Using Replicated Execution for a More Secure and Reliable Web Browser

Hui Xue Nathan Dautenhahn Samuel T. King
University of Illinois at Urbana Champaign
{huixue2, dautenh1, kingst}@uiuc.edu

Abstract

Modern web browsers are complex. They provide a high-performance and rich computational environment for web-based applications, but they are prone to numerous types of security vulnerabilities that attackers actively exploit. However, because major browser platforms differ in their implementations they rarely exhibit the same vulnerabilities.

In this paper we present Cocktail, a system that uses three different off-the-shelf web browsers in parallel to provide replicated execution for withstanding browser-based attacks and improving browser reliability. Cocktail mirrors inputs to each replica and votes on browser states and outputs to detect potential attacks, while continuing to run. The net effect of Cocktail's architecture is to shift the security burden of the system from complex browsers to a simplified layer of software. We demonstrate that Cocktail can withstand real-world browser exploits and reliability issues, such as browser crashes, while adding only 31.5% overhead to page load latency times on average, and remaining compatible with popular web sites.

1 Introduction

The near ubiquity of Internet access has put a wealth of information and ever-increasing opportunities for social interaction at the fingertips of users. Driving this revolution is the modern web browser, which has evolved from a relatively simple client application designed to display static HTML data into a complex networked operating system tasked with managing the myriad of web-based applications people use daily. Support for dynamic content, multimedia data, and extensibility has greatly enriched user's experiences at the cost of increasing the complexity of the browser itself. As a result, current web browsers are plagued with security vulnerabilities, as evidenced by Firefox, Safari, Google Chrome, Opera, and Internet Explorer reporting 374 *new* security vulnerabilities in 2009 and 500 in 2010 [18]. Unfortunately, hackers actively exploit these vulnerabilities as indicated in reports from the University of Washington [46], Microsoft [61], and Google [49, 48].

Both industry and academia have improved the security and reliability of web browsers. Current commodity browsers make large strides towards improving the security and reliability of plugins by using sandboxing techniques to isolate plugins from the rest of the browser [62, 33]. However, these browsers still scatter security logic throughout millions of lines of code, leaving these systems susceptible to browser-based attacks. Current research efforts, like Tahoma [32], the OP web browser [36], the Gazelle web browser [59], and the Illinois Browser Operating System [58] all propose building new web browsers to improve security. Although these browsers represent a vast improvement in security over monolithic commodity browsers, they require reimplementing large portions of the browser to withstand attacks. Additionally, all of these browsers exhibit fail-stop behavior when encountering a bug or attempted exploit, making them susceptible to browser crashes.

This paper presents *Cocktail,* a system that uses replicated execution of multiple existing web browsers to help withstand browser bugs and security vulnerabilities. Cocktail runs three different off-the-shelf browsers, including different plugins, in parallel with the assumption that any two of them are unlikely to be vulnerable at the same time or exploited by the same malicious page. Cocktail replicates user interaction and network requests in each of the three browsers, then votes on network outputs and browser states to detect any modifications resulting from browser bugs or web-based attacks. By mirroring inputs to three different browsers and voting on outputs, Cocktail shifts most of the browser's security enforcement into a thin and simple software layer, while reusing the mature, fast, and feature-rich implementation of existing web browsers.

Replicated execution [24, 25, 31] is conceptually simple, but it is extremely expensive to implement in practice for security purposes. To prevent replicas from falling victim to the same attack, each replica must be a *distinct* implementation of the same specification, which requires significant development resources. For example, N-version programming [30] usually requires three times as many software developers. In contrast Cocktail takes advantage of a form *opportunistic* N-version programming by using three off-the-shelf browsers directly. In Cocktail's case the specification is provided by web standards bodies [20, 13, 2, 19, 12, 3], and the implementations are represented by three major web browsing systems: Firefox, Opera, and Google Chrome.

Although modern web browsers respect common standards like HTML, HTTP, CSS, JavaScript, and the Domain Object Model (DOM), most browsers implement slight modifications to these standard web protocols. These differences require Cocktail to abstract states and to cope with non-determinism between the browsers in order to extract reliable features to vote on for security. Designing these techniques is challenging because developers did not build browsers with N-Version programming in mind.

Our experiments show that Cocktail is practical, prevents real attacks, and withstands both reliability issues and injected faults. Cocktail's replicas run in parallel, so there is almost no performance loss. While more system resources are required, multi-core systems continue to gain popularity, making this style of security practical on modern computer systems.

Our contributions are:

- Cocktail is the first system to show how to use multiple browsers as a form of opportunistic N-Version programming for improved security through replicated execution.
- We show how to abstract browser states and cope with non-determinism to enable the use of existing browsers in Cocktail, despite differences in their implementations.
- We demonstrate that Cocktail can withstand a wide range of real world attacks with little overhead to the overall browsing experience.

2 A motivating example

On July 13 2009, researchers published a public proof-of-concept exploit for Firefox 3.5 [7]. This exploit attacks a heap overflow vulnerability found in the new JavaScript just-in-time compiler that Mozilla developers introduced in Firefox 3.5. With just a single visit to a malicious web page, the attacker can run arbitrary instructions on the victim's Windows XP computer. With a few minor modifications, this exploit can also compromise the Firefox browser on Linux and Mac OS X to execute arbitrary instructions on these platforms as well [9, 8].

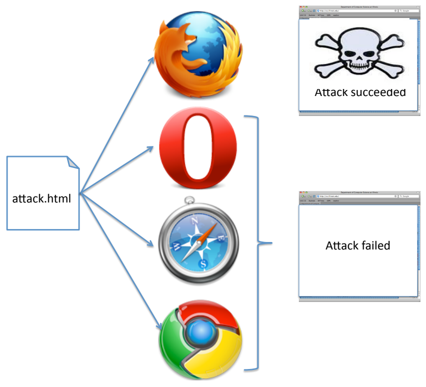

Figure 1. Results of the same exploit being served to four different browsers.

During our experiments, we ran this exploit on Firefox 3.5 and confirmed the existence of the successful attack. However, when we opened this same malicious page using Internet Explorer, Safari, Chrome, or Opera, these browsers safely avoid the attack, as illustrated in Figure 1, because none of these browsers had this same bug. Surprisingly, even older versions of Firefox on Windows XP were unaffected by this attack. This difference in processing the malicious payload motivates our architecture that uses three different browsers to withstand attacks.

3 Problem statement, threat model, and assumptions

Cocktail focuses on providing protection for browsers and plugins by running different browser implementations in parallel and detecting any inconsistencies between these browsers. We use three distinct off-the-shelf browsers and Flash player plugins.

We focus on attacks on browsers themselves where an attacker can compromise a browser vulnerability and control the browser.

In our threat model, we assume that an attacker has taken over a web site and can serve arbitrary data to Cocktail. This data can come via a web site that the user visits directly, or it can come indirectly through a web site that serves off-site network resources, like ads. We assume that this malicious data can result in a compromise to any of the browser replicas executing within Cocktail, which implies that the attacker has the ability to execute arbitrary instructions with the full privileges of the replica.

