the vast majority of the traffic we received.

We collected test results from clients at 50,171 unique IP addresses. 9,507 of these clients were referred from Slashdot, 21,333 were referred from Digg, and another 705 were referred from both Slashdot and Digg. These high numbers of referrals indicate that these sites were essential to our experiment's success.

The modifications we observed are summarized in Table 1. At a high level, clients at 657 IP addresses reported modifications to at least one of the frames on the page. About 70% of the modifications were caused by client-side proxies such as popup blockers, but 46 IP addresses did report changes that appeared to be intentionally caused by their ISP. We also discovered that the proxies used at 125 addresses left our page vulnerable to cross-site scripting attacks, while 3 addresses were affected by client-based malware.

2.3 Modification Diversity

We found a surprisingly diverse set of changes made to our measurement page. Importantly, these changes were often misaligned with the goals of the publisher or the end user. Publishers wish to deliver their content to users, possibly with a revenue stream from advertisements. Users wish to receive the content safely, with few annoyances. However, the parties in Figure 1, including ISPs, enterprises, users, and also malware authors, have incentives to modify web pages in transit. We found that these parties do modify pages in practice, often adversely impacting the user or publisher. We offer a high level survey of these changes and incentives below.

[1] In principle, a false positive could occur if an adversary forges a web tripwire alarm. Since this was a short-term measurement study, we do not expect that we encountered any adversaries or false positives.

Category	IPs	ISP	Enterprise	User	Attacker	Examples
Popup Blocker	277			✓		Zone Alarm (210), **CA Personal Firewall (17)**, Sunbelt Popup Killer (12)
Ad Blocker	188			✓		**Ad Muncher (99)**, Privoxy (58), **Proxomitron (25)**
Problem in Transit	118	✓				Blank Page (107), Incomplete Page (7)
Compression	30	✓				**bmi.js (23)**, Newlines removed (6), **Distillation (1)**
Security or Privacy	17		✓	✓		Blue Coat (15), **The Cloak (1)**, AnchorFree (1)
Ad Injector	16	✓				MetroFi (6), FairEagle (5), LokBox (1), Front Porch (1), PerfTech (1), Edge Technologies (1), knects.net (1)
Meta Tag Changes	12		✓	✓		Removed meta tags (8), Reformatted meta tags (4)
Malware	3				✓	W32.Arpiframe (2), Adware.LinkMaker (1)
Miscellaneous	3			✓		New background color (1), **Mark of the Web (1)**

Table 1: Categories of observed page modifications, the number of client IP addresses affected by each, the likely parties responsible, and examples. Each example is followed by the number of IP addresses that reported it; examples listed in bold introduced defects or vulnerabilities into our page.

ISPs. ISPs have at least two incentives to modify web traffic: to generate revenue from advertising and to reduce traffic using compression. Injected advertisements have negative impact for many users, who view them as annoyances.

In our results, we discovered several distinct ISPs that appeared to insert ad-related scripts into our measurement page. Several companies offer to partner with ISPs by providing them appliances that inject such ads. For example, we saw 5 IP addresses that received injected code from NebuAd's servers [2]. Traceroutes suggested that these occurred on ISPs including Red Moon, Mesa Networks, and XO, as well as an IP address belonging to NebuAd itself. Other frequently observed ad injections were caused by MetroFi, a company that provides free wireless networks in which all web pages are augmented with ads. We also observed single IP addresses affected by other similar companies, including LokBox, Front Porch, PerfTech, Edge Technologies, and `knects.net`.

Notably, these companies often claim to inject ads based on behavioral analysis, so that they are targeted to the pages a user has visited. Such ads may leak private information about a user's browsing history to web servers the user visits. For example, a server could use a web tripwire to determine which specific ad has been injected for a given user. The choice of ad may reveal what types of pages the user has recently visited.

We also observed some ISPs that alter web pages to reduce network traffic. In particular, several cellular network providers removed extra whitespace or injected scripts related to image distillation [16]. Such modifications are useful on bandwidth-constrained networks, though they may also unintentionally cause page defects, as we describe in Section 2.4.1.

Enterprises. Enterprises have incentives to modify the pages requested by their clients as well, such as traffic reduction and client protection. Specifically, we observed proxy caches that remove certain `meta` tags from our measurement page, allowing it to be cached against our wishes. Such changes can go against a publisher's desires or present stale data to a user. Our results also included several changes made by Blue Coat WebFilter [9], an enterprise proxy that detects malicious web traffic.

End Users. Users have several incentives for modifying the pages they receive, although these changes may not be in the best interests of the publishers. We found evidence that users block annoyances such as popups and ads, which may influence a publisher's revenue stream. Users also occasionally modify pages for security, privacy, or performance.

The vast majority of page modifications overall are caused by user-installed software such as popup blockers and ad blockers. The most common modifications come from popup blocking software. Interestingly, this includes not only dedicated software like Sunbelt Popup Killer, but also many personal firewalls that modify web traffic to block popups. In both types of software, popups are blocked by JavaScript code injected into every page. This code interposes on calls to the browser's `window.open` function, much like Naccio's use of program rewriting for system call interposition [15].

Ad blocking proxies also proved to be quite popular. We did not expect to see this category in our results, be-

cause our measurement page contained no ads. That is, ad blocking proxies that solely removed ads from pages would have gone unnoticed. However, we detected numerous ad blocking proxies due to the JavaScript code they injected into our page. These proxies included Ad Muncher, Privoxy, Proxomitron, and many others.

