UNIVERSITY OF CALIFORNIA, SAN DIEGO

Language-based Security for Web Browsers

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Dongseok Jang

Committee in charge:

Professor Sorin Lerner, Chair
Professor Samuel Buss
Professor Ranjit Jhala
Professor Todd Millstein
Professor Hovav Shacham

2014

The Dissertation of Dongseok Jang is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2014

DEDICATION

To my family.

EPIGRAPH

Meanwhile, the poor Babel fish, by effectively removing all barriers to communication between different races and cultures, has caused more and bloodier wars than anything else in the history of creation.

Douglas Adams
"The Hitchhiker's Guide to the Galaxy"

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am hugely indebted to my advisor, Professor Sorin Lerner, for his guidance and support in my research and other matters. His insightful guidance has always helped me to find a way to turn seemingly trivial ideas into exciting and productive projects. Sorin has also been a patient collaborator and encouraging colleague, and I am very grateful to have had the opportunity to work with him. I am also indebted to Professor Ranjit Jhala, who I was fortunate to work with and who was always eager to offer me guidance and warm encouragement. I also had the great opportunity to work with Professor Hovav Shacham, who has generously shared his wisdom in browser security.

I would like to express my gratitude to the rest of my Ph.D. committee members, Professor Samuel Buss and Professor Todd Millstein for spending their precious time on reviewing my dissertation. I am grateful to have a great collaborator at UC San Diego, Zachary Tatlock, who shared his insights, encouragement, and cheerfulness that made collaborating with him both fun and productive.

I would also like to thank the internship hosts at Google, Charlie Reis and Albert Wong, for helping me learn from the internship by sharing their expertise sharpened in industry for years. I am also grateful to Dave Herman from Mozilla, who introduced me to many other Mozilla engineers.

Thanks to the members of the UCSD Programming System Group and many friends of mine at UCSD for sharing their time and life with me during my graduate study. Thanks to the great atmosphere of California and the Californian sun for allowing me to feel relaxed even when I was working on a deadline.

Finally, I would like to thank Hyesuk for sharing all the joy and hardships that I have been through to finish this dissertation.

Chapter 2, in full, is a reprint of the material as it appears in Ehab Al-Shaer, Angelos D. Keromytis, Vitaly Shmatikov, editors, *Proceedings of CCS 2010.* Dongseok

Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham, ACM Press, October 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in Lujo Bauer, editor, *Proceedings of NDSS 2014.* Dongseok Jang, Zachary Tatlock, and Sorin Lerner, Internet Society, February 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium.* Dongseok Jang, Zachary Tatlock, and Sorin Lerner, USENIX Association, August 2012. The dissertation author was the primary investigator and author of this paper.

VITA

2007	B.S. in Computer Science, Sogang University, South Korea
2009	M.S. in Computer Science, KAIST, South Korea
2014	Ph.D. in Computer Science, University of California, San Diego, USA

PUBLICATIONS

Dongseok Jang, Kwang-Moo Choe, "Points-to Analysis for JavaScript", *Symposium on Applied Computing*, 2009.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, Hovav Shacham, "An Empirical Study of Privacy-Violating Information Flows in JavaScript", *Computer and Communications Security*, 2010.

Dongseok Jang, Aishwarya Venkataraman, G. Michael Sawka, Hovav Shacham, "Analyzing the Cross-domain Policies of Flash Application", *Web 2.0 Security & Privacy*, 2011.

Dongseok Jang, Zachary Tatlock, Sorin Lerner, "Establishing Browser Security Guarantees through Formal Shim Verification", *USENIX Security*, 2012.

Dongseok Jang, Zachary Tatlock, Sorin Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks", *Network and Distributed System Security*, 2014.

Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, Sorin Lerner, "Automating Formal Proofs for Reactive Systems", *Programming Language Design and Implementation*, 2014.

ABSTRACT OF THE DISSERTATION

Language-based Security for Web Browsers

by

Dongseok Jang

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Sorin Lerner, Chair

Web browsers are one of the most security-critical applications that billions of people use to access their private information ranging from bank statements to medical records. However, we have witnessed numerous browser security vulnerabilities that allow attackers to steal these information or hijack a user's machine in the last decade.

Many of these security vulnerabilities are rooted in the lack of security support from programming languages used in browsers. First, JavaScript, the browser-side scripting language, lacks flexible language constructs to isolate code originating from different websites despite the common practice of merging JavaScript programs from

untrusted sources into one web page. As a result JavaScript attacks have affected numerous sites. Second, C++, the language in which major browsers are implemented, does not guarantee memory safety. As a result, memory corruption attacks are prevalent in browsers; in the worst scenario, memory corruption attacks can hijack a user's machine. Third, due to the lack of reasoning support, C++ makes it challenging to implement correct and thorough security policies in browsers comprising millions of lines of code. Loopholes in implementations have thus been exploited to circumvent intended security measures. These factors suggest that we can retrofit these languages with security-relevant constructs or incorporate a security-oriented language to address these problems.

This dissertation argues that we can adapt language techniques to improve browser security. To support this argument, we present the following contributions. First, we present a dynamic information flow framework for JavaScript to detect and prevent data stealing attacks in JavaScript web applications. Second, we present SAFEDISPATCH, an enhanced C++ compiler that prevents C++ control flow hijacking attacks, a class of attacks that exploit vtable pointers in the browser. Third, we present QUARK, a browser with a kernel formally verified to satisfy crucial security properties even when another browser component is compromised.

We highlight experimental results showing that each of our contributions is a practical defense mechanism against various browser security problems. We have implemented our proposals in real browsers such as Chromium and Webkit and showed that they successfully run on real websites, including Google, Amazon, and Facebook.

Chapter 1

Introduction

Web browsers serve an important role to securely mediate billions of people's private information ranging from banking transactions to medical records on diverse websites, some of which turn malicious. Browsers are responsible to securely isolate and share these information from different sites. In the modern web, a website consists of not only static HTML documents, but also dynamic contents manipulated by client-side web applications, typically written in JavaScript, that interact with multiple websites having different security characteristics. The browser is responsible for defining the boundaries of web applications originating from different sources, and applies proper security checking among them so that the information on a site is neither corrupted from nor leaked into other malicious sites accidentally. The browser also has another fundamental security goal that any applications dealing with untrusted network data share : the browser should prevent websites from directly accessing system resources (e.g., file system).

However, we have witnessed numerous security problems ranging from data stealing JavaScript to arbitrary code execution in all major browsers. JavaScript code injection via cross-site scripting (XSS) accounted for 43% of *all* Internet security vulnerabilities documented during 2012 by WhiteHat Security [46]. Online advertisements also have been misused to inject malicious JavaScript code into popular sites such as The

Wall Street Journal [84]. Indeed, almost all popular websites including Google Mail and Facebook have often been affected by a certain form of malicious JavaScript injection attacks [58, 36]. Browser have also suffered from all kinds of software bugs ranging from incorrect security policy implementation to memory corruption bugs. On the one hand, browsers have been plagued with memory corruption bugs that may lead to arbitrary code execution [40, 109, 98, 71, 72]. On the other hand, browser security policies have so rapidly evolved that some newly introduced security features were circumvented [47] due to their inconsistency with old security features. Despite the efforts of researchers to fix browser security bugs, critical exploits have been regularly found. To deal with this problem, Google even has started a Vulnerability Reward Program as an attempt to harden their Chrome browser [2]. We identify several reasons for these problems as the lack of security support from the programming languages either run by the browser or used to implement the browser.

First, JavaScript, the programming language for client-side web applications, whose code the browser fetches from websites to execute, lacks language-based isolation mechanisms such as information hiding. However, JavaScript has extremely dynamic language features that sometimes allow for unexpected interference among JavaScript programs embedded in a same webpage, but originating from different sources. Instead of language-level security mechanisms, browsers have evolved with ad-hoc browser-level isolation mechanisms such as the same origin policy (SOP) for access control of the domain object model (DOM) exposed to JavaScript. These browser-level isolation mechanisms are sometimes too coarse-grained to accommodate diverse security requirements of rapidly evolving Web 2.0 websites. For example, under the SOP, all JavaScript code embedded in a same web page, no matter where it originates or how it is embedded, is

given the same privilege [1]. Under the rigidity of browser-level isolation mechanisms such as the SOP, websites are forced to put JavaScript code even from other untrusted input into the same origin of the hosting site. Consequently, over-privileged malicious JavaScript code accidentally inserted into important sites have caused a variety of security vulnerabilities such as cross-site scripting (XSS).

Second, the lack of memory safety in C++, the language in which major browsers are implemented for its performance and popularity, has resulted in numerous memory corruption bugs in the browser. C++ has notoriously many sources of memory corruption bugs. For example, C++ does not support garbage collection, and it performs no array bound checking; C++ also allows for unrestricted typecasting that lead to uncontrolled memory access via type confusion. Therefore, memory safety entirely depends on the correctness of a C++ program. However, browsers have become so complicated with millions of lines of C++ code that there have often been found serious memory bugs. Attackers exploit memory bugs in a C++ program to illegally overwrite *control data* (e.g., return addresses, indirect jump targets) to lead the program to execute unexpected instructions performing dangerous operations. In the worst case, memory corruption attacks lead to full takeover of a user machine. In the case of web browsers, attackers can exploit a victim browser by serving a maliciously crafted web content that triggers memory corruption attacks; just by visiting a maliciously crafted webpage a victim's browser can turn into a backdoor of the attacker. The security community has responded to such attacks with practical defenses [16, 81, 29, 7], and some of them have been already deployed in some browsers. However, high profile attacks have constantly shifted their focus to corrupting another class of control data such as virtual table pointers [40, 109, 98, 71, 72].

[1] A recent browser security feature, Content Security Policy(CSP) refines this coarseness by distinguishing JavaScript code by their origin and embedding method

Third, C++ lacks language features such as a strong type system that helps to assure the correctness of security-relevant code in complex browser implementations consisting of millions of lines of C++ code. As the browser evolves with more functionality, components securing them have also become complicated. For example, the same origin policy, a seemingly simple policy originally introduced only for JavaScript, is later extended to secure cross-site communication via XMLHTTPRequest (XHR), persistent browser local storage, and even each pixel from HTML5 canvases. However, the same origin policy was slightly modified for each of the cases to accommodate their security requirements. As a result, it has become challenging to assure that these security components are both temper-proof and complete due to their complexity, and there are often found security vulnerabilities of incomplete security checking. For example, the Firefox browser had bugs of missing the SOP checking in certain situations, which enables attackers to bypass the SOP [5, 6].

In essence, many of browser security vulnerabilities are rooted in the lack of security support from the programming languages used in the browser. The languages for the browser, JavaScript and C++, are designed for their flexibility and performance instead of security. However, it appear to be an unacceptable solution to re-implement millions of existing JavaScript web applications or browser implementations in new security-oriented languages. It is another social matter to convince JavaScript programmers of switching to a new language for security. It also appear impractical to rewrite a significant security-relevant portion of the browser from scratch in a new programming environment considering its huge cost. These factors suggest that we have to find a solution that harmoniously works with existing codebase without disrupting the development practice of web programmers or browser developers.

1.1 Outline of this work

This dissertation argues that we can adapt language-based techniques to address the class of problems discussed above without sacrificing compatibility. To support this argument, we propose three language-based methods, each of which alleviates a class of the problems mentioned above in practical settings. For isolating JavaScript in one web page, we adapt *dynamic information flow* [97] to specify and enforce confidentiality policies stating what parts of the web page can be read by what JavaScript code and integrity policies stating what JavaScript code can affect what parts of the page. To address a class of C++ control flow hijacking attacks frequently found in browsers [40, 109, 98, 71, 72], we developed a specialized form of control flow integrity [7] to prevent *vtable hijacking*, a class of memory corruption attacks that exploit C++ dynamic dispatch to hijack the program control flow. To raise the software assurance level of the browser, we adapt *software formal verification* to implement and verify the actual implementation of a security-critical browser component; exploiting the kernel-based browser architecture [15], our prototype browser with a verified kernel could provide strong guarantees without reasoning about other components.

We outline our three language-based techniques for better browser security and describe our research contributions associated with each of them.