Currently, Cocktail does not provide any protection against bugs in web-based applications. Fundamentally, bugs like cross-site scripting [35] and cross-site request forgery [65] result from bugs in the *server-side code.* The resulting attacks operate within current browser security policies, putting them beyond the scope of Cocktail.

We trust the layers upon which Cocktail is built. This includes the network component and the visual voter in UI component, the libraries that the output component uses, the Systrace tool we use to build browser sandboxing, the underlying operating system, and the underlying hardware. If an attacker can compromise any of these layers, they are likely to be able to defeat Cocktail.

4 Design principles

Cocktail's design is guided by the following three principles:

1. *Use different existing browser implementations for diversity.* We use the high performance, feature rich, standards compliant – yet different – implementations of existing browsers. This balance provides improved security without sacrificing performance or functionality.

2. *Avoid changing the replica's implementation.* Adhering to this principle provides two key benefits. First, avoiding changing replicas enables Cocktail to use closed-source browsers such as Opera. Second, avoiding implementation specific details about the replicas enables Cocktail to remain stable across updates of individual browsers.

3. *Focus equivalence testing on security features.* The browsers we use are off-the-shelf products from different providers with millions of lines of code in each of them. Though certain standards, such as HTTP and HTML, are available, it is still hard to find perfect abstractions to dictate their equivalence. However, if we focus on security features, we can find a layer of abstraction that omits large amounts of browser-implementation details while still preserving strong security guarantees.

In designing Cocktail the most significant issue we address is that of security state selection. The decisions we make with respect to this impact both the overall security improvements and architecture of Cocktail. In Cocktail we strive to select state representations that are not too implementation specific, yet maintain potency in deterring real-world attacks. The fundamental method we use in Cocktail to provide security is to require a majority vote before any given state is permitted to persist in the browser. If a majority is not obtained then the given action producing the invalid state is rejected.

To see this more clearly take, for instance, the state abstraction of network requests. If at least two of the three replicas attempt to fetch an image at location `foo.com/image.jpg`, then Cocktail will fetch the resource and return the data back to the requesting replicas. However, if none of the replicas request the same image then Cocktail will deny the request because it does not reach consensus. By using network requests as a key state abstraction Cocktail effectively thwarts all attacks that require additional network resources to succeed.

Conceptually, we view Cocktail as a black box with two channels of communication in and out. One channel, the display, represents the agreed upon visual output from the three replicas. The other channel, network requests, represents the agreed upon network communications by the replicas. Given our assumption that at most one replica is compromised, reaching consensus on a network request implies that at least one uncompromised browser made this individual network request, and reaching consensus on the display implies that the browser, from the user's perspective, produces output that is equivalent to an uncompromised browser. Thus, the browser's observable behavior is consistent with that of an uncompromised browser. If Cocktail's behavior is consistent with an uncompromised browser, then it has effectively thwarted the effects of potential attacks. In other words, if Cocktail looks like an uncompromised browser, and it acts like an uncompromised browser, then it is an uncompromised browser independent of what is inside of the black box.

Cocktail's state abstractions provide rigorous defenses against attacks, but have the potential to incur false positives on non-malicious sites. The problem is that there are several valid reasons for a given browser state to deviate from the other two replica states. For example, ad networks provide randomized content to end users. This content represents a major class of non-determinism that should not be considered an indicator of malice. Therefore, one of our primary design considerations is to identify and eliminate sources of non-determinism in each browser so that the requested resources are as consistent, avoiding false positives. In Section 9.4 we show that our approach handles this and other forms of non-determinism while maintaining compatibility with existing websites. Many of the design challenges faced by Cocktail stem from this problem, and are discussed in Section 5.

5 Cocktail design challenges

Cocktail is comprised of a UI component, a replica component, and a network component, as shown in Figure 2. The UI component is responsible for providing

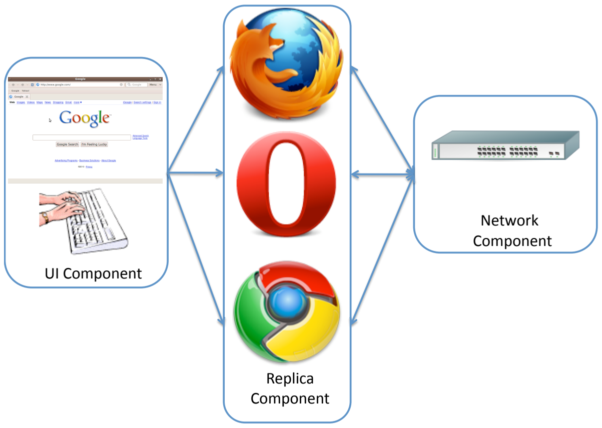

Figure 2. Cocktail overview.

the interface between the user and Cocktail, routing user input to each replica, and voting on the display states of each replica. The replica component maintains each browser replica, which all run in sandboxed environments. The network component is responsible for handling network requests from the replicas and voting on network requests. We describe the details of these components in Section 6.

The first challenge in Cocktail is defining appropriate state abstractions to enable Cocktail to behave like an uncompromised browser even if an attacker compromises one replica. This challenge is the basis for the security assurances in Cocktail. The second and third challenges are to remove enough sources of non-determinism from the server and in our browser replicas to enable Cocktail to process web sites correctly. For these challenges Cocktail does *not* need to remove all sources of non-determinism. Cocktail only has to remove enough non-determinism so that Cocktail can reach a consensus for the states encountered on a wide range of web sites because of the natural redundancy inherent in our system.

5.1 Challenge: Browser state abstraction

It is hard to abstract meaningful common states from different software. Most works [28, 63, 40] from Byzantine Fault Tolerance community provide state abstraction examples for replicas running the same software. Although this type of replication does improve the reliability and availability of these systems, they provide few security improvements because software exploits manifest as correlated failures [54]. Other systems combine independent implementations and rely on clear abstractions of states. For example, BASE [53] combines different database implementations and considers the well-defined database data to be the state of the system, and EnvyFS [23] combines different file system implementations and considers the file system data to be the state of the system. However, as distinct UI oriented software platforms without table or inode-like natural state representations, abstracting states for Firefox, Google Chrome, and Opera is hard.

The goal in defining browser states is to find states that are (1) meaningful to users, (2) low-level enough to detect a wide range of attacks, and (3) high-level enough to be uniform across different browser implementations.

5.2 Solution: Network and on-screen states

We define two states for all browsers: network and display states. Network states include the HTTP requests and headers, and browser cookies that are included in network requests. Browser cookies are a mechanism for HTTP servers to store key-value pairs on client machines persistently. Browser cookies are a first-class part of modern browsers and are generally consistent across different browser implementations. Thus, including browser cookies in our network state abstraction was an easy design decision.