Beyond these annoyance blocking proxies, we found user-initiated changes to increase security, privacy, and performance. AnchorFree Hotspot Shield claims to protect clients on wireless networks, and Internet Explorer adds a "Mark of the Web" comment to saved pages to prevent certain attacks [28]. Users also employed web-based anonymization services such as The Cloak [3], as well as proxies that allowed pages to be cached by removing certain `meta` tags.

Malware Authors. Surprisingly, our measurement tool was also able to detect certain kinds of malware and adware. Malware authors have clear incentives for modifying web pages, either as a technique for spreading exploit code or to receive revenue from injected advertisements. These changes are clearly adversarial to users.

In one instance, a client that was infected by Adware.LinkMaker [34] visited our measurement page. The software made extensive changes to the page, converting several words on the page into doubly underlined links. If the user hovered his mouse cursor over the links, an ad frame was displayed.

Two other clients saw injected content that appears consistent with the W32.Arpiframe worm [35]. In these cases, the clients themselves may not have been infected, as the Arpiframe worm attempts to spread through local networks using ARP cache poisoning [40]. When an infected client poisons the ARP cache of another client, it can then act as a man-in-the-middle on HTTP sessions. Recent reports suggest that web *servers* may also be targeted by this or similar worms, as in the recent case of a Chinese security web site [12].

2.4 Unanticipated Problems

In the cases discussed above, page modifications are made based on the incentives of some party. However, we discovered that many of these modifications actually had severe unintentional consequences for the user, either as broken page functionality or exploitable vulnerabilities. The threats posed by careless page modifications thus extend far beyond annoyances such as ad injections.

2.4.1 Page Defects

We observed two classes of bugs that were unintentionally introduced into web pages as a result of modifications. First, some injected scripts caused a JavaScript stack overflow in Internet Explorer when they were combined with the scripts in our web tripwire. For example, the `xpopup.js` popup blocking script in CA Personal Firewall interfered with our calls to `document.write`. Similar problems occurred with a compression script called `bmi.js` injected by several ISPs. These bugs occasionally prevented our web tripwire from reporting results, but users provided enough feedback to uncover the issue. In general, such defects may occur when combining multiple scripts in the same namespace without the ability to sufficiently test them.

Second, we discovered that the CA Personal Firewall modifications interfered with the ability to post comments and blog entries on many web sites. Specifically, code injected by the firewall appeared in users' comments, often to the chagrin of the users. We observed 28 instances of "`_popupControl()`" appearing on MySpace blogs and comments, and well over 20 sites running the Web Wiz Forums software [39] that had the same code in their comments. We reproduced the problem on Web Wiz Forums' demo site, learning that CA Personal Firewall injected the popup blocking code into the frame in which the user entered his comments. We observed similar interference in the case of image distillation scripts that contained the keyword "`nguncompressed`."

2.4.2 Vulnerabilities

More importantly, we discovered several types of page changes that left the modified pages vulnerable to cross-site scripting (XSS) attacks. The impact of these vulnerabilities should not be understated: the modifications made *most or all* of the pages a user visited exploitable. Such exploits could expose private information or otherwise hijack any page a user requests.

Ad Blocking Vulnerabilities. We observed exploitable vulnerabilities in three ad-blocking products: two free downloadable filter sets for Proxomitron (released under the names Sidki [33] and Grypen [19]), plus the commercial Ad Muncher product [4]. At the time of our study, each of these products injected the URL of each web page into the body of the page itself, as part of a comment. For example, Ad Muncher injected the following JavaScript comment onto Google's home page:

```
// Original URL:   http://www.google.com
```

These products did not escape any of the characters in the URL, so adversaries were able to inject script code into the page by convincing users to visit a URL similar to the following:

```
http://google.com/?</script><script>alert(1);
```

Servers often ignore unknown URL parameters (following the '`?`'), so the page was delivered as usual.

However, when Ad Muncher or Proxomitron copied this URL into the page, the "`</script>`" tag terminated the original comment, and the script code in the remainder of the URL was executed as part of the page. To exploit these vulnerabilities, an adversary must convince a user to follow a link of his own construction, possibly via email or by redirecting the user from another page.

It is worth noting that our measurement tool helped us discover these vulnerabilities. Specifically, we were able to search for page changes that placed the page's URL in the body of the page. We flagged such cases for further security analysis.

We developed two exploit pages to demonstrate the threat posed by this attack. Our exploit pages first detect whether a vulnerable proxy is in use, by looking for characteristic modifications in their own source code (e.g., an "Ad Muncher" comment).

In one exploit, our page redirects to a major bank's home page.[2] The bank's page has a login form but is served over HTTP, not HTTPS. (The account name and password are intended to be sent over HTTPS when the user submits the form.) Our exploit injects script code into the bank's page, causing the login form to instead send the user's account name and password to an adversary's server.

In a second exploit, we demonstrate that these vulnerabilities are disconcerting even on pages for which users do not normally expect an HTTPS connection. Here, our exploit page redirects to Google's home page and injects code into the search form. If the user submits a query, further exploit code manipulates the search results, injecting exploit code into all outgoing links. This allows the exploit to retain control of all subsequent pages in the browser window, until the user either enters a new URL by hand or visits an unexploited bookmark.

In the case of Ad Muncher (prior to v4.71), any HTTP web site that was not mentioned on the program's exclusion list is affected. This list prevents Ad Muncher from injecting code into a collection of JavaScript-heavy web pages, including most web mail sites. However, Ad Muncher did inject vulnerable code into the login pages for many banks, such as Washington Mutual, Chase, US Bank, and Wachovia, as well as the login pages for many social networking sites. For most social networking sites, it is common to only use HTTPS for sending the login credentials, and then revert to HTTP for pages within the site. Thus, if a user is already logged into such a site, an adversary can manipulate the user's account by injecting code into a page on the site, without any interaction from the user. This type of attack can even be conducted in a hidden frame, to conceal it from the user.