- *Dynamic Information Flow for JavaScript* : we present a framework that tracks information flow for JavaScript *dynamically*. Our framework residing in the browser inserts and propagates taints through a JavaScript program as it runs to enforce confidentiality and integrity policies. The dynamic nature of our analysis allows it to precisely track flow even through the many features of JavaScript that make static analysis hard. Our framework allows either web developers or browser users to flexibly specify their diverse security requirements in terms of information

flow, alleviating the problems caused by the rigidity of the same origin policy. To show how well our framework can capture security violation of JavaScript in the wild, we have used a Chromium browser enhanced with our flow framework to conduct a large-scale empirical study over the Alexa global top 50,000 websites of four privacy-violating flows: cookie stealing, location hijacking, history sniffing and behavior tracking.

- *Control Flow Integrity for C++ Virtual Calls* : we address the growing threat of C++ vtable hijacking with SAFEDISPATCH, enhanced C++ compiler that prevents such attacks. SAFEDISPATCH instrument C++ programs with runtime checks that precisely reflects static typing rules for dynamic dispatch. By carefully optimizing these checks, we were able to reduce runtime overhead to just 2.1% in the first vtable-safe version of the Google Chromium browser which we built with the SAFEDISPATCH compiler.

- *Formal Verification for Browser Kernel* : we present a browser with a formally verified kernel, Quark. Quark is structured similarly to the Chrome browser [15], around a small kernel that mediates access to system resources for other browser components. This architecture allows us to make strong guarantees about a millions lines of code for other components just while formally verifying about only a few hundreds line of code for the kernel within the demanding and foundational context of the mechanical proof assistant Coq. We also show that Quark is practical even with this strong guarantee : it opens popular demanding websites such as Google Maps or GMail without disruption.

We demonstrate that the proposed methodology are practical by applying our techniques to real browser such as Chrome and Webkit, and running the browsers enhanced by our techniques on actual websites such as Google Mail and Facebook.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 shows how to capture realistic JavaScript attacks using dynamic information flow. Chapter 3 presents a C++ code instrumentation technique to secure virtual function calls at runtime. Chapter 4 shows how to apply software formal verification to verify a browser kernel. Chapter 5 surveys related work on browser security and language-based applications. Chapter 6 presents further opportunities to improve our work and summarizes the dissertation.

Chapter 2

Securing JavaScript via Dynamic Information Flow Tracking

Web applications often either intentionally or accidentally load JavaScript code from untrusted sources, including ad sites and user input, which in some cases can steal or corrupt important information in web applications. Furthermore, JavaScript can cleverly manipulate the browser to exfiltrate private information residing in the browser without the user's consent. In this chapter, we present a dynamic information flow framework for JavaScript that enforces confidentiality and integrity policies that respectively specify what information can flow into and from certain JavaScript code. We have implemented our approach in the Chromium browser that we can use to prevent malicious JavaScript code from stealing the sensitive data of the hosting website. Furthermore, to show the effectiveness of our approach as a survey tool, we used our framework to conduct a large-scale study of privacy-violating information flows in JavaScript code over Alexa top 50,000 websites. We have found a significant number of JavaScript web applications exhibiting privacy-violating behavior suggesting that we need to take a step to devise an effective defense against them.

2.1 Motivation

JavaScript is a dynamically typed language that can be embedded in web pages and executed by the web browser. JavaScript is becoming the lingua franca of modern Web 2.0 applications. Almost every popular web site uses some amount of JavaScript, and many interactive web sites, like search engines, email sites and mapping applications are almost entirely implemented in client-side JavaScript.

Although JavaScript has enabled web developers to provide a richer web experience, JavaScript has also opened up the possibility for a variety of security vulnerabilities. In particular, typical JavaScript applications are made up from code originating from many different sources, including advertising websites and sometimes user-provided content. Unfortunately, JavaScript does not provide strong protection mechanisms, so that code included from a particular site, say for displaying an ad, essentially runs in the context of the hosting page. Thus, the ad code has access to all the information on the hosting web page, including the cookie, the location bar, and any other privacy-sensitive information accessible on the page. The lack of strong protection mechanisms in JavaScript has led to a variety of attacks like cross-site scripting.

To make JavaScript more secure, ideally we would like to specify and enforce confidentiality policies stating what parts of the web page can be read by what JavaScript code and integrity policies stating what JavaScript code can affect what parts of the page. One formalism that is well suited for expressing these kinds of policies is *information flow policies* which specify where in the code a given value can flow to. Thus, for example, we could state using an information flow policy that sensitive information stored in a cookie should not flow to any code loaded from third party ad servers. We could also state a flow policy that the information from untrusted code should not flow to the location bar of the hosting page.

Although there has been work on static enforcement of information flow for a variety of languages, performing static information flow on a language like JavaScript is extremely hard. JavaScript has many features that make precise static analysis all but impossible. These features include dynamic loading of code from the network, dynamic code construction and evaluation, prototypes and dynamic dispatch, dynamically added and removed fields, and dynamic field assignments (where the field name is constructed at runtime).

Rather than trying to analyze the JavaScript code statically, in this chapter we present a framework that tracks information flow for JavaScript *dynamically*. Our framework inserts and propagates taints through the program as it runs to enforce confidentiality and integrity policies. The dynamic nature of our analysis allows it to precisely track flow even through the many dynamic features of JavaScript that make static analysis hard. To show that a dynamic information flow for JavaScript is practical and flexible enough to capture a variety of JavaScript security attacks, we have carried out this study in three steps.

First, we have designed an expressive, fine-grained information flow policy language that allows us to specify different kinds of malicious information flows in JavaScript code (Section 2.2.1). In essence, our language allows us to describe different kinds of flows by specifying sites within the code where taints are *injected* and sites from which certain taints must be *blocked*. For example, to specify an information flow of cookie stealing JavaScript, we inject a “secret" taint into the cookie, and block that taint from flowing into variables controlled by third-party code. To specify flows to corrupt the location bar, we inject an “untrusted" taint onto any values originating from third-party code, and block that taint from flowing into the document’s location field.

Second, we have implemented a new JavaScript information flow engine in the Chromium browser. Unlike previous JavaScript information flow infrastructures [38,

107, 34], our engine uses a dynamic source-to-source rewriting approach where taints are injected, propagated, and blocked within the rewritten JavaScript code (Section 2.2.2). Although the rewriting is performed inside the browser, implementing our approach requires understanding only the browser's AST data structure, and none of the complexity of the JavaScript run-time. Thus, in addition to supporting an extremely flexible flow policy specification language, our approach is simple to implement and can readily be incorporated inside other browsers or web proxies [59]. Even though the taints are propagated in JavaScript, as opposed to natively, the overhead of our approach is not prohibitively high. Our approach adds on average 60 to 70% to the total page loading time over a fast school network (which is the worst condition to test our JavaScript overhead). This is efficient enough for our exploratory study focused on investigating how expressible our framework can be, and with additional simple optimizations could even be feasible for interactive use (Section 2.3).

Third, we have used the Chromium browser enhanced with our information flow framework to conduct a large-scale empirical study over the Alexa global top 50,000 websites of four kinds of privacy-violating information flows : cookie stealing, location hijacking, history sniffing, and behavior tracking. Our results reveal interesting facts about the prevalence of these flows in the wild. We did not find any instances of location hijacking on popular sites, but we did find that there are several third party ad agencies to whom cookies are leaked. We found that several popular sites — *including an Alexa global top-100 site* — make use of history sniffing to exfiltrate information about users' browsing history, and, in some cases, do so in an obfuscated manner to avoid easy detection. We also found that popular sites, such as Microsoft's, track users' clicks and mouse movements, and that `huffingtonpost.com` has the infrastructure to track such movements, even though we did *not* observe the actual flow in our experiments. Finally, we found that many sites exhibiting privacy-violating flows have built their

```
var initSettings = function(s){
  searchUrl = s;
}

initSettings("http://a.com?");

var doSearch = function() {
   var searchBox = getSearchBoxValue();
   var searchQry = searchUrl + searchBox;
   document.location = searchQry;
}

eval(load("http://adserver.com/display.js"));
```

Figure 2.1. JavaScript code on `a.com` importing untrusted code from ad servers.

own infrastructure, and do not use pre-packaged solutions, we found throughout our study, like `ClickTale`, `tynt`, `Tealium`, or `Beencounter`. Thus, our finding about them shows that popular Web 2.0 applications like mashups, aggregators, and sophisticated ad targeting are rife with different kinds of privacy-violating JavaScript code. Hence, there is a pressing need to devise flexible, precise and efficient defenses against them; and a dynamic information flow for JavaScript can be a good candidate as a defense mechanism.

2.2 Information Flow Policies

We present our approach for dynamically enforcing information flow policies through an example that illustrates the mechanisms used to generate, propagate and check taint information for enforcing flow policies. The focus of our information flow policies and enforcement mechanism is to detect many kinds of dangerous information flows, not to provide a bullet-proof protection mechanism (although our current system could eventually lead to a protection mechanism, as discussed further in Section 6). We defer a formal treatment of our rewriting algorithm to a technical report [53].

Web page Consider the JavaScript in Figure 2.1. Suppose that this code is a distillation of the JavaScript on a web page belonging to the domain `a.com`. The web page has a text box whose contents can be retrieved using a call to the function `getSearchBoxValue` (not shown). The function `initSettings` is intended to be called once to initialize settings used by the page. The `doSearch` function is called when the user clicks a particular button on the page.

Dynamically Loaded Code The very last line of the code in Figure 2.1 is a call to `load()` which is used to dynamically obtain a code string from `adserver.com`. `load()` can load only JavaScript code from other sites under the same origin policy [1]. This code string is then passed to `eval()` which has the effect of "executing" the string as a piece of program code in order to update the web page with an advertisement tailored to the particular user.

Malicious Code Suppose that the string returned by the call to `adserver.com` was:

```
initSettings("http://evil.com?");
```

When this string is passed to `eval()` and executed, it overwrites the page's settings. In particular, it sets the variable `searchUrl` which is used as the target web page to which the browser navigates with the search keyword, to refer to an attacker site `evil.com`. Now, if the user clicks the search button, the `document.location` gets set to the attacker's site, and thus the user is redirected to a malicious website which can then compromise her machine. Similarly, dynamically loaded code can cause the user to leak their password, cookie, or other sensitive information available on `a.com`

2.2.1 Policy Language

The flexibility and dynamic nature of JavaScript makes it difficult to use existing language-based isolation mechanisms. First, JavaScript does not have any information

[1] The actual browser does not support `load()`, but we introduce it here to model a variety of dynamic code evaluation methods in a unified way

hiding mechanisms like private fields that could be used to isolate `document.location` from dynamically loaded code. Indeed, a primary reason for the popularity of the language is that the absence of such mechanisms makes it easy to rapidly glue together different libraries distributed across the web. Second, the asynchronous nature of web applications makes it difficult to enforce isolation via dynamic stack-based access control. Indeed, in the example above, the malicious code has done its mischief and departed well before the user clicks the button and causes the page to redirect.

Thus, to reconcile safety and flexible, dynamic code composition, we need fine-grained isolation mechanisms that prevent untrusted code from viewing or affecting sensitive data. Our approach to isolation is information flow control [39, 75], where the isolation is ensured via two steps. First, the website's developer provides a fine-grained *policy* that describes which values *can affect* and *be affected* by others. Second, the language's compiler or run-time *enforces* the policy, thereby providing fine-grained isolation.

Policies In our framework a fine-grained information flow policy is specified by defining taints, injection sites and checking sites. A *taint* is any JavaScript object, *e.g.,* a URL string denoting the provenance of a given piece of information. A *site*

$$r.f(x\ldots)$$

corresponds to the invocation of the method f with the arguments $x\ldots$, on the receiver object r. Such site expressions can contain concrete JavaScript (*e.g.,* `document.location`), or pattern variables (*e.g.,* \$1) that can match against different concrete JavaScript values, and which can later be referenced.

In order to allow sites to match field reads and writes, we model these using getter and setter methods. In particular, we model a field read using a call to method `getf`,

which takes the name of the field as an argument, and we model a field write using a call to a method `setf`, which takes the name of a field and the new value as arguments. To make writing policies easier, we allow some simple syntactic sugar in expressing sites: $r.x$ used as an r-value translates to $r.\texttt{getf}(x)$ and $r.x = e$ translates to $r.\texttt{setf}(x, e)$.