Cocktail also uses the visual output of the browser as a state because the it is the most meaningful state to users. The browser display can also indicate severe security problems, especially when the attack requires user interaction. For instance, in Sept. 2009, a vulnerability [10] found in Firefox 3.0.x(x<14) pops up a dialog trying to coax users into installing a malicious PKCS11 module. This vulnerability does not affect Safari, Google Chrome, Opera or Internet Explorer and they showed no dialog box. This is a typical example where a visual difference in one browser can indicate the existence of an active security exploit.

Although web standards dictate how a browser should render and display a web page, small implementation differences can cause visual discrepancies. For example, browsers might use different font libraries, causing small discrepancies in how the browser displays text. These discrepancies make it difficult to do a pixel-by-pixel comparison of two browsers. Other researchers have applied vision algorithms to web browsers to quantify visual similarity [57], they use the SIFT algorithm [41] for comparing the rendering results from the same browser for offline analysis. However, the SIFT algorithm takes tens of seconds to compare browser displays, making it unsuitable for real-time analysis. The key attribute that makes SIFT attractive is that it works on more coarse-grained visual information in an image, allowing SIFT to match similar pages despite small differences in the rendering of the page.

In Cocktail we try to blend the high-level feature extraction properties of SIFT with an algorithm that can run in real time. In our algorithm, we apply Gaussian smoothing methods to mute small pixel differences in browsers and we use Canny edge detection [27] to pull out higher-level features of each replica's display. These techniques are computationally efficient, allowing us to take periodic snapshots of each replica's display, and we can detect coarse-grained changes to a page, like an attacker who overlays content on top of a web page. However, our technique is unable to detect small changes to a web page, like an attacker who changes a few words on an existing web page.

5.3 Challenge: Server-side non-determinism and side effects

Our second challenge is coping with server-side non-determinism and side effects. Each time a browser makes an HTTP request, the server can return different results. For example, each time one visits a news site the main page will include any recently updated news stories. Furthermore, some HTTP requests have side effects, like sending a friend a message on facebook. These issues cause problems for Cocktail because we run three browsers in parallel, so we must ensure that each browser sees the same network data to create the same browser states, and we must ensure that HTTP requests avoid inducing unanticipated side effects and maintain current browser semantics.

Our goal for coping with server-side non-determinism and side effects is to mask these effects from our browser safely and efficiently *without* modifying the implementation of the browser replica.

5.4 Solution: Local web proxy

Our solution to this challenge is to implement a local web proxy that interposes on all network connections made from our browsers. This local proxy runs as a separate user-mode process and makes network requests on behalf of the Cocktail replicas as shown in the network component in Figure 3. To cope with server non-determinism, our local web proxy buffers the results of HTTP requests in its cache component and uses this buffered data to ensure that all replicas receive the same result for all requests. To avoid inducing side effects on the server, the local web proxy makes only a single outgoing HTTP request for all equivalent requests from the replicas.

To handle encrypted HTTPS traffic, we install our own self-signed certificate in each of our browser replicas to implement a man-in-the-middle proxy, similar to the SSL-MITM proxy by Boneh, Inguava, and Baker [14]. The HTTPS proxy establishes one encrypted connection with the web browser and another encrypted connection with the requested HTTPS site while relaying clear text data in between the two encrypted channels. As a result, the proxy is able to read the unencrypted web traffic and replicate it to all of the browser replicas.

5.5 Challenge: Client-side non-determinism

Our third challenge is avoiding client-side non-determinism that can cause our browsers to generate different abstract states for the same HTML and JavaScript inputs. Based on our observations, browsers exhibit three types of client-side non-determinism.

First, standard JavaScript functions may return non-deterministic or random values. The two most common examples of these functions are `Math.random()` and `Date.getTime()`. In practice, web pages use these functions to select random advertisements for a given content pane.

Second, some browsers include standard functions that return implementation specific results, and some browsers include non-standard JavaScript interfaces to provide extended functionality. For example, the `navigator.userAgent` property will disclose the identity of the browser and the `window.opera` object exposes Opera-specific functionality not found on other browsers. `navigator.language` also differs slightly between Opera and Firefox. Browser locale setting differences make Opera indicate English as "en" whereas Firefox and Chrome both indicate "en-US". Similar differences also exist for `document.characterSet` which has value "utf-8" on Opera whereas "UTF-8" on Chrome.

Third, users interact with the browser via input devices, such as the mouse and keyboard. These user interface actions induce computation in the browser and can result in non-determinism if the system delivers UI events to the incorrect UI widget or with varied timing.

Our two goals in coping with client-side non-determinism are to remove enough sources of non-determinism to enable Cocktail to render a wide range of web pages correctly, and to remove these sources of non-determinism *without* modifying the implementation of the replicas or eliminating browser features.

5.6 Solution: Browser extensions and configurations

Our primary mechanism for coping with client-side non-determinism is to overwrite non-deterministic functions by injecting JavaScript into every page via browser extensions. The goal is to eliminate non-determinism

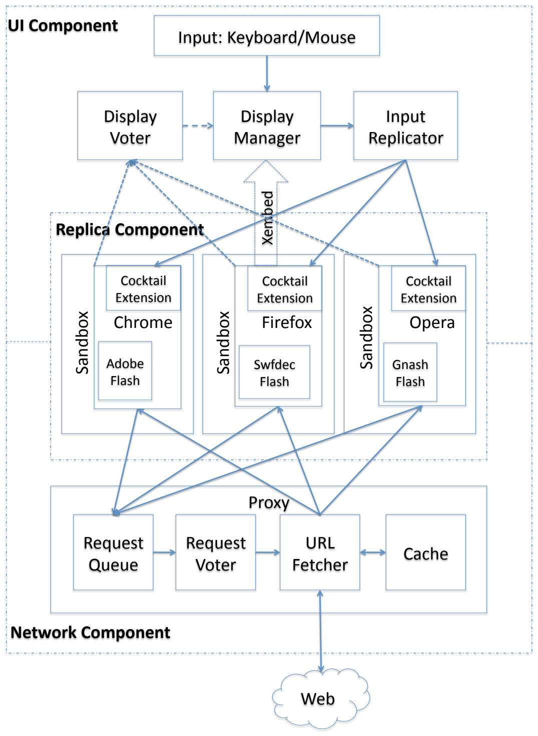

Figure 3. Cocktail architecture.

across each replica. Browser extensions are a form of browser extensibility that gives users the ability to extend and modify pages that they visit. One common example of a browser extension is NoScript for removing JavaScript from select web pages. Extensions are written in JavaScript and have access to a wide range of states and events in the browser. We use extensions in each of our replicas to inject JavaScript into all pages that we visit to overwrite and normalize nondeterministic features of the browser. We implemented extensions for each of our browser replicas as shown in the replica component in Figure 3.

One important property of our injected JavaScript is that we do *not* trust this code – it serves solely as a mechanism for preventing false positives.

Injection by browser extensions works for random functions, such as `Math.random()` and `Date.getTime()` as well as other functions provided by the `Date` object. In order to make `Date.getTime()` deterministic we discretize time and make the clock tick in three second intervals. In this way each of the replicas generate the same return value for each call to `Date.getTime()`.