[2] We actually ran the exploit against an accurate local replica of the bank's home page, to avoid sending exploit code to the bank's server.

In both Proxomitron filter sets (prior to September 8, 2007), all HTTP traffic is affected in the default configuration. Users are thus vulnerable to all of the above attack scenarios, as well as attacks on many web mail sites that revert to HTTP after logging in (e.g., Gmail, Yahoo Mail). Additionally, Proxomitron can be configured to also modify HTTPS traffic, intentionally acting as a "man in the middle." If the user enables this feature, all SSL encrypted pages are vulnerable to script injection and thus leaks of critically private information.

We reported these vulnerabilities to the developers of Ad Muncher and the Proxomitron filter sets, who have released fixes for the vulnerabilities.

Internet Explorer Vulnerability. We identified a similar but less severe vulnerability in Internet Explorer. IE injects a "Mark of the Web" into pages that it saves to disk, consisting of an HTML comment with the page's URL [28]. This comment is vulnerable to similar attacks as Ad Muncher and Proxomitron, but the injected scripts only run if the page is loaded from disk. In this context, the injected scripts have no access to cookies or the originating server, only the content on the page itself. This vulnerability was originally reported to Microsoft by David Vaartjes in 2006, but no fix is yet available [37].

The Cloak Vulnerabilities. Finally, we found that the "The Cloak" anonymization web site [3] contains two types of XSS vulnerabilities. The Cloak provides anonymity to its users by retrieving all pages on their behalf, concealing their identities from web servers. The Cloak processes and rewrites many HTML tags on each page to ensure no identifying information is leaked. It also provides users with options to rewrite or delete all JavaScript code on a page, to prevent the code from exposing their IP address.

We discovered that The Cloak replaced some tags with a comment explaining why the tag was removed. For example, our page contained a `meta` tag with the name "generatorversion." The Cloak replaced this tag with the following HTML comment:

```
<!-- the-cloak note - deleting possibly dangerous
  META tag - unknown NAME 'generatorversion' -->
```

We found that a malicious page could inject script code into the page by including a carefully crafted `meta` tag, such as the following:

```
<meta name="foo--><script>alert(1);</script>">
```

This script code runs and bypasses The Cloak's policies for rewriting or deleting JavaScript code. We reported this vulnerability to The Cloak, and it has been resolved as of October 8, 2007.

Additionally, The Cloak faces a more fundamental problem because it bypasses the browser's "same origin

policy," which prevents documents from different origins from accessing each other [31]. To a client's browser, all pages appear to come from `the-cloak.com`, rather than their actual origins. Thus, the browser allows all pages to access each other's contents. We verified that a malicious page could load sensitive web pages (even HTTPS encrypted pages) from other origins into a frame and then access their contents. This problem is already known to security professionals [20], though The Cloak has no plans to address it. Rather, users are encouraged to configure The Cloak to delete JavaScript code to be safe from attack.

OS Analogy. These vulnerabilities demonstrate the power wielded by web page rewriting software, and the dangers of any flaws in its use. An analogy between web browsers and operating systems helps to illustrate the severity of the problem. Most XSS vulnerabilities affect a single web site, just as a security vulnerability in a program might only affect that program's operation. However, vulnerabilities in page rewriting software can pose a threat for *most or all* pages visited, just as a root exploit may affect all programs in an operating system. Page rewriting software must therefore be carefully scrutinized for security flaws before it can be trusted.

3 Web Tripwires

Our measurement study reveals that in-flight page modifications can have many negative consequences for both publishers and users. As a result, publishers have an incentive to seek integrity mechanisms for their content. There are numerous scenarios where detecting modifications to one's own web page may be useful:

- Search engines could warn users of injected scripts that might alter search results.
- Banks could disable login forms if their front pages were modified.
- Web mail sites could debug errors caused by injected scripts.
- Social networking sites could inform users if they detect vulnerable proxies, which might put users' accounts at risk.
- Sites with advertising could object to companies that add or replace ads.

Publishers may also wish to *prevent* some types of page changes, to prevent harm to their visitors or themselves.

HTTPS provides one rigid solution: preventing page modifications using encryption. However, the use of HTTPS excludes many beneficial services, such as caching by web proxies, image distillation by ISPs with low bandwidth networks, and security checks by enterprise proxies. HTTPS also imposes a high cost on the server, in terms of financial expense for signed certificates, CPU overhead on the server, and additional latency for key exchange.

In cases where HTTPS is overly costly, we propose that publishers deploy web tripwires like those used in our measurement study. Web tripwires can effectively detect most HTML modifications, at low cost and in today's web browsers. Additionally, they offer more flexibility than HTTPS for reacting to detected changes.

3.1 Goals

Here, we establish a set of goals a publisher may have for using a web tripwire as a page integrity mechanism. Note that some types of tripwires may be worthwhile even if they do not achieve all of the goals.

First, a web tripwire should detect any changes to the HTML of a web page after it leaves the server and before it arrives at the client's browser. We exclude changes from browser extensions, as we consider these part of the user agent functionality of the browser. We also currently exclude changes to images and embedded objects, although these could be addressed in future work.

Second, publishers may wish for a web tripwire to prevent certain changes to the page. This goal is difficult to accomplish without cryptographic support, however, and it may not be a prerequisite for all publishers.