An *injection* site

$$\textsf{at } S \textsf{ if } P \textsf{ inject } T$$

stipulates that the taint T be *added to* the taints of the object output by the method call described by the site S as long as the condition P holds at the callsite. For example, the following injection site unconditionally injects a "secret" taint at any point where `document.cookie` is read:

$$\textsf{at } \texttt{document.cookie} \textsf{ if } \texttt{true} \textsf{ inject } \text{“}\texttt{secret}\text{”}$$

To make the policies more readable, we use the following syntactic sugar: when "if P" is omitted, we assume "if `true`". As a result, the above injection site can be expressed as:

$$\textsf{at } \texttt{document.cookie} \textsf{ inject } \text{“}\texttt{secret}\text{”} \tag{2.1}$$

A *checking* site

$$\textsf{at } S \textsf{ if } P \textsf{ block } T \textsf{ on } V$$

stipulates that at site S, if condition P holds, the expression V must *not contain* the taint T. We allow the guard (P), the taint (T) and the checked expression (V) to refer to the any pattern variables that get bound within the site S. As before, when "if P" is omitted, we assume "if `true`".

For example, consider the following checking site:

$$\text{at } \$1.x{=}\$2 \text{ if } \$1.\texttt{url} \neq \text{“}\texttt{a.com}\text{”} \text{ block “}\texttt{secret}\text{” on } \$2 \tag{2.2}$$

The `url` field referenced above is a special field added by our framework to every object, indicating the URL of the script that created the object. The above checking site therefore ensures that no value tainted with “secret” is ever assigned into an object created by a script that does not originate from `a.com`.

Confidentiality policies can be specified by injecting a special “secret" taint to the getter-methods for confidential variables, and checking that such taints do not flow into the inputs of the setter-methods for objects controlled by code originating at domains other than `a.com`. Such a policy could be formally specified using (2.1) and (2.2) above.

Integrity policies can be specified by injecting special “untrusted" taints to getter-methods for untrusted variables, and checking that such taints do not flow into the inputs of setter-methods for trusted variables. Such a policy could be formally specifed as:

$$\text{at } \$1.x \text{ if } \$1.\texttt{url} \neq \text{“}\texttt{a.com}\text{” inject “}\texttt{untrusted}\text{”}$$

$$\text{at } \texttt{document.location} = \$1 \text{ block “}\texttt{untrusted}\text{” on } \$1$$

We can specify even finer grained policies by refining the taints with information about individual URLs. The expressiveness of our policy language allows us to quickly experiment with different kinds of flows within the same basic framework, and could also lay the foundation for a browser-based protection mechanism.

2.2.2 Policy Enforcement

The dynamic nature of JavaScript, particularly dynamic code evaluation, makes precise static policy enforcement problematic, as it is impossible to predict what code will be loaded at run-time. Thus, our approach is to carry out the enforcement in a fully dynamic manner, by rewriting the code in order to inject, propagate and checks taints appropriately.

Although there are known dangers to using rewriting-based approaches for protection [68], our current goal is actually not protection, bur rather to find as many privacy-violating flows as possible. As such, one of our primary concerns is ease of prototyping and flexibility; in this setting, rewrite-based approaches are very useful. In particular, implementing our approach only required understanding the browser's AST data structure, and none of the complexities of the JavaScript runtime, which allowed us to quickly build and modify our prototype as needed. Furthermore, keeping our framework clearly separate from the rest of the browser gives us the flexibility of quickly porting our approach to other browsers.

Policy Enforcement Our framework automatically rewrites the code using the specified injection and checking sites to ensure that taints are properly inserted, propagated and checked in order to enforce the flow policy in pure JavaScript. First, we use the checking (resp. injection) sites to synthesize wrappers around the corresponding methods that ensure that the inputs do not contain (resp. outputs are tainted with) the taints specified by the corresponding taint expressions whenever the corresponding guard condition is met. Second, we rewrite the code so that it (dynamically) propagates the taints with the objects as they flow through the program via assignments, procedure calls *etc.*. We take special care to ensure correct propagation in the presence of tricky JavaScript features like `eval`, prototypes, and asynchronous

calls.

Rewriting Strategy Our strategy for enforcing flow policies, is to extend the browser with a function that takes a code string and the unique identifier for the security domain of the code (in our example, the URL from which the code string was loaded), and returns a rewritten string which contains operations that perform the injection, propagation, and checking of taints. Thus, to enforce the policy, we ensure that the code on the web page is appropriately rewritten before it is evaluated. We ensure that "nested" `eval`-sites are properly handled as follows. We implement our rewriting function as a procedure in the browser source language(C++) that can be called from within JavaScript using the name `RW` and the rewriter wraps the arguments of `eval` within a call to `RW` to ensure they are (recursively) rewritten before evaluation [114].

When the rewriting procedure is invoked on the code from Figure 2.1 and the URL `a.com`, it emits the code shown in Figure 2.2. The rewriting procedure rewrites each statement and expression. (In Figure 2.2, we write the original code as a comment above the rewritten version.) Next, we step through the rewritten code to illustrate how taints are injected, checked, and propagated, for the integrity property that specifies that `document.location` should only be influenced by `a.com`.

Injection To inject taints, we extend *every* object with two special fields `url` and `taint`. To achieve this, we wrap all object creations inside a call to a special function `box` which takes a value and a *url* and creates a boxed version of the value where the `url` field is set to *url* indicating that the object originated at *url*, and the `taint` field is set to the empty set of taints. We do this uniformly to all objects, including functions (*e.g.,* the one assigned to `initSettings`), literals (*e.g.,* the one passed as a parameter to `initSettings`), *etc.*. Next, we use the specified injection sites to rewrite the code in order to (conditionally) populate the `taint` fields at method calls that match the sites. However, the integrity injection site does not match anywhere in the code so far, and so no taints are injected

```
//var initSettings = function(){...}
tmp0 = box(function(s){searchUrl = s;}, "a.com"),
var initSettings = tmp0;

//initSettings("http://a.com?");
tmp1 = box("http://a.com?", "a.com"),
initSettings(tmp1);

//var doSearch = function(){...}
var doSearch = box(function(){

  var searchBox = getSearchBoxValue();

  //var searchQry = searchBox + searchUrl;
  var searchQry = TSET.direct.add(searchUrl), tmp2 = unbox(searchUrl),
                  TSET.direct.add(searchBox), tmp3 = unbox(searchBox),
                  tmp4 = tmp2 + tmp3, TSET.boxAndTaint(tmp4, "a.com");

  //document.location = searchQry;
  check(searchQry, "untrusted"), document.location = searchQry;

  }, "a.com");

//eval(load("http://adserver.com/display.js"));
tmp5 = box("http://adserver.com/display.js", "a.com"),
tmp6 = box(load(tmp5), "a.com"),
tmp6.url = tmp5,
eval(RW(tmp6, tmp6.url));
```

Figure 2.2. Rewritten code from a.com. The comments denote the original code.

yet – they will be injected when code gets loaded from the ad server.

Checking Before each call site that matches a specified check site, we insert a call to a special `check` function. The call to `check` is predicated on the check site's condition. The function `check` is passed the checked expression V and taint T corresponding to the matching check site. The function determines whether the taints on the checked expression contain the prohibited taint, and if so, halts execution.

For example, consider the rewritten version of the assignment to `document.location` in the body of `doSearch` which matches the checking site from

the integrity policy. The rewrite inserts a call to `check`. At run-time, when this call is executed it halts the program with a flow-violation message if `searchQry.taint` has a (taint) value of the form "`untrusted`".

Propagation Next, we consider how the rewriting instruments the code to add instructions that propagate the taints.

- For assignments and function calls, as all objects are boxed, the taints are carried over directly, once we have created temporaries that hold boxed versions of values. For example, the call to `initSettings` uses `tmp0`, the boxed version of the argument, and hence passes the taints into the function's formals. The assignment to `searchBox` is unchanged from before, as the right-hand side is function call (whose result has already been appropriately boxed).

- For binary operations, we must do a little more work, as many binary operations (*e.g.,* string concatenation) require their arguments be *unboxed*. To handle such operations, we extend the code with a new object called the *taint-set*, named `TSET`. We use this object to accumulate the taints of sub-expressions of compound expressions. The object supports two operations. First, `TSET.direct.add(x,url)`, which adds the taints in `x.taint` to the taint-set. Second, `TSET.boxAndTaint(x,url)`, which creates a boxed version of `x` (if it is not boxed), and the taints accumulated on the taint-set, clears the taint-set, and returns the boxed-and-tainted version of `x`. We use the `direct` field as there are several other uses for the `TSET` object that are explained later. For example, consider the rewritten version of `searchBox + searchUrl`. We add the taints from `searchBox` (resp. `searchUrl`) to the taint-set, and assign an unboxed version to the fresh temporary `tmp2` (resp. `tmp3`). Next, we concatenate the unboxed strings, and assign the result to `tmp4`. Finally, we call `TSET.boxAndTaint(tmp4,"a.com")`, which boxes `tmp4` with the taints for the

sub-expressions stored in the taint-set and set its `url` to "`a.com` " as its creator, and returns the boxed-and-tainted result.

- For code loading operations (modeled as `load`$(\cdot)$), the rewriting boxes the strings, and then adds a `url` field to the result that indicates the domain from which the string was loaded. For example, consider the code loaded from `adserver.com`. The name of the URL is stored in the temporary `tmp5`, and the boxed result is stored in a fresh temporary `tmp6`, to which we add a `url` field that holds the value of `tmp5`.

- For `eval` operations, our rewriting interposes code that passes the string argument to `eval` and the URL from which the string originated to the the rewriting function `RW`, thereby ensuring the code is rewritten before it is evaluated. For example, consider the operation at the last line of the code from Figure 2.1 which `eval`'s the string loaded from `adserver.com`. In the rewritten version, we have a boxed version of the string stored in `tmp6`; the rewriting ensures that the string that gets executed is actually `tmp6` rewritten assuming it originated at `tmp6.url`, which will have the effect of ensuring that taints are properly injected, propagated and checked within the dynamically loaded code.

The above code assumes, for ease of exposition, that the fields `taint` and `url` are not read, written or removed by any code other than was placed for tracking. However, our actual implementation makes sure that the user code cannot touch those fields by interposing field access.

Attack Prevention Suppose that the `load`() operation returns the following code string.

```
initSettings("evil.com");
```

The rewritten code invokes the rewriting function `RW` on the string, and the URL `adserver.com` yielding the string

```
tmp10 = box("http://evil.com?", "adserver.com"),
if (tmp10.url != "a.com"){
  tmp10.taint += ["untrusted"]
},
initSettings(tmp10);
```

The `if`-expression injects the taints to the value returned by the implicit getter-call (*i.e.,* the read of `tmp10`) that yields the value passed to `initSettings`. Thus, the argument passed to `initSettings` carries the taint "`untrusted`", which flows into `searchUrl` when the assignment inside `initSettings` is executed. Finally, when the button click triggers a call to `doSearch`, the taint flows through the taint-set into the value returned by the call `TSET.boxAndTaint(tmp4,"a.com")`, and from there into `searchQry`. Finally, the `check` (just before the assignment to `document.location`) halts execution as the flow violates the integrity policy, thereby preventing the location hijacking attack.

Rewriting for Confidentiality Policies The above example illustrates how rewriting enforces integrity policies. The case for confidentiality policies differs only in how taints are injected and checked; the taints are propagated in an identical fashion. To *inject* taints, the rewriting adds a "`secret`" taint to the results of each *read* from a confidential object (*e.g.,* `document.cookie`). To *check* taints, the rewriting inserts calls to `check` before any writes to variables (*i.e.,* invocations of setter methods) in code originating in untrusted URLs. The `check` halts execution before any value with the "`secret`" taint can flow into an untrusted location.

Robustness Even though the primary purpose of our tool so far has been to evaluate existing flows (a scenario under which we don't need to worry about malicious code trying

to subvert our system), our framework does in fact protect its internal data structures from being maliciously modified. In particular, our tool disallows a user JavaScript program from referencing any of the variables and fields used for taint tracking such as `TSET`. More specifically, since our framework tracks reads and writes to *all* variable and field, it can simply stop a program that tries to read or write to any of the internal variables that we use to track information flow.

2.2.3 Indirect Flows

Next, we look at how the rewriting handles indirect flows due to control dependencies. We start with the data structure that dynamically tracks indirect flows, and then describe the key expressions that are affected by indirect flows.