This overwriting technique also enables us to remove browser-specific attributes whose existence indicate the exact browser, such as the `window.opera` object, by replacing them with undefined objects. To normalize the `navigator.userAgent` property we update configuration values for Chrome and Opera to make them all appear as if they were Firefox. We also change browser locale configurations to solve the `navigator.language` and `document.characterSet` value differences among browsers. The net effect of these modifications is that Cocktail has a reduce set of functionality as compared to a conventional browser, but this subset is large enough in practice to handle current web pages correctly based on our experience in Section 9.4.

```
<iframe src="
http://www.adfusion.com/Adfusion.PartnerSite/ca
tegoryhtml.aspx?userfeedguid=948fbed8-69ae-4659
-b3c1-b9863e5ab24e&clicktag=http://ads.bluelith
ium.com/clk?2,13%3B738a290b44284f0c%3B12c3705b5
e2,0%3B%3B%3B1410192806,5jBaAEmJEgC.5m.........
...AAAAAA=,http%3A%2F%2Fglobal.ard.yahoo.com...
........0%2F%2A%24,http%3A%2F%2Fadjax.flickr.ya
hoo.com%2Fads%2F792600119%2Flrec%2F,&CB={REQUES
TID}

*****URL missing " here*****

width="300" height="250" scrolling="no"
frameborder="0" marginheight="0"
marginwidth="0"></iframe>
```

Figure 4. iframe source URL not quote-closed

Researchers study user action tracking in areas such as collaborative browsing [64, 52, 22, 42] with commercialized services [16] available. Recently it also has been applied for security [39, 11, 21] and web testing purposes [44, 17]. In Cocktail, we use similar techniques to record user interface events, broadcast these events to all replicas, and replay these events correctly at each replica to induce the proper computation on each browser. To accomplish this Cocktail must correctly identify the exact event that occurred and the object or element it occurred upon. Then Cocktail translates this event to the replica browsers for replication.

Although browsers adhere to most standards, there is room for some implementation specific interpretations of the standards, especially when dealing with buggy HTML. Figure 4 shows an iframe used by `flickr.com` for an advertisement. Its source URL is missing the ending quote mark. Firefox and Opera both automatically try to complete the quote and issue the HTTP request whereas Chrome drops the request. Unfortunately, these types of errors are difficult for Cocktail to compensate for and can prevent Cocktail from making legitimate network requests if all three replicas interpret the data differently.

6 Cocktail implementation

In this section we describe each of the internal Cocktail components in detail. Figure 3 provides more internal details for each component.

6.1 UI component

The UI component presents the user with a single browser window that dynamically embeds one of the replica browser's display. The input replicator replicates the user's inputs, such as mouse clicks and keystrokes, to all of the replicas. For example, when the user loads `facebook.com`, all three browser replicas load the Facebook homepage, but the display manager presents only one of them to the user. After the user types in his or her user name and password, he or she will login in all three browser replicas, clicking on a picture causes all three replicas load the same picture, but the user only sees one replica's output. During these actions, the display voter executes in the background, continually comparing the display of the three browsers to detect suspicious display differences among the replicas. If a difference is detected the display voter alerts the display manger, which will then thwart the malicious browser.

In general, we speculatively select a single browser to serve as the display replica, which we refer to as the *control replica*. If the control replica fails to meet consensus then the display manager switches to another replica, thus, enabling the user to continue browsing. We use XEmbed and Qt widgets to implement our display manager and ImageMagick libraries to capture browser display for the display voter. We use OpenCV library to implement image processing in display voter. More details are given in Section 8.

6.2 Replica component

The replica component maintains the three distinct browsers, executing each one inside an OS-level Systrace sandbox [47] that provides an extra layer of isolation between each browser replica and the rest of the system. OS-level sandboxes prevent replicas from communicating with each other and from accessing unauthorized system resources. Although OS-level sandboxes can help limit the effects of an attack, they operate on OS-level abstractions, such as files and sockets, which are too low level to enforce browser-level security policies that operate on high-level browser abstractions, like cookies and HTTP requests. In Cocktail, the main benefits of our OS-level sandboxing are to force the replicas to use the Cocktail system and to limit access to sensitive OS states.

We also apply replication to browser plugins. For Flash, we use Swfdec for Firefox, Gnash for Opera, and the Adobe Flash Player for Chrome, giving us diversity both for the browser itself and for Flash plugins.

6.3 Network component

The network component's primary responsibility is to verify network requests. The network component accomplishes this task by interposing on network requests to enforce a majority vote for each resource requested by the replicas. The network manager is comprised of a request queue, request voter, URL fetcher, and cache. As Figure 3 shows, the request queue receives all network requests from the replicas. These requests are then voted upon by the request voter. If a $\frac{2}{3}$ majority vote is obtained for a given resource then it is considered validated and a single copy of this resource is obtained by the URL fetcher. The URL fetcher stores the resource in the cache and serves it to each replica that requested the resource. The cache is necessary due to performance and correctness considerations. Instead of having a timeout period for each request, Cocktail immediately fetches an URL once two requests have been made for a given resource – indicating a majority vote. This means that the third replica may still request the resource, at which time the URL fetcher will serve the page from the cache. The network component is implemented with 4704 lines of Java code.

7 Discussion

This section discusses the ramifications of some of the design decisions that we make in Cocktail.

One potential disadvantage of making all of the replicas appear to be the same browser is that Cocktail has only the functionality found in all of our replicas. Fortunately, most pages we tested work correctly in Cocktail, indicating that the least common subset of our three replicas is sufficient for today's popular pages. Even more advanced features, like browser extensibility, are still supported in Cocktail. All of our browsers support extensions and the extensions we tested that work in Chrome and Opera also work as Firefox Greasemonkey scripts. Unfortunately, the Firefox extension API is more extensive than the extension API for Chrome and Opera and some Firefox extensions will not work in Cocktail.

One problem with N-Version or Byzantine Fault Tolerant systems is that attackers can exploit vulnerabilities in two or more replicas causing the attack to behave like a correlated failure. Although browsers do tend to contain many security vulnerabilities, browser vendors tend to patch these vulnerabilities quickly, narrowing the window for this type of correlated attack. According to a recent report from Symantec [18], in 2009 the average window of exposure for a vulnerability is less than one day for Firefox and Opera, one day for Internet Explorer, two days for Chrome, and 13 days for Safari. In 2010 the average window of exposure is less than one day for Safari and Chrome, one day for Opera, two days for Firefox and four days for Internet Explorer. Keeping the Cocktail replicas up-to-date is paramount to avoiding this type of attack.

8 Voting

In general, our voting mechanism has two main tasks: validate outgoing network requests and check visual states. For network requests, Cocktail checks for equivalence among all of the replica's HTTP requests. HTTP requests contain HTTP header information, cookies, the payload, and the URL of the request. HTTP headers contain information about the client, such as types of content the browser is willing to accept, cache controls, and the user-agent to let servers know what type of browser is making the request. The only items we check in the header are items that we perceive to be security critical: HTTP authorization credentials and the referrer string to signify the page that originated the request. Our algorithm ignores all other items in the header because they tend to depend heavily on the browser and because they have little bearing on the security properties of the network request.