Third, a web tripwire should be able to pinpoint the modification for both the user and publisher, to help them understand its cause.

Fourth, a web tripwire should not interfere with the functionality or performance of the page that includes it. For example, it should preserve the page's semantics, support incremental rendering of the page, and avoid interfering with the browser's back button.

3.2 Designs & Implementations

Several implementation strategies are possible for building web tripwires. Unfortunately, limitations in popular browsers make tripwires more difficult to build than one might expect. Here, we describe and contrast five strategies for building JavaScript-based web tripwires.[3] We also compare against the integrity properties of HTTPS as an alternative mechanism. The tradeoffs between these strategies are summarized in Table 2.

Each of our implementations takes the same basic approach. The web server delivers three elements to the browser: the *requested page*, a *tripwire script*, and a *known-good representation* of the requested page.

[3] We focus on JavaScript rather than Flash or other content types to ensure broad compatibility.

Goal	Count Scripts	Check DOM	XHR then Overwrite	XHR then Redirect	XHR on Self	HTTPS
Detects all HTML changes	✗	✓	✓	✓	✓	✓
Prevents changes*	✗	✗	✓	✗	✗	✓
Displays difference	✗	✗	✓	✓	✓	✗
Preserves semantics	✓	✓	✗	✓	✓	✓
Renders incrementally	✓	✓	✗	✗	✓	✓
Supports back button	✓	✓	✗	✗	✓	✓

Table 2: Comparison of how well each tripwire implementation achieves the stated goals. (*Neither "XHR then Overwrite" nor HTTPS can prevent all changes. The former allows full page substitutions; the latter allows changes by proxies that act as the encryption endpoint, at the user's discretion.)

The known-good representation may take one of several forms; we use either a checksum of the page or a full copy of the page's HTML, stored in an encoded string to deter others from altering it. A checksum may require less space, but it cannot easily pinpoint the location of any detected change. When all three of the above elements arrive in the user's browser, the tripwire script compares the requested page with the known-good representation, detecting any in-flight changes.

We note that for all tripwire implementations, the web server must know the intended contents of the page to check. This requirement may sound trivial, but many web pages are simply the output of server-based programs, and their contents may not be known in advance. For these *dynamic* web pages, the server may need to cache the contents of the page (or enough information to reconstruct the content) in order to produce a tripwire with the known-good representation. Alternatively, servers with dynamic pages could use a web tripwire to test a separate static page in the background. This technique may miss carefully targeted page changes, but it would likely detect most of the agents we observed.

We have implemented each of the strategies described below and tested them in several modern browsers, including Firefox, Internet Explorer, Safari, Opera, and Konqueror. In many cases, browser compatibility limited the design choices we could pursue.

3.2.1 Count Scripts

Our simplest web tripwire merely counts the number of script tags on a page. Our measurement results indicate that such a tripwire would have detected 90% of the modifications, though it would miss any changes that do not affect script tags (*e.g.*, those made by the W32.Arpiframe worm). Here, the known-good representation of the page is simply the expected number of script tags on the page. The tripwire script compares against the number of script tags reported by the Document Object Model (DOM) to determine if new tags were inserted.

If a change is detected, however, it is nontrivial to determine which of the scripts do not belong or prevent them from running. This approach does miss many types of modifications, but it is simple and does not interfere with the page.

3.2.2 Check DOM

For a more comprehensive integrity check, we built a web tripwire that compares the full page contents to a known-good representation. Unfortunately, JavaScript code cannot directly access the actual HTML string that the browser received. Scripts only have access to the browser's internal DOM tree, through variables such as `document.documentElement.innerHTML`. This internal representation varies between browsers and often even between versions of the same browser. Thus, the server must pre-render the page in all possible browsers and versions in order to provide a known-good representation of the page for any client. This technique is thus generally impractical.

Additionally, the server cannot always accurately identify a client's user agent, so it cannot know which representation to send. Instead, it must send all known page representations to each client. We send a list of checksums to minimize space overhead. The tripwire script verifies that the actual page's checksum appears in the array. Because checksums are used, however, this strategy cannot pinpoint the location of a change.

3.2.3 XHR then Overwrite

Rather than checking the browser's internal representation of the page, our third strategy fetches the user's requested page from the server as data. We achieve this using an `XmlHttpRequest` (XHR), which allows scripts to fetch the contents of XML or other text-based documents, as long as the documents are hosted by the same server as the current page. This is an attractive technique for web tripwires for several reasons. First, the

tripwire script receives a full copy of the requested page as a string, allowing it to perform comparisons. Second, the request itself is indistinguishable from a typical web page request, so modifying agents will modify it as usual. Third, the response is unlikely to be modified by browser extensions, because extensions expect the response to contain XML data that should not be altered. As a result, the tripwire script can get an accurate view of any in-flight modifications to the page.

In our first XHR-based web tripwire, the server first sends the browser a small *boot page* that contains the tripwire script and a known-good representation of the requested page (as an encoded string). The tripwire script then fetches the requested page with an XHR. It compares the response with the known-good representation to detect any changes, and it then overwrites the contents of the boot page, using the browser's `document.write` function.

This strategy has the advantage that it could *prevent* many types of changes by always overwriting the boot page with the known-good representation, merely using the XHR as a test. However, adversaries could easily replace the boot page's contents, so this should not be viewed as a secure mechanism.