Indirect Taint Stack (`TSET.indirect`) To track indirect flows, we augment the taint set object with an *indirect-taint* stack (named `TSET.indirect`). Our rewriting ensures that indirect taints are added and removed from the indirect taint stack as the code enters and leaves blocks with new control dependencies at runtime. The `TSET.boxAndTaint`$(\cdot,\cdot)$ function, which is used to gather the taints for the RHS of assignments, embellishes the (RHS) object with the direct taints at the *top of* the direct taint stack, *and* the indirect taints stored *throughout* the indirect taint stack. The latter ensures that at each assignment also propagates the indirect taints that held at the point of the assignment.

Branches For branch expressions of the form `if` e_1 e_2 e_2, we first assign the rewritten guard to a new temporary $\texttt{tmp}_1$, and push the taints on the guard onto the indirect taint stack. These taints will reside on the indirect taint stack when (either) branch is evaluated, thereby tainting the assignments that happen inside the branches. After the entire branch expression has been evaluated, the rewritten code pops the taints, thereby reverting the stack to the set of indirect taints before the branch was evaluated.

Example Consider the branch expression: `if (z) { x = 0 }` To ensure that taints

from z flow into x when the assignment occurs inside the then-branch, the expression is rewritten to:

```
tmp = z,
TSET.indirect.push(tmp),
if (unbox(tmp)){
  x = TSET.boxAndTaint(box(0,...),...)
},
TSET.indirect.pop()
```

The ellipses denote the URL string passed to RW and we omit the calls to `check` and `TSET.direct.add`$(\cdot,\cdot)$ for brevity. The rewrite ensures that the taints from the guard z are on the indirect taint stack inside the branch, and these taints are added to the (boxed version of) 0 that is used for the assignment, thereby flowing them into x. The pop after the branch finishes reverts the indirect stack to the state prior to the branch.

Indirect vs. Implicit Flows. The above example illustrates a limitation of our fully dynamic approach; we can track *indirect* flows induced by a taken branch (such as the one above) but not *implicit* flows that occur due to a not-taken branch. For example, if the above branch was preceded by an assignment that initialized x with 1, then an observer that saw that x had the value 1 after the branch would be able to glean a bit of information about the value of z. Our rewriting, and indeed, any fully dynamic analysis [32] will fail to detect and prohibit such implicit flows.

Function Calls Our rewriting adds an indirect taint parameter to each function definition. This parameter holds the indirect taints that hold at the start of the function body; the taints on it are pushed onto the indirect taint stack when the function begins execution. Furthermore, the rewriting ensures that at each function callsite, the indirect taints (in the indirect taint stack) at the caller are passed into the indirect taint parameter of the callee.

Event Handlers cause subtle information flows. For example, if `foo` is the handler for the `MouseOver` event, the fact that `foo` is executed contains the information that the mouse hovered over some part of the page. We capture these flows as indirect flows triggered by the tests within the DOM event dispatch loop

```
while(1){
  e = getEvent();
  if (e.isClick()) onClick(e);
  if (e.isMouseOver()) onMouseOver(e);
  if (e.isScroll()) onScroll(e);
  ...
}
```

Thus, to capture flows triggered by a `MouseOver` event, we simply inject a taint at the output of the `$1.isMouseOver(...)`. The `if` ensures that the taint flows into the indirect taint parameter of the registered handler (bound to `onMouseOver`).

Full JavaScript The examples described in this section have given a high-level overview of our rewriting-based approach to enforcing information flow policies. Our implementation handles all of JavaScript including challenging language features like prototypes, with-scoping, and higher-order functions. Our implementation also incorporates several subtle details pertaining to how taints can be stored and propagated for *unboxed* objects, to which a `taint` field cannot be added. The naive strategy of boxing all objects breaks several websites as several DOM API functions require unboxed objects as arguments. We refer the reader to an accompanying technical report [53] for the details.

2.3 Implementation and Performance Evaluation

This section presents our implementation of rewrite-based information flow framework for JavaScript in the Chromium browser, and describes several experiments that quantify the performance overhead of our approach.

Implementation We implement the rewriting module in C++ within the V8 JavaScript engine of Chromium that is invoked on any JavaScript code just before it gets sent into the JavaScript engine. Thus, our implementation rewrites *every* piece of JavaScript including that which is loaded by `<script>` tags, executed by `eval` or executed by via `document.write`. Our rewriting module is given the information on how a piece of code is inserted, and it can decide the security domain (`url` in our discussion so far) of the code. The `TSET` library is implemented in pure JavaScript, and we modified the resource loader of Chromium to insert the `TSET` library code into every JavaScript program it accesses. The `TSET` library is inserted into each web page as ordinary JavaScript using a `<script>` tag before any other code is loaded. The flow-enhanced Chromium can run in normal mode or in taint tracking mode. When the taint tracking is on, the modified Chromium tracks the taint flow as a user surfs on websites.

Optimizations We describe the three most important optimizations we performed for the "optimized" bar. The first and most important optimization is that we implemented the two most frequently executed TSET methods using 65 lines of C++, namely the methods for taint lookup and unboxing. Second, in the `TSET.direct` stack, when there is a pop followed by a push, and just before the pop there are *no* taints stored at the top of the stack, we cache the object at the top of the stack before poping, and then reuse that same object at the next push, thus avoiding having to create a new object. Because the push is called on every assignment, removing the object creation provides a significant savings. Third, we also cache field reads in our optimized `TSET` library. For example, whenever a

property `a.b` is referenced several times in the library, we store the value of the property in a temporary variable and reuse the value again. This produces significant savings, despite the fact that all our measurements used the JIT compiler of the V8 engine.

Benchmarks We employ a set of stress experiments using the standard cookie confidentiality and location integrity policies to evaluate the efficiency of our approach and the effect of optimizations. These policies require a significant amount of taint propagation, as the cookie is heavily accessed and our benchmark sites contain lots of third-party code. As our benchmarks, we use the front pages on the websites from the latest Alexa global top 100 list. Alexa is a company which ranks websites based on traffic. The websites on the Alexa global top 100 vary widely in size and how heavily they use JavaScript, from 0.1 KLOC to 31.6 KLOC of JavaScript code. We successfully ran our dynamic analysis on all of the pages of the Alexa global top 100 list, and we visited many of them manually to make sure that they function properly.

2.3.1 Policies

To measure efficiency, we checked two important policies on each site. First, `document.cookie` should remain confidential to the site. Second, `document.location` should not be influenced by another site. Both policies depend on a definition of what "another site" is. Unfortunately, using exactly the same URL or domain name often leads to many false alarms as what looks like a single website is in fact the agglomeration of several different domain names. For example, `facebook.com` refers to `fbcdn.net` for many of their major resources, including JavaScript code. Moreover, there are relatively well known and safe websites for traffic statistics and advertisements, from which JavaScript libraries are imported on many other websites, and one may want to consider those as safe. Thus, we considered three URL policies (*i.e.,* three definitions for "another site") (1) the *same-origin policy* stating that any website whose hostname

Table 2.1. Flow results for a subset of the Alexa global 100 (last row summarizes results for all 100 sites).

Site and rank	Total KLOC	Other KLOC			# Taint Val(k)			Cookie		
		s	d	w	s	d	w	s	d	w
1. google	1.8	0	-	-	×	×	×	×	×	×
2. yahoo	7.4	7.0	-	0	×	×	×	×	×	×
3. facebook	9.1	9.1	9.1	9.1	×	×	×	×	×	×
4. youtube	7.5	7.3	7.3	5.7	✓	✓	✓	×	×	×
5. myspace	12.2	11.9	-	8.6	✓	✓	✓	×	×	×
6. wikipedia	<0.1	0	-	-	×	×	×	×	×	×
7. bing	0.7	0	-	-	×	×	×	×	×	×
8. blogger	1.8	1.1	-	0	✓	✓	×	×	×	×
9. ebay	13.6	13.4	-	12.9	✓	✓	✓	×	×	×
10. craigslist	0	0	-	-	×	×	×	×	×	×
11. amazon	5.3	4.8	-	-	×	×	×	×	×	×
12. msn	7.3	6.7	6.1	5.6	✓	×	×	×	×	×
13. twitter	5.6	5.5	-	1.3	×	×	×	×	×	×
14. aol	12.7	9.6	-	<0.1	✓	✓	×	×	×	×
15. go	1.1	0.9	0.2	-	✓	×	×	×	×	×
−. Average	8.2	6.1	4.5	3.0	48	38	17	0	0	0

is different from the hostname of the current one is considered a different site. (2) the *same-domain policy*, which is the same as the same-origin policy, except that websites from the same public domain suffix are considered to be the same (*e.g.,* `ads.cnn.com` is considered the same as `www.cnn.com`). (3) the *white-list policy*, which is the same as the same-domain policy, except that there is a global list of common websites that are considered the same as the source website. For our experiments, we treat websites referenced by three or more different Alexa benchmarks as safe. The white-list mainly consists of well-known statistics and advertisement websites. We use a whitelist only to evaluate the performance of our system under such conditions; we leave the exact criteria for trusting a given third-party site to future work. Our rewriting framework makes it trivial to consider different URL policies; we need only alter the notion of URL equality in the checks done inside `TSET.boxAndTaint` and `TSET.check`.

Detected Flows Table 2.1 shows the results of running our dynamic information frame-

work on the Alexa global top 100 list using the above policies. Because of space constraints, we only show a subset of the benchmarks, but the average row is for *all* 100 benchmarks.

The columns in the table are as follows: “Site and rank" is the name of the website and its rank in the Alexa global 100 list; “Total KLOC" is the number of lines of JavaScript code on each website, including code from other sites, as formatted by our pretty printer; “Other KLOC" is the number of lines of code from other sites; “# Taint Val" is the number of dynamically created taint values; “Cookie" describes the `document.cookie` confidentiality policy: ✓indicates policy violation, and ×indicates no flow *i.e.,* policy satisfaction. The column for the location intergrity policy is omitted since we have not found any violation of the policy on our benchmarks.

The above columns are sub-categorized into three subcolumns depending on the applied URL policy: “s" is for the same-origin policy; “d" is for the same-domain policy; “w" is for the white-list policy. A dash in a table entry means that the value for that table entry is the same as the entry immediately to its left.

The code for each website changes on each visit. Thus, we ran our enhanced Chromium 10 times on each website. To gain confidence in our tool, we manually inspected every program on which a flow is detected, and confirmed that every flow was indeed real.

Variation based on URL policies The number of lines of code from other sites decreases as we move from the same-origin policy to the same-domain policy to the white-list policy. Note that in some cases, for example `facebook.com`, code from other sites is almost the same as the total lines of code. This is because most of the JavaScript code for `facebook.com` comes from a website `fbcdn.net`. This website is not in the same domain as `facebook.com`, and it is only referenced by one website and hence, not included in our whitelist. In such situations, a *site-specific* white-list would help, but

we have not added such white-lists because it would be difficult for us to systematically decide for all 100 benchmarks what these white-lists should be. Thus, as we do not use site-specific white-lists, our policy violations may not correspond to undesirable flows.

As the amount of other-site code decreases as we move from "s" to "d" to "w", the number of dynamically created taint values also decreases, at about the same rate. That is, a large drop in other-site code leads to a correspondingly large drop in the number of taint values created. Moreover, as expected, the number of policy violations also decreases, as shown on the last line of the table: the violations of the `document.cookie` policies goes from 48 to 38 to 17. We did not see a violation of the `document.location` policy in any of our benchmarks.

2.3.2 Timing Measurements

Our rewrite-based information flow technique performs taint-tracking dynamically, and so it is important to evaluate the performance overhead of our approach. We measure performance using two metrics: total page load time, and JavaScript run time. We modified the Chromium browser to allow us to measure for each website (1) the time spent executing JavaScript on the site, and (2) the total time spent to download and display the site. Figures 2.3 describes our timing measurements for JavaScript time, and total download time on the 10 benchmarks with the largest JavaScript code bases. The measurements were performed while tracking both the `document.cookie` confidentiality and `document.location` integrity policies. The "average" benchmark represents the average time over all 10 benchmarks. For each benchmark there are five bars which represent running time, so smaller bars mean faster execution. For each benchmark, the 5 bars are normalized to the time for the unmodified Chromium browser for that benchmark. Above each benchmark we display the time in milliseconds for the unmodified Chromium browser (which indicates what "1" means for that benchmark).