In addition to checking authorization credentials and referrer strings, our algorithm also checks equivalence for cookies, for the full payload of the request, and for URLs. Any differences in cookies or payloads will cause the voting algorithm to consider two network requests as being different. For URLs, our algorithm does a complete check for equivalence except for query strings. Cocktail compares URL query strings using a case-insensitive string matching function to compensate for some of the implementation specific capitalization that we observe.

When designing our visual state comparison algorithm, we try to balance the desire for capturing detailed visual information from each replica with potential false positives due to small implementation differences. Plus, our algorithm has to be fast enough to run in real-time.

In the display voter, our display capture and image processing methods reveal the structure of the page by extracting the position information of its rectangular components. To identify rectangular components, we apply a Gaussian smoothing filter to blur the image. Then, we use a Canny edge detector to extract horizontal and vertical edges from the blurred image to identify rectangular structures in the display. The display voting algorithm examines the number, size, and relative position of these rectangles. For instance, if there is a big rectangle in the center of only one browser suggests there could be a dialog window displayed in only one browser. In addition to rectangle detection, our algorithm also checks pixel color information by running the image through a filter to mute any small color differences that may occur naturally in different browsers.

By checking these high-level features, we can detect large changes to the web page, like an attacker causing a popup dialog box to cover part of a web page, or an attacker making large changes to the content of a compromised replica. Also, the particular algorithms we use have efficient implementations available from the OpenCV library, and our visual state comparison algorithm adds effectively no overhead to our system.

For visual states, checking can be expensive, so currently Cocktail scans each of the replicas every 1.5 seconds to check for equivalence among the three replicas. If Cocktail detects a divergent replica, it shows the user a display from one of the replicas in agreement.

Because Cocktail replicates inputs for browser replicas, three browser replicas are enough to tolerate one faulty browser. This simplification is possible because the order of operations on each replica does not rely on distributed decision-making process, which requires $3f + 1$ replicas to tolerate f faulty ones.

9 Evaluation

This section describes our evaluation of Cocktail. In our evaluation, we measure Cocktail's ability to withstand attacks, ability to withstand replica crashes, Cocktail's compatibility with existing websites and the overhead.

We run all experiments on a 2.80GHz Intel Core i7 machine with 8GB of memory and 220GB SATA hard drive. For our performance experiments we use Ubuntu 10.04, and Firefox 3.6.6, Opera 10.60, and Google Chrome 7.0 as Cocktail's replica browsers. For our security experiments, we use a vulnerable browser version as one of our browser replica on a corresponding operating system.

9.1 Performance

To measure performance, we measure the page load latency time for Cocktail and compare to the page load latency times for each of our replicas running alone. We test the page load latency time for seven popular pages as shown in Figure 5. The reported page load latency time is the average of ten runs for each site.

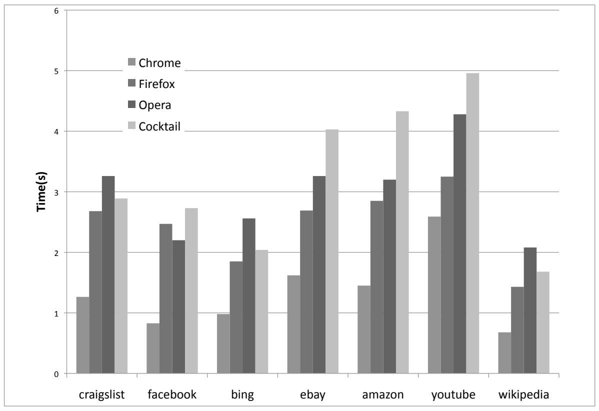

Figure 5. Page load latency comparison for Cocktail and individual browsers.

Page load latency is defined as the time between when a user initiates a visit to a new web page and when the browser "onload" event fires. In our experiments we use our Firefox replica as our display replica and measure the page load latency times for that individual replica. Figure 5 shows the average page load latency time for our Cocktail display replica. We also show the page load latency time for Firefox, Google Chrome and Opera running individually outside of Cocktail on the same hardware and same operating system. For our Firefox replica to reach the "onload" event means that for each network request at least one other replica has requested the same network resources as our Firefox replica.

Overall, Cocktail adds 31.5% overhead on average to the page load latency for the seven sites we tested comparing to Firefox running alone. `youtube.com` was the slowest site – Cocktail added 52.6% overhead , which was the largest percent slowdown in our experiments. This slowdown is because the order of network requests cause Firefox in Cocktail to block for some network requests.

9.2 Reliability

We designed Cocktail to improve reliability by allowing the system to remain running even if one of our replicas crashes, similar to other replicated execution systems [24, 23, 28, 53, 40, 63]. To measure Cocktail's ability to withstand browser crashes, we trigger a bug in our Firefox replica, a bug in our Opera replica, and we inject faults in all of our replicas using the DieHard fault injection tool [24]. Table 1 shows that Cocktail successfully masks the reliability bugs in Firefox and in Opera, and for injected faults on each of our replicas.

Fault type	**Individual browser**	**Cocktail**
CVE-2009-0071 [6]	Firefox 3.0.6 crash	masked
Opera Crash [15]	Opera 10.10 crash	masked
Fault Injections	Firefox Crash	masked
Fault Injections	Opera Crash	masked
Fault Injections	Chrome Crash	masked

Table 1. Reliability test results for Cocktail.

9.3 Security analysis

To measure Cocktail's ability to withstand attacks, we exploit our Firefox replica using four real-world exploits that represent four broad classes of attacks.

First, we tested Cocktail against a category of attacks that requires user's interaction to succeed. We created a page containing JavaScript code that exploits CVE-2009-3076 [10] targeting Firefox 3.0.x(x<14) on Ubuntu 8.04. Our attack page entice the user to click on a button in a dialog window. When the user clicks on this button, this attack installs a malicious PKCS11 module into the browser, compromising the integrity of the cryptography. In our experiment, the dialog did not pop up in our Opera or Chrome replica, because they did not contain this vulnerability. Our display voting algorithm catches the dramatic visual differences between replica displays.

Second, we tested Cocktail against a remote code execution attack category that runs automatically, without user interaction. We created a web page containing CVE-2009-2477 [9] heap overflow attack exploiting Firefox 3.5 on a Ubuntu 8.04 machine. We crafted the heap overflow attack to launch our payload code that tries to download a trojan file from a web site. Although our Firefox replica did attempt to download the file, the Chrome and Opera replicas never attempt to download this file because they were not vulnerable, thus Cocktail squashes this malicious HTTP request.

Third, we investigated Cocktail's ability to withstand DOS attacks. We crafted a DOS attack [5] against Firefox 3.0.4 on Ubuntu Linux 8.04. This attack causes the browser to run into an endless loop blocking access to the UI. Although the Firefox replica in Cocktail stops responding, the Opera and Chrome replicas continue to run, defeating this attack successfully.