Unfortunately, the overwriting strategy has several drawbacks. First, it prevents the page from rendering incrementally, because the full page must be received and checked before it is rendered. Second, the use of `document.write` interferes with the back button in Firefox, though not in all browsers. Third, we discovered other significant bugs in the `document.write` function in major browsers, including Internet Explorer and Safari. This function has two modes of operation: it can append content to a page if it is called as the page is being rendered, or it can replace the entire contents of the page if called after the page's `onload` event fires. Many web sites successfully use the former mode, but our tripwire must use the latter mode because the call is made asynchronously. We discovered bugs in `document.write`'s latter mode that can cause subsequent XHRs and cookie accesses to fail in Safari, and that can cause Internet Explorer to hang if the resulting page requests an empty script file. As a result, this overwriting approach may only be useful in very limited scenarios.

However, our measurement tool in Section 2 was small and simple enough that these limitations were not a concern. In fact, we used this strategy in our study.

3.2.4 XHR then Redirect

We made a small variation to the above implementation to avoid the drawbacks of using `document.write`. As above, the tripwire script retrieves the originally requested page with an XHR and checks it. Rather than overwriting the page, the script redirects the browser to the requested page. Because we mark the page as cacheable, the browser simply renders the copy that was cached by the XHR, rather than requesting a new copy from the server. However, this approach still prevents incremental rendering, and it loses the ability to prevent any changes to the page, because it cannot redirect to the known-good representation. It also consistently breaks the back button in all browsers.

3.2.5 XHR on Self

Our final implementation achieves all of our stated goals except change prevention. In this XHR-based approach, the server first delivers the requested page, rather than a small boot page. This allows the page to render incrementally. The requested page instructs the browser to fetch an external tripwire script, which contains an encoded string with the known-good representation of the page. The tripwire script then fetches another copy of the requested page with an XHR, to perform the integrity check. Because the page is marked as cacheable (at least for a short time), the browser returns it from its cache instead of contacting the server again.[4]

This strategy cannot easily prevent changes, especially injected scripts that might run before the tripwire script. However, it can detect most changes to the requested page's HTML and display the difference to the user. It also preserves the page's semantics, the ability to incrementally render the page, and the use of the back button. In this sense, we view this as the best of the implementations we present. We evaluate its performance and robustness to adversarial changes in Section 4.

3.2.6 HTTPS

Finally, we compare the integrity properties of HTTPS with those of the above web tripwire implementations. Notably, the goals of these mechanisms differ slightly. HTTPS is intended to provide confidentiality and integrity checks for the *client*, but it offers no indication to the server if these goals are not met (e.g., if a proxy acts as the encryption end point). Web tripwires are intended to provide integrity checks for the *server*, optionally notifying the client as well. Thus, HTTPS and web tripwires can be seen to complementary in some ways.

As an integrity mechanism, HTTPS provides stronger security guarantees than web tripwires. It uses encryption to detect all changes to web content, including images and binary data. It prevents changes by simply rejecting any page that has been altered in transit. It also

[4] If the page were not cached, the browser would request it a second time from the server. In some cases, the second request may see a different modification than the first request.

preserves the page's semantics and ability to incrementally render.

However, HTTPS supports fewer policy decisions than web tripwires, such as allowing certain beneficial modifications. It also incurs higher costs for the publisher, as we discuss in Section 4.

4 Evaluation

To evaluate the strengths and weaknesses of web tripwires for publishers who might deploy them, we ask three questions:

1. Are web tripwires affordable, relative to HTTP pages without tripwires?
2. How do the costs of web tripwires compare to the costs of HTTPS?
3. How robust are web tripwires against adversaries?

We answer these questions by quantifying the performance of pages with and without web tripwires and HTTPS, and by discussing how publishers can react to adversarial page modifications.

4.1 Web Tripwire Overhead

To compare the costs for using web tripwires or HTTPS as page integrity mechanisms, we measured the client-perceived latency and server throughput for four types of pages. As a baseline, we used a local replica of a major bank's home page, served over HTTP. This is a realistic example of a page that might deploy a tripwire, complete with numerous embedded images, scripts, and stylesheets. We created two copies of this page with web tripwires, one of which was rigged to report a modification. In both cases, we used the "XHR on Self" tripwire design, which offers the best strategy for detecting and not preventing changes. We served a fourth copy of the page over HTTPS, without a web tripwire.

All of our experiments were performed on Emulab [41], using PCs with 3 GHz Xeon processors. We used an Apache 2 server on Fedora Core 6, without any hardware acceleration for SSL connections.

Latency. For each page, we measured client-perceived latency using small scripts embedded in the page. We measured the start latency (i.e., the time until the first script runs) to show the responsiveness of the page, and we measured the end latency (i.e., the time until the page's onload event fires) to show how long the page takes to render fully. We also measured the number of bytes transferred to the client, using Wireshark [14]. Our tests were conducted with a Windows XP client running Firefox, using a simulated broadband link with 2 Mbps bandwidth and 50 ms one-way link latency. Each reported value is the average of 5 trials, and the maximum relative error was 3.25%.

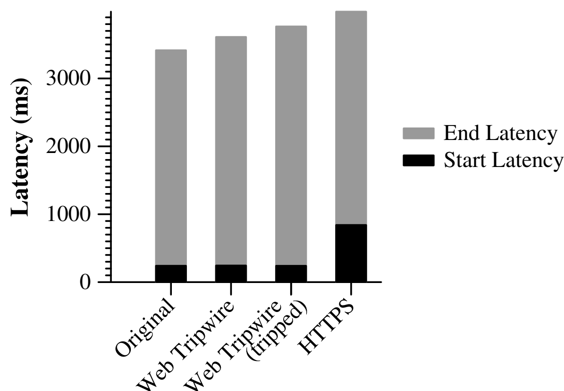

Figure 3: Impact of web tripwires and HTTPS on client perceived latency.