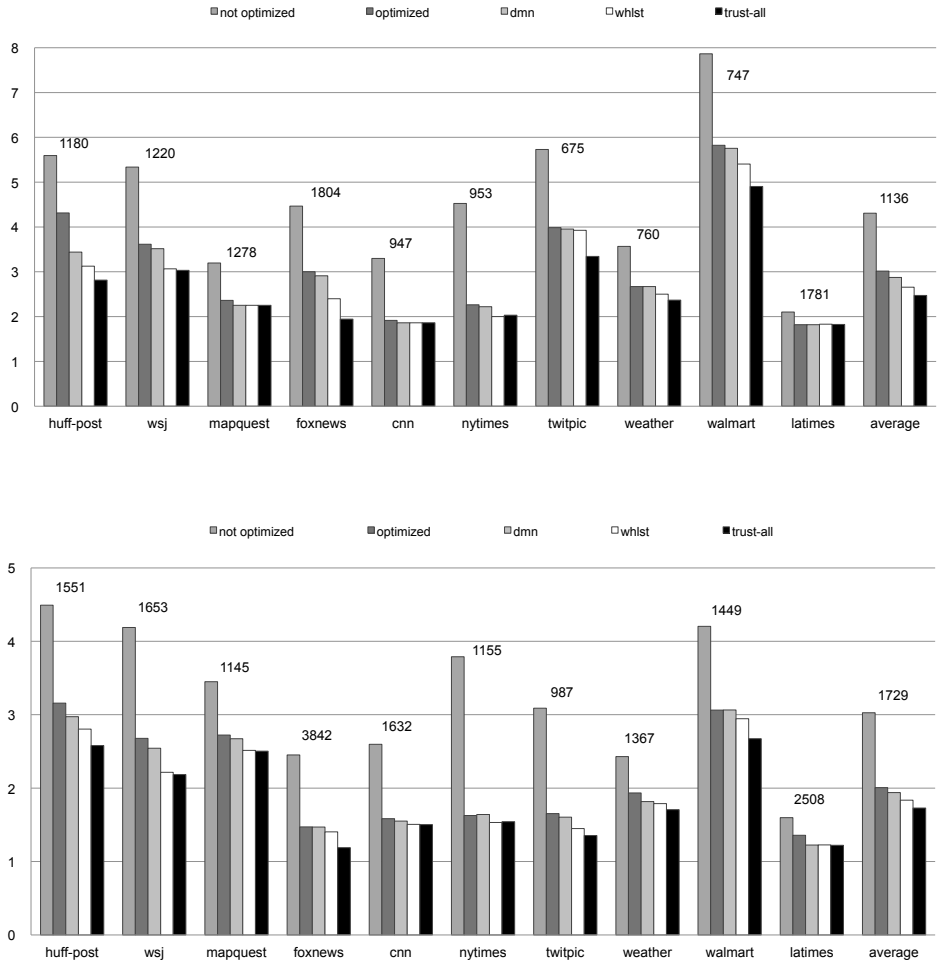


Figure 2.3. Slowdown for JavaScript (upper) and Page Loading (lower)

The left most bar "not-optimized" represents our technique using the original version of our `TSET` library, and using the same-origin URL policy. For the remaining bars, each bar represents a single change from the bar immediately to its left: "optimized" uses a hand-optimized version of our `TSET` library, rather than the original version; "dmn" changes the URL policy to same-domain; "whlist" changes the URL policy to white-list; and "trust-all" changes the URL policy to the trivial policy where *all* websites are trusted.

JavaScript execution time The top chart of Figure 2.3 shows just the JavaScript execu-

tion time. As expected, the bars get shorter from left-to-right; from "not-optimized" to "optimized", we are adding optimizations; and then the remaining bars consider progressively more inclusive URL policies meaning there are fewer taints to generate, propagate and check.

The data from Figure 2.3 shows that our original `TSET` library slows down JavaScript execution significantly – anywhere from about 2.1X to 7.9X, and on average about 4.3X. The optimized `TSET` library provides significant performance gains over the original library and provides 3.0X slowdown. The various white-lists provide some additional gain, but the gain is relatively small. To understand the limits of how much white-lists can help, we use the "trust-all" bar, which essentially corresponds to having a white-lists with every website on it. Overall, it seems that even in the best case scenario, white-lists do not help much in the overhead of our approach. This is because our approach needs to track the flow of `cookie` regardless of the number of external sites.

Total execution time The bottom chart of Figure 2.3 shows the *total* execution time of the enhanced Chromium while loading the web page and running the scripts on it. These measurements were collected on a fast network at a large university. The faster the network, the larger the overheads in Figure 2.3 will be, as the time to download the web page can essentially hide the overhead of running JavaScript. Thus, by using a fast network, Figure 2.3 essentially shows some of the worst case slowdowns of our approach. Here again, we see that the "optimized" bar is significantly faster than the "not-optimized" bar. We can also see that the "whlst" bar provides a loading experience that is about 73% slower.

2.4 Empirical Study of History Sniffing

Next, we present an empirical study of the prevalence of history sniffing on popular websites. In most browsers, all application domains share access to a single visited-page history, file cache, and DNS cache [49]. This leads to the possibility of history sniffing attacks [51], where a malicious site (say, `attacker.com`) can learn whether a user has visited a specific URL (say, `bankofamerica.com`), merely by inducing the user to visit `attacker.com`. To this end, the attack uses the fact that browsers display links differently depending on whether or not their target has been visited [24]. In JavaScript, the attacker creates a link to the target URL in a hidden part of the page, and then uses the browser's DOM interface to inspect how the link is displayed. If the link is displayed as a visited link, the target URL is in the user's history.

In essence, the attack works by inserting invisible links into the web page and having JavaScript inspect certain style properties of links, for example the color field, thereby determining whether the user has visited a particular URL. While researchers have known about the possibility of such attacks, hitherto it was not known how prevalent they are in real, popular websites. We have used our JavaScript information flow framework to detect and study the prevalence of such attacks on a large set of websites, and show that history sniffing is used, even by quite popular websites, and finally showcase the effectiveness of dynamic information flow as a detection mechanism for such information stealing JavaScript attacks.

Policies We formalize history sniffing in our framework using the following information flow policies:

at $1.`getComputedStyle`($2, ...) if $2.`isLink`() inject "`secret`"

at `send`($1, $2) block "`secret`" on $2

In particular, whenever the computed style of a link is read using `getComputedStyle`, the return value is marked as secret. Whenever a value is sent on the network using `send`, the second parameter (which is the actual data being sent) should not be tainted.

Benchmarks and Summary of Results To evaluate the prevalence of history sniffing, we ran our information flow framework using the above two policies on the front pages of the Alexa global top 50,000 websites.[2] To visit these sites automatically, we implemented a simple JavaScript web page that directs the browser to each of these websites. We successfully ran our framework on these sites in a total of about 50 hours. The slowdown for history sniffing was as follows: the JavaScript code slowed down by a factor 2.4X and total page loading time on a fast network increased by 67%. Overall, we found that of these 50,000 sites, 485 of them inspect style properties that can be used to infer the browser's history. Out of 485 sites, 63 are reported as transferring the browser's history to the network, and we confirmed that 46 of them are actually doing history sniffing, one of these sites being in the Alexa global top 100.

Real cases of history sniffing Out of 63 websites reported as transferring the browser's history by our framework, we confirmed that the 46 cases were real history sniffing occurrences. Table 2.2 lists these 46 websites. For each history-sniffing site, we give its Alexa rank, its URL, a description of the site, where the history-sniffing code comes from, and a list of some of the URLs inspected in the browser history.

Each one of the websites in Table 2.2 extracts the visitor's browsing history and transfers it to the network. Many of these websites seem to try to obfuscate what they are doing. For example, the inspected URLs on youporn.com are listed in the JavaScript source in encoded form and decoded right before they are used. On other websites, the history-sniffing JavaScript is not statically inserted in a web page, but dynamically generated in a way that makes it hard to understand that history sniffing is occurring by

[2]Here and elsewhere in the chapter we use the Alexa list as of February 1st, 2010.

```
 1:var k = { 0: "qpsoivc/dpn", 1: "sfeuvcf/dpn", ... };
 2:var g = [];
 3:for(var m in k) {
 4:  var d = k[m];
 5:  var a = "";
 6:  for(var f = 0; f < d.length; f++) {
 7:    a += String.fromCharCode(d.charCodeAt(f)-1)
 8:  }
 9:  var h = false;
10:  for(var j in {"http://":"","http://www.":""}) {
11:    var l = document.createElement("a");
12:    l.href = j + a;
13:    document.getElementById("ol").appendChild(l);
14:    var e = "";
15:    if(navigator.appName.indexOf("Microsoft") != -1 ) {
16:      e = l.currentStyle.color;
17:    } else {
18:      e = document.defaultView.getComputedStyle(l, null).
                                  getPropertyValue("color");
19:    }
20:    if(e == "rgb(12, 34, 56)" || e == "rgb(12,34,56)") {
21:      h = true
22:    }
23:  }
24:  if(h) { g.push(m) }
25:}
26:var b = (g instanceof Array)? g.join(",") : "";
27:var c = document.createElement("img");
28:c.src= "http://ol.youporn.com/blank.gif?id="+b;
29:document.getElementById("ol").appendChild(c)
```

Figure 2.4. Attack code as found on youporn.com

just looking at the static code. We also found that many of these websites make use of a handful of third-party history-sniffing libraries. In particular, of the 46 cases of confirmed sniffing, 22 sites use history-sniffing code from interclick.com and 14 use history-sniffing code from meaningtool.com.

Figure 2.4 shows the JavaScript attack code exactly as found on youporn.com. The code has an obfuscated list of inspected websites (line 1). We only show part of the list — the actual list had 23 entries. For each site, the code decodes the website name (line 6–8), creates a link to the target site on the page (lines 11–13), reads the color of the link that was just created (lines 15–19), and finally tests the color (line 20–22). If the

Table 2.2. Websites that perform real sniffing. “Src" is the source of the history sniffing JavaScript code: “I”, “M” and “F”, indicate the code came from interclick.com, meaningtool.com, and feedjit.com respectively, and “H” indicates the code came from the site itself.

Rank	Site	Desc	Src	Inspected URLs
61	youporn.com	adult	H	pornhub,tube8,+21
867	charter.net	news	I	cars,edmunds,+46
2333	feedjit.com	traffic	F	twitter,facebook,+6
2415	gamestorrents.com	fun	M	amazon,ebay,+220
2811	newsmax.com	news	I	cars,edmunds,+46
3508	namepros.com	forum	F	twitter,facebook,+6
3603	fulltono.com	music	M	amazon,ebay,+220
4266	youporngay.com	adult	H	pornhub,tube8,+21
4581	osdir.com	tech	I	cars,edmunds,+46
5233	gamesfreak.com	fun	I	cars,edmunds,+46
5357	morningstar.com	finance	I	cars,edmunds,+46
6500	espnf1.com	sports	I	cars,edmunds,+46
7198	netdoctor.com	health	I	cars,edmunds,+46
7323	narutocentral.com	fun	I	cars,edmunds,+46
8064	subirimagenes.com	hosting	M	amazon,ebay,+220
8644	fucktube.com	adult	H	tube8,xvideos,+9
9616	straightdope.com	news	I	cars,edmunds,+46
10152	guardafilm.com	movie	M	amazon,ebay,+220
10415	estrenosdtl.com	movie	M	amazon,ebay,+220
11330	bgames.com	fun	I	cars,edmunds,+46
12084	10best.com	travel	I	cars,edmunds,+46
12164	twincities.com	news	I	cars,edmunds,+46
16752	kaushik.net	blog	H	facebook,+100
17379	todocvcd.com	content	M	amazon,ebay,+220
17655	filmannex.com	movie	I	cars,edmunds,+46
17882	planet-f1.com	sports	I	cars,edmunds,+46
18361	trailersplay.com	movie	M	amazon,ebay,+220
20240	minyanville.com	finance	I	cars,edmunds,+46
20822	pixmac.com	hosting	H	istockphoto,+27
22010	fotoflexer.com	widget	I	amazon,ebay,+220
23577	xepisodes.com	fun	M	amazon,ebay,+220
23626	sincortespublicitarios.com	movie	F	facebook,youtube,+8
24109	mimp3.net.com	music	M	amazon,ebay,+220
24414	allaccess.com	news	I	amazon,ebay,+220
24597	petitchef.com	food	M	amazon,ebay,+220
24815	bleachcentral.com	fun	I	amazon,ebay,+220
25750	hoopsworld.com	sports	I	amazon,ebay,+220
27366	net-games.biz.com	fun	I	cars,edmunds,+46
31638	6speedonline.com	car	I	cars,edmunds,+46
34661	msgdiscovery.com	tech	M	amazon,ebay,+220
35773	moneynews.com	finance	I	cars,edmunds,+46
37333	answersingenesis.org	religion	H	facebook,+62
41490	divxatope.com	content	M	amazon,ebay,+220
45264	subtorrents.com	content	M	amazon,ebay,+220
48284	sesionvip.com	movie	M	amazon,ebay,+220
49549	youporncocks.com	adult	H	pornhub,tube8,+21

color indicates a visited link, the `h` variable is set to true, which in turn causes the link to be inserted in the list `g` of visited sites (line 24). This list is then flattened into a string (line 26), and the flattened string is concatenated as a query string into a URL pointing to

urlol.youporn.com, a subdomain of youporn.com, and finally sent to the network via the `src` name of an image (lines 27–29).