Fourth, we tested an attack that uses a browser vulnerability to bypass the same origin policy. We tested Cocktail against a cookie stealing vulnerability CVE-2007-0981 [4] targeting Firefox 2.0.0.1. Interestingly, Cocktail defeats this attack because Cocktail uses a proxy for network requests. This specific attack exploits the inconsistency between location.hostname and DNS look up results when there is a null character in location.hostname in Firefox 2.0.0.1. Specifically, a site with "evil.com\0x00www.victim.com" will be treated as "evil.com" for DNS look up, whereas the site is instead treated as a subdomain as "victim.com" by the browser. This is because Firefox 2.0.0.1 treats null characters as part of local.hostname whereas underlying C/C++ implementing DNS look up code treat null as the end of the string. Therefore, the browser can send cookies belonging to "victim.com" to "evil.com". However, Firefox in Cocktail uses our proxy for network connections and our proxy did not have this vulnerability. However, even if a cookie stealing attack succeeds in one replica, we believe the network voting algorithm of Cocktail will detect it by observing different network requests from the replicas.

9.4 Compatibility

An integral component of evaluating Cocktail is to assess its compatibility with existing websites. It is important to consider compatibility because, in Cocktail, browser functionality is modified, which may break web developer assumptions about how the browser will interpret code. Additionally, there could still be non-determinism left in the browser that is not handled, which could lead to incompatibility issues. Therefore, in this section we analyze each of the major client side modifications of Cocktail and discuss the potential problems arising from them. Additionally, we describe the experimental results of testing Cocktail on the top 100 websites (as defined by `alexa.com`). Our goal is to provide ample evidence that Cocktail is usable as a primary browser with today's websites.

To evaluate Cocktail's compatibility, we discuss how the modifications to client side functionality lead to potential incompatibilities. In general, incompatibilities manifest to the user as either missing or buggy dynamic functionality (e.g., JavaScript or AJAX), or web resources failing to load because of a lack of consensus by the replicas (e.g., non-determinism in URL formation). We divide dynamic functionality into two classes: user-interactive and non-user-interactive functionality. Non-user-interactive functionality is the activity present on a page without any user interaction (e.g., updating news feeds). We make this distinction because it allows us to separate compatibility issues due to UI replication from other client side modifications. Resource related problems are characterized by missing web page content. The client side modifications can impact how URLs are generated for resource requests and, if inconsistent, will generate valid requests that are subsequently rejected by the Cocktail voter due to lack of consensus.

In general we found Cocktail to be comparable to existing web browsers in its ability to successfully interact with and render web pages. Table 2 displays an overview of our experimental results, which were obtained by manually visiting and testing the top 100 global websites as identified by `alexa.com`. We assess each website with respect to dynamic functionality and resource related issues, and report results with regards to the potential underlying causes. Each particular problem and results are discussed in detail in the following sections. In addition to manually testing Cocktail we also examine Cocktail's score on the common web standard test `acid3` [1].

9.4.1 Methodology

In this evaluation we test the top 100 websites for both dynamic functionality and resource related competencies. For each site we compare the results of evaluating Cocktail to that of a standard Firefox browser. If the results diverge from Firefox's results then we determine a fault for the given site test and mark it down as such. To test for non-user-interactive dynamic functionality we view the page for ten seconds, verifying that the functionality matches that of an unmodified Firefox version. To test for user-interactive dynamic functionality we examine Cocktail's ability to replicate dynamic functionality as a byproduct of mouse clicks and keyboard input.

In terms of resource related issues we analyze web pages with respect to the resources that it successfully fetches from the Internet. A web page is comprised of an initial landing page (e.g., an HTML file) and then a set of resources that are downloaded for insertion into that page (e.g., hyperlinked content, images, videos, etc.). We analyze Cocktail's ability to obtain all resources in a page and render them on that page. In our experiments we replace pornography sites with the next highest ranked sites from `alexa.com`.

9.4.2 Results and Discussion

Synchronized Deterministic Time As discussed in Section 5.5, to handle non-determinism due to time dependent functionality we discretized time. The primary issue related to compatibility here is that a large percentage of websites use JavaScript time functionality. In general we observed two types of uses of JavaScript time: to create randomness for seeding dynamic content (generating dynamic links to ads), and for dynamically updating page content via JavaScript/AJAX. Problems occur on any code that employs time intervals less than

Evaluation Type	**Result**
Dynamic Functionality	
Non-user-interactive	100%
User-Interactive	
Keyboard Input	99%
Mouse Clicks	99%
Web application replication	
`live.com`: login and check email	✓
`amazon.com`: login, navigate, browse items, and view cart	✓
`yahoo.com`: web search, interact with Javascript, and navigation site	✓
Resources	
Site content acquisition	98/100 sites
`paypal.com`	43/46 web resources
`nytimes.com`	85/86 web resources
Ad content acquisition	96/100 sites
Rendering	100%
Web Standards Testing	
Acid3	100/100, 99/100, 100/100 (Chrome, Firefox, and Opera Replica scores)

Table 2. Results Summary. Results are given in terms of successes out of the total number of objects evaluated unless otherwise stated.

Cocktail's discrete time interval of three seconds. Cocktail's deterministic time functionality will allow time based changes to occur, but only in three second intervals, which manifests in delayed functionality. In order to improve the resolution of `Date.getTime()` we can employ a more robust solution of a distributed clock.

In order to evaluate the impact of deterministic time in Cocktail, we analyze the set of functionality that a given website exhibits without user interaction. When including interaction the problem or associated set of issues are related to UI replication, which will be discussed in a subsequent section. Our results indicate that sites using JavaScript time functionality for seeding content do not experience any problematic behavior, and work fully with Cocktail. Furthermore, for seeding dynamic content, Cocktail's time methods provide consistent results in all three browsers and maintain full compatibility. In terms of the second area of interest, JavaScript/AJAX dynamic updating of page content, Cocktail successfully handles all top 100 sites.

Deterministic *Math.random()* Problems arising from modifying *Math.random()* include any type of functionality that requires randomness to function properly. In our evaluation, in which we analyzed non-user-interactive JavaScript and content acquisition, Cocktail did not incur any problems emerging from modifications to the *Math.random()* function in either content acquisition or dynamic page functionality. This does not rule out the potential for issues, but shows the viability and robustness of Cocktail at handling top websites. Recall that the goal of these modifications is to eliminate non-determinism across the replicas, not necessarily non-determinism with respect to the website code. As long as the replicas all deterministically select the next random number they will be consistent, maintaining compatibility, but still present random numbers to the function callers.

Browser Identification Normalization In Section 5.5 we discussed modifications that normalize replicas so that each browser appears to be the same browser, which minimizes browser specific functionality. The types of problems that could occur include issues with content acquisition, as well as modified dynamic functionality. For example, website code could traverse a specific Firefox only code path resulting in different output based upon the `navigator.userAgent` property. In practice Cocktail did not experience any reduced functionality on the top 100 sites due to browser normalization.