Figure 3 shows that the pages with web tripwires did not increase the start latency over the original page (i.e., all were around 240 ms). In comparison, the extra round trip times for establishing an SSL connection contributed to a much later start for the HTTPS page, at 840 ms.

The time spent rendering for the web tripwires was longer than for the HTTP and HTTPS pages, because the tripwires required additional script computation in the browser. The web tripwire that reported a modification took the longest, because it computed the difference between the actual and expected page contents. Despite this, end-to-end latencies of the tripwire pages were still lower than for the HTTPS page.

Table 3 shows that transmitting the web tripwire increased the size of the transferred page by 17.3%, relative to the original page. This increase includes a full encoded copy of the page's HTML, but it is a small percentage of the other objects embedded in the page.

Future web tripwire implementations could be extended to check all data transferred, rather than just the page's HTML. The increase in bytes transferred is then proportional to the number of bytes being checked, plus the size of the tripwire code. If necessary, this overhead could be reduced by transmitting checksums or digests instead of full copies.

Throughput. We measured server throughput using two Fedora Core 6 clients running httperf, on a 1 Gbps network with negligible latency. For each page, we increased the offered load on the server until the number of sustained sessions peaked. We found that the server was CPU bound in all cases. Each session simulated one visit to the bank's home page, including 32 separate requests.

Technique	Data Transferred
Original	226.6 KB
Web Tripwire	265.8 KB
Web Tripwire (tripped)	266.0 KB
HTTPS	230.6 KB

Table 3: Number of kilobytes transferred from server to client for each type of page.

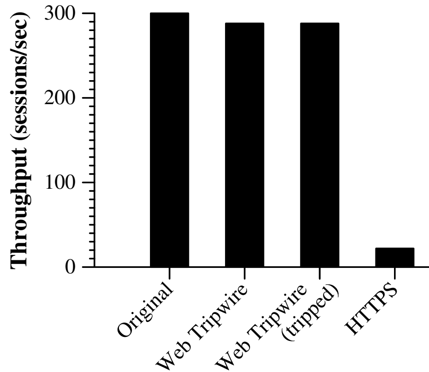

Figure 4: Impact of web tripwires and HTTPS on server throughput.

Figure 4 shows our results. The web tripwire caused only a 4% degradation of throughput compared to the original page. In comparison, the throughput dropped by over an order of magnitude when using HTTPS, due to the heavy CPU load for the SSL handshake.

For well-provisioned servers, HTTPS throughput may be improved by using a hardware accelerator. However, such hardware introduces new costs for publishers.

4.2 Handling Adversaries

In some cases, agents that modify web pages may wish for their behavior to remain undetected. For example, adversarial agents in the network may wish to inject ads, scripts, or even malicious code without being detected by the user or the publisher. Similarly, end users may wish to conceal the use of some proxies, such as ad-blockers, from the publisher.

In general, web tripwires cannot detect all changes to a page. For example, web tripwires cannot detect *full page substitutions*, in which an adversary replaces the requested content with content of his choice. Thus, we cannot address adversaries who are determined to deliver malicious content at all costs.

Instead, we consider a threat model in which adversaries wish to preserve the functionality of a page while introducing changes to it. This model assumes that adversaries can observe, delay, and modify packets arbitrarily. However, it reflects the fact that end users often have some expectation of a page's intended contents.

Under such a threat model, we hypothesize that publishers can make web tripwires effective against adversaries. Adversaries must both identify *and disable* any web tripwire on a page. Publishers can make both tasks difficult in practice using code obfuscation, using approaches popular in JavaScript malware for evading signature-based detection (e.g., code mutators [36], dynamic JavaScript obfuscation [43], and frequent code repacking [18]). Several additional techniques can challenge an adversary's ability to identify or disable tripwires on-the-fly: creating many variants of web tripwire code, employing web tripwires that report an encoded value to the server even if no change is observed, and randomly varying the encoding of the known-good representation. Also, integrating web tripwire code with other JavaScript functionality on a page can disguise tripwires even if adversaries monitor the behavior of a page or attempt to interpret its code.

Ultimately, it is an open question whether an arms race will occur between publishers and agents that modify pages, and who would win such a race. We feel that the techniques above can help make web tripwires an effective integrity mechanism in practice, by making it more difficult for adversaries to disable them. However, using HTTPS (alternatively or in addition to web tripwires) may be appropriate in cases where page integrity is critical.

4.3 Summary

Overall, web tripwires offer an affordable solution for checking page integrity, in terms of latency and throughput, and they can be much less costly than HTTPS. Finally, though they cannot detect all changes, web tripwires can be robust against many types of agents that wish to avoid detection.

5 Configurable Toolkit

Based on our findings, we developed an open source toolkit to help publishers easily integrate web tripwires into their own pages. When using tripwires, publishers face several policy decisions for how to react to detected modifications. These include: (1) whether to notify the end user, (2) whether to notify the server, (3) whether the cause can be accurately identified, and (4) whether an action should be taken. Our toolkit is configurable to support these decisions.

The web tripwire in our toolkit uses the same "XHR on Self" technique that we evaluated in Section 4. We offer two implementations with different deployment sce-

narios: one to be hosted entirely on the publisher's server, and a second to be hosted by a centralized server for the use of many publishers.

The first implementation consists of two Perl CGI scripts to be hosted by the publisher. The first script produces a JavaScript tripwire with the known-good representation of a given web page, either offline (for infrequently updated pages) or on demand. The second script is invoked to log any detected changes and provide additional information about them to the user. Publishers can add a single line of JavaScript to a page to embed the web tripwire in it.