The flow in Figure 2.4 from the color property `e` to the array of visited sites `g` is actually an *indirect* flow that passes through two conditionals (on lines 20 and 24). Our framework's ability to track such indirect flows allowed us to find this history-sniffing attack. Note however that our framework found the flow because the sites being tested had actually been previously visited (because we had already run the experiments once, and so all the top 50,000 Alexa global sites were in the history). If none of the tested sites had been visited, the `g` array would have remained empty, and no violation of the policy would have been observed, even though in fact the user's empty history would have been leaked. This example is precisely the *implicit flow* limitation that was mentioned in Section 2.2.3.

False-positive cases of history sniffing Of the 63 sites flagged by our framework, 17 are false positives in that a manual examination of the source code and run-time behavior did not allow us to conclude that they were real cases of history sniffing. Out of these 17 sites, 12 contain JavaScript code that is too complicated to understand. The remaining 5 sites contain a history sniffing widget from interclick.com, but no suspicious runtime behavior was detected by monitoring their network access. Our framework reported these sites either because they inspected style properties for purposes other than history sniffing, or because too many irrelevant values were tainted by our handling of indirect flows.

A more stringent policy To investigate the possibility of history sniffing further, we also looked at all the sites that simply read the computed style of a link. This uncovered an additional 422 websites that read style properties of links, but did not send the properties out on the network. Unfortunately, because our framework does not cover all the corner cases of information flow in JavaScript (as discussed later), we cannot immediately

Table 2.3. Characteristics of suspicious websites depending on JavaScript widget provider.

Provider	Description	Sites	Inspected URLs
`addtoany`	social	83	120.2
`infolinks`	advertisement	124	14
`kontera`	advertisement	87	11.5
other	-	32	44.4

conclude that these sites did not transfer the browser history. Even if we were certain that the style information was not sent to the network, it is still possible that the *absence* of sending data was used to reveal information about the browsing history. For example, if a site sent the browsing history only if a link was visited, then the server could have learned about certain links' not being visited without any information's being transfered from the client. Thus, to better understand the behavior of these additional websites, we inspected them in detail, and categorized them into two bins: *suspicious websites*, and *non-suspicious websites*.

Suspicious sites Of the 422 sites, 326 sites exhibit what we would categorize as suspicious behavior. In particular, these suspicious websites inspect a large number of external links, and some of these links are dynamically generated, or they are located in an invisible iframe. We found that many of them embed a JavaScript widget developed by another website that inspects the browser history systematically.

Table 2.3 shows how such widgets are used on the 326 sites. For each JavaScript widget, we give the name of its provider, a description of its provider, the number of sites embedding it, and the number of URLs it inspects on average over the sites on which it is embedded. The most notable is a menu widget developed by addtoany.com which inspects around 120 URLs on average to activate or deactivate each menu item depending on the browser history.

Non-suspicious sites The remaining 96 sites seemed non-suspicious. Of these, 77 simply inspect their own website history. The remaining 19 samples have JavaScript code that is too complicated for us to fully understand, but where the sites seem non-suspicious.

Incompleteness Our current implementation would miss information flow induced by certain browser built-in JavaScript APIs. For example, consider the code:

```
arr.push(z); var result = arr.join(',')
```

The value `z` is inserted into an array and then all the elements of the array are joined into a string using the the built-in method `join`. Even though we have implemented a wrapper object for arrays to track array assignments and reads, we have not yet implemented a complete set of wrappers for all built-in methods. Thus, in the above case, even though `result` should be tainted, our current engine would not discover this. It would be straightforward, although time-consuming, to create precise wrappers for all built-in methods that accurately reflect the propagation of taints. Moreover, our current implementation does not track information flow through the DOM, although recent techniques on tracking information flow through dynamic tree structures [91] could be adapted to address this limitation.

Even if our implementation perfectly tracked the taints of all values through program execution, our approach would still miss certain history sniffing attacks. For example, the attacking website can use a style sheet to set the font of visited links to be much larger than the size of unvisited links. By placing an image below a link, and using JavaScript to observe where the image is rendered, the attacker can determine whether the link is visited or not. These kinds of attacks that use layout information would currently be very hard to capture using a taint-based information flow engine. Some attacks in fact don't even use JavaScript. For example, some browsers allow the style of visited links to be customized with a background image that is specific to that link, and this

background image can be located on the attacker's server. By observing which images are requested, the attacker can infer which links have been visited, without using any JavaScript. However, these history sniffing attacks are usually not as efficient as the ones via JavaScript detectable by our framework.

Despite all these sources of incompleteness, our JavaScript information flow framework can still be used as a diagnostic tool to find real cases of history sniffing via JavaScript, considered as the most dangerous one due to its efficiency. By running experiments on the Alexa global top 50,000 we have found that 46 sites really do perform history sniffing, and one of these sites is in the Alexa global top 100. We have also found several sites that have suspicious behavior, even though our current tool does not allow us to conclude with full certainty that these sites transfer the browser's history.

2.5 Empirical Study of Behavior Tracking

We have also conducted an empirical study on the prevalence of keyboard/mouse tracking on popular websites. JavaScript code can install handlers for events related to the mouse and keyboard to collect detailed information about what a user is doing on a given website. This information can then be transfered over the network. It is not enough to take a naive approach of simply prohibiting information from being transfered into the network while the event handler is being executed since the gathered information can be accumulated in a global variable, and then sent over the network in bulk (which is what we actually observed in our study).

Policies To use our information flow framework for detecting keyboard/mouse tracking,

we use the following policies in our framework:

at $1.isMouseOver() inject "secret"

at $1.isClick() inject "secret"

at $1.isScroll() inject "secret"

...

at document.send($1,$2) block "secret" on $2

Benchmarks and Summary of Results We ran our information flow framework using the above policies on the front pages of the Alexa global top 1,300 websites. One of the challenges in performing this empirical study automatically is that, to observe keyboard/mouse tracking, one has to somehow simulate keyboard and mouse activity. Instead of actually simulating a keyboard and mouse, we instead chose to automatically call event handlers that have been registered for any events related to the keyboard or mouse (click,mousemove,mouseover,mouseout,scroll,copy,select). To this end, in each web page we included a common piece of JavaScript code that automatically traverses the DOM tree of the current page and systematically triggers each handler with an event object that is appropriately synthesized for the handler. Another challenge is that many of the sites that track keyboard/mouse activity accumulate information locally, and then send the information in bulk back to the server at regular intervals, using timer events. These timer events are sometimes set to intervals spanning several minutes, and waiting several minutes per site to observe any flow would drastically increase the amount of time needed to run our test suite. Furthermore, it's also hard to know, a priori, how long to wait. To sidestep these issues, in addition to calling keyboard and mouse event handlers, we also automatically call timer event handlers. We successfully ran our framework on the Alexa top 1,300 websites in a total of about two hours.

Overall, we found 328 websites on which network transfers were flagged as transferring keyboard/mouse information to the network. Of these transfers, however, many are visually obvious to the user. In particular, many websites use mouse-over events to change the appearance of the item being moused-over. As an example, it is common for a website to display a different image when the mouse moves over a thumbnail (possibly displaying a larger version of the thumbnail). Although these kinds of flows can be used to track mouse activity, they are less worrisome because the user sees a change to the web page when the mouse movement occurs, and so there is a hint that something is being sent to the server.

Ideally, we would like to focus on *covert* keyboard/mouse tracking, in which the user's activities are being tracked without any visual cues that this is happening (as opposed to *visible* tracking where there is some visual cue).[3] However, automatically detecting covert tracking is challenging because it would require knowing if the keyboard/mouse activity is causing visual changes to the web page. Instead, we used a simple heuristic that we developed after observing a handful of sites that perform *visible* keyboard/mouse tracking. In particular, we observed that when the mouse/keyboard information is sent to the server because of a visual change, the server responds with a relatively large amount of information (for example a new image). On the other hand, we hypothesized that in *covert* tracking, the server would not respond with any substantial amount of data (if any at all). As a result, of all the network transfers found by our information flow tool, we filtered out those where the response was larger than 100 bytes (with the assumption that such flows are likely to be visible tracking). After this filtering, we were left with only 115 websites. We sampled the top 10 ranked websites among these 115 sites.

[3]One could view this heuristic as *charging* sites, in bandwidth, for the privilege of exfiltrating user attention data.

Table 2.4. Top 7 websites that perform real behavior sniffing.

Rank	Site	Description	Events
3	youtube	contents	click
11	yahoo.co.jp	portal	click
15	sina.com.cn	portal	click
19	microsoft	software	mouseover,click
34	mail.ru	email	click
53	soso	search engine	click
65	about	search engine	click

Real cases of covert tracking Of the 10 sites we sampled, we found that 7 actually perform covert keyboard and mouse tracking that we were able to reliably replicate. These 7 websites are listed in Table 2.4. For each site, we give its Alexa rank, its URL, a short description, and events being tracked covertly. One may be surprised to see "clicking" as being tracked covertly. After all, when a user clicks on a link, there is a clear visual cue that information is being sent over the network – the target of the link will know that the user has clicked. However, when we list clicking as being tracked covertly, we mean that there is an additional event-handler that tracks the click, and sends information about the click to another server. google is known for doing this: when a user clicks on a link on the search page, the click is recorded by google through an event handler, without any visual cue that this is happening (we do not list google in Table 2.4 because we only visit the front pages of websites, and google's tracking occurs on the search results page)

The most notable example in Table 2.4 is the microsoft.com site, which covertly tracks clicking and mouse behavior over many links on the front page and sends the information to the web statistics site webtrends.com.

Cases of visible tracking Of the 10 sites that were sampled, 3 were actually cases of visible tracking, despite our filtering heuristic. In one of these cases, the server responded

Table 2.5. Websites that perform real behavior sniffing using tynt.com.

Rank	Site	Description	Events
503	`thesun.co.uk`	news	copy, mouseover
548	`metrolyrics`	music	copy, mouseover
560	`perezhilton`	entertainment	copy, mouseover
622	`wired`	news	copy, mouseover
713	`suite101`	blog	copy, mouseover
910	`technorati`	blog	copy, mouseover
1236	`answerbag`	search engine	copy, mouseover

with very small images (less than 100 bytes) that were being redrawn in response to mouse-over events. In an other case, the server responded with small JSON commands that caused some of the web page to be redrawn. In all of these cases, there was a clear visual cue that the information was being sent to the server.

Cases of using tracking libraries Of the 115 sites on which the filtered flows were reported, we found that 7 used a behavior tracking software product developed by tynt.com to track what is copied off the sites. These 7 websites are listed in Table 2.5. The library monitors the copy event. When a visitor copies the content of a web page to her clipboard, the library inserts the URL of the page into the copied content. Thus, the URL is contained within subsequent pastes from the clipboard, *e.g.,* in emails containing the pasted text, thereby driving more traffic to the URL. Using our framework, we discovered that on each client website, the copied content is also transferred to tynt.com.

Suspicious website While investigating several sites that installed event handlers, we also found that the `huffingtonpost.com` site exhibits suspicious behavior. In particular, every article on the site's front page has an on-mouse-over event handler. These handlers collect in a global data structure information about what articles the mouse passes over. Despite the fact the information is never sent on the network, we still consider this case to be suspicious because not only is the infrastructure present, but it in fact collects the

information locally.