UI Replication The goal of the replication system is to take user input from the control replica and invoke the same actions in the other two replicas. The control replica sends the event information to the other replicas, which then invokes the specific action. The replication, if incorrectly working, can be a source of both dynamic functionality and content acquisition errors.

Evaluating the replication system is challenging because it is infeasible to traverse all code paths on a given website. Furthermore, replication has complex interactions with elements in the page. If a specific event replicates correctly on a given page there is no guarantee that it will correctly replicate on another webpage. Therefore, we evaluate replication by verifying that a minimal subset of events are correctly replicated in each of the replicas. The specific events we test on all 100 sites include keyboard input and mouse clicks. Most user interaction events can be represented as one of these, and as such we feel as though this subset is representative of a major portion of dynamic interaction between the user and a webpage. Based on this subset of functionality Cocktail faithfully replicates events and dynamic functionality on 99 out of 100 sites (sina.com.cn being the lone site missing functionality).

It is important to note that we only tested the front page of the top 100 sites. Therefore, in an effort to show the efficacy of Cocktail's replication we evaluate Cocktail's ability to execute common web application tasks. We tested three sites `live.com`, `amazon.com`, and `yahoo.com`. For the first two we successfully login and execute common tasks for the particular web application. On `yahoo.com` we browse news items and interact with pages in an effort to examine the viability of Cocktail, and find that we can perform general navigation without error. Table 2 displays the exact tasks.

Voting Recall that the voting mechanism rejects any web request that does not have a majority vote. If any two of the replicas either fail to make a given request, or in some way have different URLs for the same content area (e.g., ad, image, video) then that web resource will be missing from the rendered page. The source of rejections is client side non-determinism, which are those things that Cocktail does not handle. The results discussed here are due to non-determinism in the replicas after Cocktail modifications, and not from the client side sources as discussed in the previous section.

Our experiments reveal that a primary source of non-determinism causing resource fetch failures is ad content areas in web pages. Ad content by nature is non-deterministic, and in the cases where content is missed, the non-determinism comes from ads that are being generated by an ad server of different origin than the webpage being viewed. Although, this was the source of Cocktail's greatest problems, Cocktail is successful at correctly obtaining all resources for 94 out of 100 sites evaluated. Note that the only missing content on these sites is ad content and not all ads failed to load for the six sites.

Site content was another type of resource experiencing fetch issues. Out of the top 100 sites only two sites, `nytimes.com` and `paypal.com`, experienced a site content related miss. `nytimes.com` successfully loads 85 out of 86 resources indicating that most content and features are still available on the page. The one resource that fails on `nytimes.com` is a flash video player. `paypal.com` succeeds on 43 out of 46 resources, and is incompatible due to non-determinism in the way that flash plugins identify themselves to the browser. Normalization of the flash players would eliminate this issue. Table 2 displays our results on content acquisition analysis.

10 Limitations

In previous sections we show that Cocktail can withstand several real-world attacks, and we argue that it will help prevent a wide range of browser-based attacks. In this section we discuss ways an attacker can avoid detection and carry out attacks on Cocktail.

An attacker can evade Cocktail by exploiting a vulnerability in a shared system service, a shared library, or by exploiting vulnerabilities in two or more replicas. Although Cocktail uses different browsers for replicas, these browsers share the underlying operating system and link to some of the same libraries, like `libc`. Exploiting a bug in the OS or a library used by all replicas will likely have similar effects in all of the replicas and Cocktail would be unable to detect this type of attack. In addition to shared resources, and attacker could evade Cocktail by exploiting two separate bugs in different replicas to cause them to do perform the same state transitions and produce the same outputs. Fortunately browsers tend to patch vulnerabilities quickly [18].

Cocktail abstracts some implementation specific details to compare visual states for our replicas and to account for some implementation-specific artifacts in network requests. Because of these abstractions, the attacker has some room to modify states in a meaningful way that Cocktail will still consider equivalent. For example, an attacker could change a few key words on a web page to hide malicious activities, and the Cocktail visual detection algorithm will still likely consider these two displays to be equivalent. However, this limitation does still place significant restrictions on attackers by preventing them from making large visual changes to a web page.

Finally, Cocktail does not include HTML5 storage and file system states as part of its voting algorithm. HTML5 storage is a browser mechanism that enables JavaScript code to store persistent state in the browser. Omitting checks on HTML5 storage does *not* affect Cocktail adversely because these states are contained within the browser and ultimately will affect the network or display states to have an effect on the browser, thus Cocktail will still prevent damage that can be done from attackers accessing HTML5 storage. However, file system state changes from the browser can affect the rest of the state on the system. Fortunately our sandboxing system prevents replicas from accessing many sensitive states on the system, like binaries and libraries. However, there are still some files that the replicas can access, leaving the opportunity for attacks to cause damage. This limitation is not fundamental and a result of our current implementation – we plan to add checks for file system modifications in future work.

11 Additional related work

A number of recent systems improve the security of web browsers. BrowserShield [51] rewrites suspicious script codes to safe equivalents with web-proxy-based injection and rewriting. Tahoma [38] and OP [36] enforce different levels of separation in browser construction. NativeClient [62] and Xax [33] provide built-in sandboxing techniques to contain potentially unsafe code execution. Gazelle [59] and MashupOS [37] protect information from one party from other parties' access. Nozzle [50] detects heap spray attacks with language-based techniques. StriderMonkey [60] builds honeyfarms for browser exploit detection and SpyProxy [45] renders web contents before they reach the user in a proxy-based architecture to detect malicious contents.

Cocktail is also related to some techniques used in browser testing. Selenium automates web application testing among different browsers by injecting the same inputs into different browsers. There are also "Hybrid" browsers with multiple layout engine to allow users to choose a layout engine for each site, giving them more flexibility such as Lunascape and the Sogou browser. However, none of these systems focus on improving security by comparing behaviors of different browsers.

Previous projects have studied using computer vision techniques to match web pages. For example, Alhambra [57] uses the SIFT algorithm [41] to detect differences resulting from different browser security policies. However, the SIFT algorithm takes seconds to match and finds similarity features among images caused by scaling and rotation, making it unsuitable for use in real time. In the Web search area, two projects segment the rendered web page into different layout areas with the help of the DOM information [26, 29]. These projects trust the web page to be non-malicious, otherwise can be easily confused by maliciously misplaced elements on the page. There are also projects that build connections between browser visual displays and security to check for inconsistencies between the DOM and what the browser is displaying [34, 59, 55]. Some recent projects in the anti-phishing area [43] compare the difference between phishing sites and corresponding authentic sites using computer vision techniques such as 2D Haar wavelet transformation [56].

12 Conclusions

In this paper we presented Cocktail, a browser designed to improve security and reliability for modern web browsers. To achieve this improvement, we used three off-the-shelf browsers in parallel to provide an opportunistic N-Version programming system. Cocktail mirrors all inputs across the different browser replicas and votes on all outputs to withstand attacks, even if one of the replicas becomes compromised or crashes. To enable voting, Cocktail abstracts key security relevant states from each of our replicas and compares these states across all replicas to withstand potential attacks. By abstracting states, Cocktail focuses on security features of the system despite the implementation-specific idiosyncrasies of each of our replicas. Our results showed that Cocktail withstood four exploits on real browser vulnerabilities, kept running after real browser crash issues and fault inject experiments, and added little overhead to the page load latency times for the web sites we tested.