Our second implementation acts as a web tripwire service that we can host from our own web server. To use the service, web publishers include one line of JavaScript on their page that tells the client to fetch the tripwire script from our server. This request is made in the background, without affecting the page's rendering. Our server generates a known-good representation of the page by fetching a separate copy directly from the publisher's server, and it then sends the tripwire script to the client. Any detected changes are reported to our server, to be later passed on to the publisher. Such a web tripwire service could easily be added to existing web site management tools, such as Google Analytics [17].

In both cases, the web tripwire scripts can be configured for various policies as described below.

Notifying the User. If the web tripwire detects a change, the user can be notified by a message on the page. Our toolkit can display a yellow bar at the top of the page indicating that the page has changed, along with a link to view more information about the change. Such a message could be beneficial to the user, helping her to complain to her ISP about injected ads, remove adware from her machine, or upgrade vulnerable proxy software. However, such a message could also become annoying to users of proxy software, who may encounter frequent messages on many different web sites.

Notifying the Server. The web tripwire can report its test results to the server for further analysis. These results may be stored in log files for later analysis. For example, they may aid in debugging problems that visitors encounter, as proposed in AjaxScope [24]. Some users could construe such logging as an invasion of their privacy (e.g., if publishers objected to the use of ad blocking proxies). We view such logging as analogous to collecting other information about the client's environment, such as IP address and user agent, and use of such data is typically described under a publisher's privacy policy.

Identifying the Cause. Accurately identifying the cause of a change can be quite difficult in practice. It is clearly a desirable goal, to help guide both the user and publisher toward an appropriate course of action. In our own study, for example, we received feedback from disgruntled users who incorrectly assumed that a modification from their Zone Alarm firewall was caused by their ISP.

Unfortunately, the modifications made by any particular agent may be highly variable, which makes signature generation difficult. The signatures may either have high false negative rates, allowing undesirable modifications to disguise themselves as desirable modifications, or high false positive rates, pestering users with notifications even when they are simply using a popup blocker.

Our toolkit allows publishers to define patterns to match known modifications, so that the web tripwire can provide suggestions to the user about possible causes or decide when and when not to display messages. We recommend to err on the side of caution, showing multiple possible causes if necessary. As a starting point, we have built a small set of patterns based on some of the modifications we observed.

Taking Action. Even if the web tripwire can properly identify the cause of a modification, the appropriate action to take may depend highly on the situation. For example, users may choose to complain to ISPs that inject ads, while publishers may disable logins or other functionality if dangerous scripts are detected. To support this, our toolkit allows publishers to specify a callback function to invoke if a modification is detected.

6 Related Work

6.1 Client Measurements

Unlike web measurement studies that use a "crawler" to visit many servers, our work measures the paths from one server to many clients. Like the ANA Spoofer project [8], we drew many visitors by posting notices to sites like Slashdot and Digg. Opportunistic measurements of client traffic have been useful in other network studies as well, such as leveraging BitTorrent peers in iPlane [27], Coral cache users in Illuminati [10], and spurious traffic sources by Casado et al [11]. In particular, Illuminati also uses active code on web pages to measure properties of clients' network connections, and AjaxScope uses JavaScript to monitor web application code in clients' browsers [24].

6.2 Script Injection

We found that 90.1% of page modifications injected script code, showing that scripts play a prominent (but not exclusive) role in page rewriting. Interestingly, many publishers actively try to prevent script injection, as XSS attacks have had notable impact [1, 7].

Many such efforts aim to prevent attacks on the server, ranging from security gateways [32] to static analysis [42] or runtime protection [21]. These efforts do not prevent any injections that occur after a page leaves the server, so they do not address either the modifications or the vulnerabilities we discovered.

Some researchers defend against script injection on the client by limiting the damage that injected scripts can cause. These approaches include taint analysis [38] and proxies or firewalls that detect suspicious requests [22, 26]. Each of these approaches faces difficulties with false positives and false negatives, as they must infer unwanted behavior using heuristics.

BEEP proposes a whitelisting mechanism in which publishers can inform enhanced web browsers which scripts are authorized to run on a given web page [23]. The whitelist contains digests for each script fragment on the page, and the browser ignores any script fragment whose digest is not in the whitelist. Such whitelists can prevent browsers from running scripts injected in transit, as well as XSS attacks against vulnerable proxies like Ad Muncher and Proxomitron. However, whitelists would also prevent potentially desirable script injections, such as popup blockers, unless the browser granted exceptions for known scripts. BEEP's mechanism is no more secure than web tripwires, as it could also be modified in transit over an HTTP connection, and it cannot address modifications of other HTML tags than scripts.

6.3 Integrity Mechanisms

The name for our mechanism is inspired by the Tripwire project [25], an integrity checking mechanism for UNIX file systems. Tripwire detects changes to files by comparing their "fingerprints" to a known database. Our web tripwires achieve a similar goal for web pages, where the pages clients receive are analogous to the files Tripwire checks. In both cases, tripwires detect changes, notify administrators, and have configurable policies.

Methods for tamper-proofing software or content achieve similar goals, detecting unwanted changes to programs [13]. Also, the "XHR then Overwrite" strategy described in Section 3.2.3 has similarities to secure boot mechanisms such as AEGIS [6], both of which verify the integrity of an execution environment. In contrast, we forgo costly cryptographic mechanisms for inexpensive integrity tests, much like cyclic redundancy checks [29].