Acknowledgements

Chapter 2, in full, is a reprint of the material as it appears in Ehab Al-Shaer, Angelos D. Keromytis, Vitaly Shmatikov, editors, *Proceedings of CCS 2010.* Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham, ACM Press, October 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Securing C++ Virtual Function Calls

Several defenses have increased the cost of traditional, low-level attacks that corrupt control data, *e.g.,* return addresses saved on the stack, to compromise program execution in modern web browsers [101]. In response, creative adversaries have begun circumventing these defenses by exploiting programming errors to manipulate pointers to virtual tables, or *vtables*, of C++ objects. These attacks can hijack program control flow whenever a virtual method of a corrupted object is called, potentially allowing the attacker to gain complete control of the underlying system. In this chapter we present SAFEDISPATCH, a novel defense to prevent such *vtable hijacking* by statically analyzing C++ programs and inserting sufficient runtime checks to ensure that control flow at virtual method call sites cannot be arbitrarily influenced by an attacker. We implemented SAFEDISPATCH as a Clang++/LLVM extension, used our enhanced compiler to build a vtable-safe version of the Chromium browser, and measured the performance overhead of our approach on popular browser benchmark suites. By carefully crafting a handful of optimizations, we were able to reduce average runtime overhead to just 2.1%.

3.1 Motivation

Web browsers demand both performance and abstraction, making a low-level, object-oriented language like C++ the tool of choice for their implementation. Unfortu-

nately, this focus on performance has all too often taken precedence over critical security concerns. Malicious attacks frequently exploit the low-level programming errors that plague these systems, allowing an adversary to corrupt *control data*, pointers to code which the program later jumps to. By compromising control data, attackers are able to hijack program execution, in the worst case leading to arbitrary code execution.

Buffer overflows are one of the most familiar techniques for corrupting control data: by overwriting the return address in a function's activation record on the stack, the attacker can specify which instruction the CPU will jump to when the function returns, thus hijacking the program's execution. The security community has responded to such attacks with numerous defenses, including stack canaries [29], data execution prevention [70], and custom allocators to protect the heap [16]. These successful defenses have increased the cost of mounting traditional attacks, forcing adversaries to adopt increasingly sophisticated approaches.

Instead of overwriting return addresses saved on the stack, several recent, high profile attacks have shifted their focus to corrupting another class of control data: heap-based pointers to virtual tables, or *vtables*. A C++ class's vtable contains function pointers to the implementations for each of its methods. All major C++ compilers, including GCC, Visual C++, and LLVM, use vtables to implement dynamic dispatch: whenever an object invokes a virtual method, the vtable for that object's class is consulted to determine which function should be called. This layer of indirection enables polymorphism in C++ by allowing a subclass to invoke its own version of a method, overriding its parent class.

For performance, the first word of a C++ object with virtual methods is a pointer to its class's vtable. Unfortunately, this efficiency comes at a price: memory safety violations can nullify an important invariant: *the vtable pointer stored in an object of type τ always points to the vtable of τ or one of its subclasses*. If an attacker can corrupt an object's vtable pointer to instead point to a counterfeit vtable, then they can hijack

program control flow whenever that object calls one of its virtual methods, potentially executing malicious shellcode [88]. In this chapter, we call such attacks *vtable hijacking* and describe an efficient technique to prevent them.

Security researchers previously demonstrated one of the *many* ways an attacker can hijack vtables: by exploiting use-after-free errors. In this particular attack method, an adversary first identifies a dangling pointer, a reference to an object that has been freed. The attacker then tricks the program into (1) allocating a counterfeit vtable, and (2) overwriting the first word pointed-to by the dangling pointer with a pointer to *that* counterfeit vtable. Finally, the attacker manipulates the program to invoke a virtual method via the dangling pointer. Because the attacker has overwritten the vtable pointer in the freed object, this method call will jump to an address of the attacker's choosing, as specified by their counterfeit vtable. Exploiting such use-after-free errors is just one way to launch vtable hijakcing attacks, others include traditional buffer overflows on the stack or the heap [88] and type confusion [106, 33] attacks. Unfortunately, such vtable hijacking attacks are no longer merely a hypothetical threat particularly in web browsers [71, 72].

We increasingly observe robust vtable hijacking attacks in the wild, often leading to the execution of malicious shellcode. Such attacks have recently been shown practical in complex applications, including major web browsers: in recent Pwn2Own competitions, vtable hijacking enabled multiple arbitrary code execution attacks in Google Chrome [40], Internet Explorer [98], and Mozilla Firefox [109]. In fact, abusing dynamic dispatch in C++ was the major security weakness in all these browsers. In a recent Google Chrome exploit, Pinkie Pie employed a vtable hijacking attack to construct a Zero-day vulnerability to escape the renderer sandbox and execute arbitrary code [35]. As a result of such attacks, researchers have recently singled out vtable hijacking as one of the most straightforward attack vectors exploiting heap vulnerabilities, as an attacker

can often construct inputs to influence when a program allocates and frees objects and even their contents.

Unfortunately, existing defenses that could prevent vtable hijacking are either incomplete or do not specifically take advantage of the C++ type system to provide the best possible accuracy and performance. Techniques like reference counting can help mitigate vtable hijacking attacks that exploit dangling pointers, *e.g.,* by preventing dangling pointers from being used for invoking methods. Unfortunately, there are many other ways to mount vtable hijacking attacks that do not require a dangling pointer. Other techniques like control flow integrity [7, 117, 116, 115, 118] can secure all indirect jumps to prevent many kinds of control flow hijacking attacks, including vtable hijacking. However, these techniques do not take advantage of the C++ type system for the specific task of securing virtual method calls, and therefore none of these techniques treat C++ virtual method calls *both* precisely and efficiently.

In this chapter, we address the growing threat of vtable hijacking with SAFEDISPATCH, an enhanced C++ compiler that prevents such attacks. SAFEDISPATCH first performs a static class hierarchy analysis (CHA) to determine, for each class c in the program, the set of valid method implementations that may be invoked by an object of static type c. SAFEDISPATCH uses this information to instrument the program with dynamic checks, ensuring that, at runtime, all method calls invoke a valid method implementation according to C++ dynamic dispatch rules. By carefully optimizing these checks, we were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5% in the first vtable-safe version of the Google Chromium browser which we built with the SAFEDISPATCH compiler.

In this chapter, we explain the following contributions:

- We develop SAFEDISPATCH, a comprehensive defense against vtable hijacking attacks. We detail the static analysis and compilation techniques to efficiently

ensure control flow integrity through virtual method calls.

- We detail the implementation of SAFEDISPATCH as an enhanced C++ compiler and discuss several security and performance trade offs that influenced our design.

- We applied SAFEDISPATCH to the entire Google Chromium web browser code base to evaluate the effectiveness and efficiency of our approach. By developing a handful of carefully crafted optimizations, we were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5%.

In the next section we provide additional background on C++ dynamic dispatch and vtable hijacking and then overview how SAFEDISPATCH prevents such attacks. Section 3.3 follows, where we detail the SAFEDISPATCH compiler, key optimizations we developed to minimize overhead, and some of the different security and performance tradeoffs we considered. Next, in Section 3.6, we evaluate our SAFEDISPATCH implementation along several dimensions, including performance overhead, while in Section 3.7 we discuss the security implications of our approach.

3.2 SAFEDISPATCH Overview

In this section we provide additional background on dynamic dispatch in C++, illustrate vtable hijacking with a detailed example, and provide a high level description of how SAFEDISPATCH prevents such attacks.

3.2.1 Dynamic Dispatch in C++

Before detailing an example vtable hijacking attack, we briefly review how dynamic dispatch invokes object methods in C++. Consider the code in the upper part of Figure 3.1, which declares two classes: a `Window` class with one virtual method named

```
// for displaying content on screen
class Window: {
  public: virtual void display(string s) { ... }
};

// specialized for small screens on mobile devices
class MobileWin: public Window {
  public: virtual void display(string s) { ... }
};

Window* w = flag ? new Window() : new MobileWin();
w->display("Hello");      // invoke virtual method
delete w;                 // free w, now dangling
```

— — — — — — — — — — — — — — — — —

```
// behavior of code generated for w->display("Hello")
typedef void*   method;    // method is func ptr of any type
typedef method* vtable;    // vtable is array of methods
vtable t = *((vtable *)w); // 1. vtable @ 1st word of object
method m = t[0];           // 2. lookup by display's id, 0
m(w, "Hello");             // 3. make virtual call
```

Figure 3.1. C++ Dynamic Dispatch. Consider the simple `Window` class above for displaying a string on the screen. C++ compilers translate each virtual method call into lower level code that performs three steps: (1) dereference the first word of the calling object to retrieve its class's vtable pointer, (2) index into the vtable by the method's position in the class to retrieve the appropriate function pointer, and (3) call the retrieved function pointer, passing the calling object as the first argument, followed by any additional arguments. If an attacker corrupts an object's vtable pointer to point to a counterfeit vtable, possibly by exploiting a dangling pointer, then they can cause steps (1) and (2) to lookup malicious code and step (3) to execute it.

`display` for displaying a string on the screen and a `MobileWin` subclass of `Window` which overrides `display` to provide an implementation specialized for smaller screens.

C++ dynamic dispatch rules dictate that when an object calls a virtual method, the actual implementation invoked depends on the *runtime type* of the calling object. This layer of indirection allows subclasses to override their parent class's implementation of methods and is one of the key mechanisms for polymorphism in C++. For example, in the code snippet from Figure 3.1, the call `w->display("Hello")` will either invoke `Window::display` or `MobileWin::display`, depending on what `w` refers to at run-time, which in turn is determined by the `flag` variable.

Of the many implementation strategies for dynamic dispatch, *Virtual Method Tables*, or vtables are the most common. Prevalent C++ compilers, including GCC, Visual C++, and Clang++, all use vtables due to their efficiency. To implement vtables, the compiler assigns each virtual method in a class an identifier, which for simplicity we assume is done by numbering virtual methods sequentially. A vtable for class C is then an array t such that $t[i]$ is the implementation of method i for class C. At compile time, the compiler constructs a vtable for each class, and inserts code in the constructor of each class to initialize the first word of the constructed object with a pointer to the vtable for that class.

To implement a virtual method call the compiler generates code that performs three steps: (1) load the vtable pointer, located at position 0 in the calling object, (2) lookup index i in the vtable, where i is the index of the method being called (3) call the method implementation found at index i in the vtable. The lower part of Figure 3.1 uses C++ notation to illustrate the behavior of code generated for `w->display("Hi")`, assuming that `display` is given index 0 by the compiler. Note that if `w` points to a `Window` object, then the vtable will contain `Window::display` at location 0, whereas if `w` points to a `MobileWin` object, then the vtable will contain `MobileWin::display` at location 0.

Because vtables are used in determining control flow, if an attacker can illegally manipulate an object's vtable pointer, they can hijack program execution whenever that object invokes a virtual method. Since objects are ubiquitous in C++ programs, such control data is abundant, making vtable hijacking an attractive target for adversaries seeking to exploit low-level programming errors. We next illustrate how an attacker may mount such attacks.

3.2.2 vtable Hijacking

Having reviewed C++ dynamic dispatch, we now illustrate an example of vtable hijacking using the code in Figure 3.2. This code mimics the structure of a browser kernel in the style of OP [42] or Google Chrome [87, 15]. In these browsers, tabs run as separate, strictly sandboxed, processes whose only capability is communicating with the browser kernel process. To perform privileged operations, *e.g.,* rendering to the screen or initiating a network connection, a tab process must send requests to the browser kernel process which enforces access control for privileged operations. This architecture provides strong security properties: even fully compromising a tab does not immediately grant an attacker the ability to run arbitrary code since the tab sandbox prevents an exploited tab from performing any privileged operations. Of course, if the browser kernel contains an exploitable bug, the attacker may take full control of the underlying system.