Acknowledgment

We would like to thank Anthony Cozzie and Matt Hicks their feedback on a draft of our paper. We would also like to thank our shepherd Lujo Bauer and the annonymous reviewers who provided valuable feedback on this paper. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, and by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

References

[1] Acid3 web standards test. http://acid3.acidtests.org.

[2] Ansi (american national standards institute) http://www.ansi.org/.

[3] Ecma (european association for standardizing information and communication systems) http://www.ecma.ch/.

[4] Firefox 2.0.0.1 location.hostname vulnerability https://bugzilla.mozilla.org.

[5] Firefox 3.0.4 dos attack http://blog.zoller.lu/2009/04/advisory-firefox-denial-of-service.html.

[6] Firefox 3.0.6 cve-2009-0071 http://bugzilla.mozilla.org.

[7] Firefox heap spray exploit http://www.milw0rm.com/exploits/9137.

[8] Firefox heap spray exploit proof of concept code for os x http://www.milw0rm.com/exploits/9247.

[9] Firefox vulnerability cve-2009-2477 http://blog.mozilla.com/security/2009/07/14.

[10] Firefox vulnerability cve-2009-3076 http://www.mozilla.org/security/announce /2009/mfsa2009-48.html.

[11] The information layer www.dagstuhl.de/materials/files /09/09141/09141.borderskevin.slides.ppt.

[12] The internet engineering task force http://www.ietf.org/.

[13] Iso (international organization for standards) http://www.iso.ch/.

[14] Mitm proxy from stanford http://crypto.stanford.edu/ssl-mitm.

[15] Opera 10.10 xml parser caused crash http://www.exploit-db.com/exploits/11247.

[16] Rightnow http://www.rightnow.com/cx-suite-co-browse.php.

[17] Selenium http://seleniumhq.org/.

[18] Symantec internet security threat report http://www.symantec.com/business/threatreport.

[19] The unicode consortium http://www.unicode.org/.

[20] W3c standards http://code.google.com/p/quake2-gwt-port/.

[21] Web tap http://www.webtapsecurity.com.

[22] R. Atterer. Tracking the interaction of users with ajax applications for usability testing. In *in CHI 07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1347–1350. ACM Press, 2007.

[23] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *To appear in the Proceedings of the Usenix Annual Technical Conference (USENIX '09)*, San Diego, California, June 2009.

[24] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.

[25] E. D. Berger and B. G. Zorn. Diehard: Efficient probabilistic memory safety. *ACM Transactions on Computers*, 2007.

[26] D. Cai, S. Yu, J. rong Wen, W. ying Ma, D. Cai, S. Yu, J. rong Wen, and W. ying Ma. Vips: a vision-based page segmentation algorithm. Technical report, Microsoft Technical Report (MSR-TR-2003-79), 2003.

[27] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8:679–698, November 1986.

[28] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Operating Systems Design and Implementation*, 1999.

[29] D. Chakrabarti, R. Kumar, and K. Punera. A graph-theoretic approach to webpage segmentation. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 377–386, New York, NY, USA, 2008. ACM.

[30] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. FTCS-8:3–9, 1978.

[31] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[32] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, California, May 2006.

[33] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In R. Draves and R. van Renesse, editors, *OSDI*, pages 339–354. USENIX Association, 2008.

[34] L. Falk, A. Prakash, and K. Borders. Analyzing websites for user-visible security design flaws. In *SOUPS '08: Proceedings of the 4th symposium on Usable privacy and security*, pages 117–126, New York, NY, USA, 2008. ACM.

[35] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[36] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[37] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: operating system abstractions for client mashups. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–7, Berkeley, CA, USA, 2007. USENIX Association.

[38] R. C. Jacob, R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *In IEEE Symposium on Security and Privacy*, pages 350–364, 2006.

[39] V. K., P. Abhishek, and L. Benjamin. Ripley: automatically securing web 2.0 applications through replicated execution. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, New York, NY, USA, 2009. ACM.

[40] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *In Symposium on Operating Systems Principles (SOSP)*, 2007.

[41] D. G. Lowe. Object recognition from local scale-invariant features. In *Proc. of the International Conference on Computer Vision, Corfu*, 1999.

[42] D. Lowet and D. Goergen. Co-browsing dynamic web pages. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 941–950, New York, NY, USA, 2009. ACM.

[43] E. Medvet, E. Kirda, and C. Kruegel. Visual-similarity-based phishing detection. In *SecureComm '08: Proceedings of the 4th international conference on Security and*

privacy in communication netowrks, pages 1–6, New York, NY, USA, 2008. ACM.

[44] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI'10: Proceedings of 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[45] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. Spyproxy: execution-based detection of malicious web content. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, 2007.

[46] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.

[47] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 2003 USENIX Security Symposium*, pages 257–272, August 2003.

[48] N. Provos, P. Mavrommatis, M. Abu, R. F. Monrose, G. Inc, N. Provos, P. Mavrommatis, M. Abu, and R. F. Monrose. All your iframes point to us. *Google Inc*, 2008.

[49] N. Provos, D. Mcnamee, P. Mavrommatis, K. Wang, N. Modadugu, and G. Inc. The ghost in the browser: Analysis of web-based malware. In *In Usenix Hotbots*, 2007.

[50] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: Protecting browsers against heap spraying attacks. In *In Proc. USENIX Security*, 2009.

[51] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proceedings of The 7th Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[52] A. Richard, W. Monika, and S. Albrecht. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 203–212, New York, NY, USA, 2006. ACM.

[53] R. Rodrigues, M. Castro, and B. Liskov. Base: using abstraction to improve fault tolerance. In *IN PROC. 18TH SOSP*, pages 15–28. ACM Press, 2001.

[54] F. B. Schneider and L. Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security and Privacy*, 3:34–43, 2005.

[55] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy*, 2010.

[56] R. Stankovic and B. Falkowski. The haar wavelet transform: its status and achievements. In *Computers and Electrical Engineering, 29:25–44*, 2003.

[57] S. Tang, C. Grier, O. Aciicmez, and S. T. King. Alhambra: A system for creating, enforcing, and testing browser security policies. In *WWW '08: Proceeding of the 19th international conference on World Wide Web*, Raleigh, NC, USA, 2010. ACM.

[58] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[59] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[60] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verboswki, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *IN NDSS*, 2006.

[61] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.

[62] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society, 2009.

[63] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *IN PROC. SOSP*, pages 253–267. ACM Press, 2003.

[64] C. Yue, Z. Chu, and H. Wang. Rcb: A simple and practical framework for real-time collaborative browsing. In *USENIX Annual Technical Conference 2009*, 2009.

[65] W. Zeller and E. W. Felton. Cross-site request forgeries: Exploitation and prevention. Technical report, Department of Computer Science, Princeton University, 2008.