7 Conclusion

Using measurements of a large client population, we have shown that a nontrivial number of modifications occur to web pages on their journey from servers to browsers. These changes often have negative consequences for publishers and users: agents may inject or remove ads, spread exploits, or introduce bugs into working pages. Worse, page rewriting software may introduce vulnerabilities into otherwise safe web sites, showing that such software must be carefully scrutinized to ensure the benefits outweigh the risks. Overall, page modifications can present a significant threat to publishers and users when pages are transferred over HTTP.

To counter this threat, we have presented "web tripwires" that can detect most modifications to web pages. Web tripwires work in current browsers and are more flexible and less costly than switching to HTTPS for all traffic. While they do not protect against all threats to page integrity, they can be effective for discovering even adversarial page changes. Our publisher-hosted and service-hosted implementations are easy to add to web pages, and they are available at the URL below:

```
http://www.cs.washington.edu/research/
  security/webtripwires.html
```


Acknowledgments

Hank Levy, Steve Balensiefer, and Roxana Geambasu provided useful feedback on drafts of this paper. We would also like to thank our shepherd, Paul Barham, and the anonymous reviewers for their suggestions. Scott Rose, Voradesh Yenbut, Erik Lundberg, and John Petersen helped prepare our servers for the "Slashdot effect," and Ed Lazowska, Dave Farber, and Keith Dawson helped us gain wide exposure. We also thank the thousands of users who chose to visit our page to learn about changes to their web traffic.

This research was supported in part by the National Science Foundation under grants CNS-0132817, CNS-0430477, CNS-0627367, CNS-0722035, and NSF-0433702, by the Torode Family Endowed Career Development Professorship, and by gifts from Cisco and Intel.

References

[1] Technical explanation of The MySpace Worm. http://namb.la/popular/tech.html, 2005.

[2] NebuAd / Service Providers. http://www.nebuad.com/providers/providers.php, Aug. 2007.

[3] The Cloak: Free Anonymous Web Surfing. http://www.the-cloak.com, Oct. 2007.

[4] Ad Muncher. The Ultimate Popup and Advertising Blocker. http://www.admuncher.com/, 2007.

[5] B. Anderson. fair eagle taking over the world? http://benanderson.net/blog/weblog.php?id=D20070622, June 2007.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, May 1997.

[7] C. Babcock. Yahoo Mail Worm May Be First Of Many As Ajax Proliferates. `http://www.informationweek.com/security/showArticle.jhtml?articleID=189400799`, June 2006.

[8] R. Beverly. ANA Spoofer Project. `http://spoofer.csail.mit.edu/`, Nov. 2006.

[9] Blue Coat. Blue Coat WebFilter. `http://www.bluecoat.com/products/webfilter`, Oct. 2007.

[10] M. Casado and M. J. Freedman. Peering Through the Shroud: The Effect of Edge Opacity on IP-Based Client Identification. In *NSDI*, Apr. 2007.

[11] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic Measurement: Extracting Insight from Spurious Traffic. In *HotNets-IV*, 2005.

[12] Chinese Internet Security Response Team. ARP attack to CISRT.org. `http://www.cisrt.org/enblog/read.php?172`, Oct. 2007.

[13] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation: Tools for Software Protection. In *IEEE Transactions on Software Engineering*, Aug. 2002.

[14] G. Combs. Wireshark: The World's Most Popular Network Protocol Analyzer. `http://www.wireshark.org/`, Oct. 2007.

[15] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[16] A. Fox and E. A. Brewer. Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation. In *WWW*, May 1996.

[17] Google. Google Analytics. `http://www.google.com/analytics/`, 2008.

[18] D. Gryaznov. Keeping up with Nuwar. `http://www.avertlabs.com/research/blog/index.php/2007/08/15/keeping-up-with-nuwar/`, Aug. 2007.

[19] Grypen. CastleCops: About Grypen's Filter Set. `http://www.castlecops.com/t124920-About_Grypens_Filter_Set.html`, June 2005.

[20] B. Hoffman. The SPI laboratory: Jikto in the wild. `http://devsecurity.com/blogs/spilabs/archive/2007/04/02/Jikto-in-the-wild.aspx`, Apr. 2007.

[21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *WWW*, May 2004.

[22] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguichi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *AINA*, 2004.

[23] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW*, May 2007.

[24] E. Kiciman and B. Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *SOSP*, Nov. 2007.

[25] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *CCS*, Nov. 1994.

[26] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[27] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *OSDI*, Nov. 2006.

[28] Microsoft Developer Network. Mark of the Web. `http://msdn2.microsoft.com/en-us/library/ms537628.aspx`, Oct. 2007.

[29] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. In *Proceedings of IRE*, Jan. 1961.

[30] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *OSDI*, Nov. 2006.

[31] J. Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[32] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *WWW*, May 2002.

[33] sidki. Proxomitron. `http://www.geocities.com/sidki3003/prox.html`, Sept. 2007.

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

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

[36] P. Ször and P. Ferrie. Hunting For Metamorphic. In *Virus Bulletin Conference*, Sept. 2001.

[37] D. Vaartjes. XSS via IE MOTW feature. `http://securityvulns.com/Rdocument866.html`, Aug. 2007.

[38] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[39] Web Wiz Guide. Web Wiz Forums - Free Bulletin Board System, Forum Software. `http://www.webwizguide.com/webwizforums/`, Oct. 2007.

[40] S. Whalen. An Introduction to Arp Spoofing. `http://www.rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf`, Apr. 2001.

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, Dec. 2002.

[42] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, Aug. 2006.

[43] B. Zdrnja. Raising the bar: dynamic JavaScript obfuscation. `http://isc.sans.org/diary.html?storyid=3219`, Aug. 2007.