The attack we demonstrate here assumes an adversary has already compromised a tab process which they now use to mount an attack against the highly privileged browser kernel. Although the code in this example is greatly simplified, a similar attack was central to Pinkie Pie's 2012 Zero-day exploit against Google Chrome [35]. Furthermore, while this example shows how vtable hijacking can be used to compromise a browser kernel, the approach generalizes to mounting attacks against many kinds of software, allowing an adversary to hijack program control flow, and thus potentially execute

```
class Shell {
  public: virtual string run(string cmd) { ... }
};

// for displaying content on screen
class Window: {
  public: virtual void display(string s) { ... }
};

// specialized for small screens on mobile devices
class MobileWin: public Window {
  public: virtual void display(string s) { ... }
};

void tab_request_handler_loop(void) {
  Shell*  sh  = NULL;
  Window* win = SMALL_SCREEN ? new MobileWin() : new Window();

  while (TRUE) {
    TabRequest r = recv_tab_request();
    switch (r.kind) {
      case GET_DATE:
        if (sh == NULL) sh = new Shell();
        // run shell with safe, const string
        string d = sh->run("date");
        send_tab_response(r.originating_tab, d);
        break;
      case DISPLAY_ALERT:
        win->display(r.msg);
        // equivalently:
        //   vtable t = *((vtable *)win);
        //   method m = t[0];
        //   m(win, r.msg)
        //
        // If the object that win points to was accidentally deleted, and a Shell object
        // was allocated in its place, then the above call invokes method 0 of Shell via
        // the dangling win ptr, namely "run" with a tab-controlled arg!
        break;
      case GET_HTML:
        ...
        // BUG: accidental delete, win ptr now dangling
        delete win;
        ...
        break;
    }
  }
}
```

— — — — — — — — — — — — — — — — —

```
// attack request sequence to run arbitrary shell command
GET_HTML, GET_DATE, DISPLAY_ALERT
```

Figure 3.2. Example vtable Hijacking. The above code sketches the core of a browser kernel in the style of Google Chrome: tabs run as separate, strictly sandboxed processes and send requests to the kernel to perform privileged operations like running shell commands or accessing the network. The main loop above illustrates how such a browser kernel responds to unprivileged tab requests. Due to a use-after-free error, an attacker can craft a sequence of requests causing the above code to run arbitrary shell commands.

malicious shellcode.

The core of Figure 3.2 depicts a loop inside the browser kernel to handle requests from unprivileged tab processes. For this simplified example, we consider three handlers which together enable a vtable hijacking attack that will allow an adversary to execute an arbitrary shell command.

The handler for `GET_DATE` uses a `Shell` object to execute a shell command which retrieves the system's date information, and then sends the result back to the requesting tab. Note that the parameter passed to `Shell::run` is a safe, constant string.

The handler for `DISPLAY_ALERT` renders a tab-provided string to the screen using a `Window` object. According to the C++ type system, at runtime this object will be an instance of `Window` or any of its subclass. In this case, there are two possibilities, either the `Window` class or the `MobileWin` class, which is specialized to render on smaller screens, and is used depending on the setting in the `SMALL_SCREEN` variable flag.

These two handlers alone do not contain an exploitable bug. However, we now introduce a third handler for `GET_HTML` requests which, somewhere in the process of fetching HTML for a tab-provided URL, inadvertently deletes the `Window` object pointed to by `win`, leaving the `win` pointer dangling.

The attack now consists of the adversary controlled tab sending three requests: `GET_HTML`, `GET_DATE`, and `DISPLAY_ALERT`. First, when kernel processes the `GET_HTML` request, the `win` object is accidently deleted. Second, when the kernel processes the `GET_DATE` request, a new `Shell` object is allocated. The memory allocator places this object *at the same memory location just freed by the previous handler* in a certain situation, leaving the dangling `win` pointer to refer to this newly allocated `Shell` object. Third, when the kernel processes the `DISPLAY_ALERT` request, the method call `win->display(r.msg)` dereferences the first word of `win` to get a vtable and calls the first function contained in that vtable. However, since `win` now points to a `Shell` object,

its vtable pointer refers to `Shell`'s vtable whose first element is the `run` method. Therefore, `win->display(r.msg)` actually calls `Shell::run` with `r.msg` as a parameter, a value provided by the attacker controlled tab. Thus, by sending these three requests in order, the compromised tab has tricked the kernel into running an arbitrary shell command, completely violating the kernel's security guarantee: the browser kernel's prime directive is to ensure all privileged operations are appropriately guarded, even in the face of a fully comprised tab processes.

This example illustrates just one of the *many* ways an attacker may mount a vtable hijacking attack. In addition to exploiting use-after-free errors, traditional buffer overflows (on the stack or heap), type confusion attacks, and vtable escape vulnerabilities are some of the techniques an attacker can employ to corrupt an object's vtable pointer and hijack program execution. We next sketch how SAFEDISPATCH prevents the attack shown in this example and consider the general case in subsequent sections.

3.2.3 SAFEDISPATCH vtable Protection

The attack illustrated in Figure 3.2 hijacks the control flow of the program through the `win->display(r.msg)` method call to trick the program into invoking `Shell::run(r.msg)` instead. To prevent such attacks, SAFEDISPATCH inserts code to check the integrity of control-flow transfers for virtual method calls. In particular, at each virtual method call site, SAFEDISPATCH inserts checks to ensure that the code being invoked is a valid implementation of the called method according the static type of the object being called. For example, Figure 3.3 sketches the code that SAFEDISPATCH generates to protect the call `win->display(r.msg)`. The additional checking code, shown in bold, guarantees that the method being called is either `Window::display` or `MobileWin::display`, which SAFEDISPATCH knows are the only two valid possibilities given the static type of `win`. This checking code not only prevents the previously described

```
// SafeDispatch protection for win->display(r.msg)
vtable t = *((vtable *)win);    // load vtable
method m = t[0];                // lookup method
if(m == Window::display ||
   m == MobileWin::display)     // check ensures m valid
  m(win, r.msg);
else                            // otherwise, signal error
  error("bogus method implementation!");
```

Figure 3.3. SAFEDISPATCH Protection. The SAFEDISPATCH compiler inserts checks at each method call site, analogous to those shown in **bold** above, to ensure that a method looked up from an object's vtable is valid given the object's static type, *i.e.,* that it is the requested method of the object's class or one of its subclasses. Since our `Window` class has one subclass which overrides `display`, there are two valid methods in this case, `Window::display` and `MobileWin::display`. This check ensures that control flow through method calls is valid under the C++ type system, effectively preventing the attacker from executing arbitrary code. We detail our general approach in Section 3.3.

attack, but also adds only minimal overhead compared to the existing dynamic dispatch code.

So far, we have shown how SAFEDISPATCH prevents an attack on a simple example. In the remainder of the chapter we explain how SAFEDISPATCH works in the general case, and present experimental results demonstrating that the overhead on complex, industrial scale applications is relatively low.

3.3 The SAFEDISPATCH Compiler

At their core, vtable hijacking attacks cause a virtual method call to jump into code which is not a valid implementation of that method. SAFEDISPATCH defends against all such attacks by instrumenting programs to *dynamically* ensure that, at every virtual method call site, the function pointer retrieved from the object's vtable is a valid implementation of the method being called (according to C++ dynamic dispatch rules), even if an attacker has managed to corrupt memory by exploiting a bug in the program.

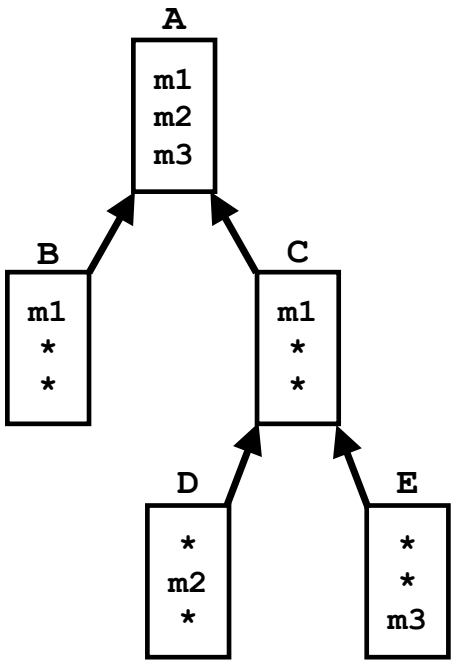
In this section we describe our implementation of SAFEDISPATCH as an enahnced C++ compiler, built on top of the Clang++/LLVM compiler infrastructure [62].

SAFEDISPATCH extends this infrastructure with three major passes to insert checks which protect a C++ program from vtable hijacking: (1) a variant of static Class Hierarchy Analysis [31] (CHA) which allows us to determine, *at compile time*, all the valid method implementations that may be invoked by an object of a particular static type at a given method call site, (2) a pass which uses the results from CHA to insert runtime checks that will ensure all method calls jump to valid implementations during program execution, and (3) various optimizations to reduce the SAFEDISPATCH runtime and code size overhead. We describe each of these three passes in more detail below.

3.3.1 Class Hierarchy Analysis

SAFEDISPATCH instruments a program to ensure all virtual method calls are valid at *runtime*, but before inserting these dynamic checks we must first determine, at *compile time*, which implementations are valid for each virtual method call site. Class Hierarchy Analysis [31] (CHA) is a static analysis that gathers this information by constructing the program's class hierarchy, *i.e.,* immediate subtyping relation, and then traverses this class hierarchy to compute the set of valid implementations for each virtual method of every class. The end result produced by CHA will be a map `ValidM` which gives us, for each class c and each virtual method n, the set $\texttt{ValidM}[c][n]$ of method implementations that could be invoked at runtime if an object with static type c were used to call method n.

Consider the example CHA results in Figure 3.4. In this case, the program being analyzed only contains five classes forming a three-layer hierarchy: `D` and `E` are subclasses of `C` while `B` and `C` are subclasses of `A`. Conceptually, this hierarchy is computed by creating a graph containing a node for each class in the program and then adding an edge from class c to c' whenever c extends c'. Each node also stores information about its class's methods, in particular indicating which implementations are inherited from parents (which we depict using $*$) and which the class overrides with its own



ValidM

	m1	m2	m3
A	A::m1 B::m1 C::m1	A::m2 D::m2	A::m3 E::m3
B	B::m1	A::m2	A::m3
C	C::m1	A::m2 D::m2	A::m3 E::m3
D	C::m1	D::m2	A::m3
E	C::m1	A::m2	E::m3

Figure 3.4. Example Class Hierarchy Analysis (CHA). Our Class Hierarchy Analysis is a static (compile time) analysis that uses the class hierarchy to compute which method implementations can be invoked by objects of each class type. The left diagram above shows an example hierarchy of five classes where subclasses point to their parent class: `D` and `E` are subclasses of `C` while `B` and `C` are subclasses of `A`. These classes have three methods: `m1`, `m2`, `m3`. In each class's box, we denote inheriting a parent's method implementation with `*` and list the names of overridden methods. For example, in this case `C` overrides `A`'s implementation of `m1`, but inherits the implementations of `m2` and `m3`. The results of our Class Hierarchy Analysis (CHA) is the `ValidM` table, specifying for each object type which implementations of a method may be invoked at runtime, according to C++ dynamic dispatch rules. In the example table above right, we see that calling method `m2` on an object pointed-to by a pointer of type `C` can invoke either class `A`'s or `D`'s implementation of `m2`.

```
// ValidM maps class C and method name N to the set of
// function pointers implementing N for C and its subclasses
map<class, map<string, set<method>> ValidM;

// computing ValidM at compile time
ValidM = new map<class, map<string, set<method>>();
foreach (class c in all_classes()) {
  ValidM[c] = new map<string, set<method>>();
  // all_method_names(c) returns all method names of class c,
  // including any methods inherited from parent classes
  foreach (string n in all_method_names(c)) {
    ValidM[c][n] = new set<method>();
    // all_subclasses(c) returns c and all its subclasses
    foreach (class sc in all_subclasses(c)) {
      // static_lkup(sc, n) returns the func ptr
      // implementing the method named n for an object of
      // class sc, according to C++ dynamic dispatch rules
      ValidM[c][n].add(static_lookup(sc,n));
    }
  }
}
```

Figure 3.5. Our CHA which constructs `ValidM` at Compile Time. At compile time SAFEDISPATCH performs CHA to construct `ValidM`, a table specifying for each method of each class type which implementations may legitimately be invoked at runtime. The SAFEDISPATCH compiler generates `ValidM` by iterating over all the program's classes. For each class *c*, SAFEDISPATCH considers all the names of *c*'s methods, including those transitively inherited from parent classes. For a given method name *n*, SAFEDISPATCH determines which implementations of *n* may be invoked at runtime by iterating over all of *c*'s (transitive) subclasses, including *c* itself. For each subclass *sc* of *c*, SAFEDISPATCH determines statically which implementation of *n* an *sc* object would invoke and adds it to the set of valid implementations in `ValidM[c][n]`.

implementation (which we depict using the method's name).

Our version of CHA analyzes, for each method *n* of each class *c*, which of *c*'s subclasses override *n* with their own implementation. Along with *c*'s (possibly inherited) implementation, the set of such method implementations are the only valid callees that may be invoked by an object of static type *c* when it calls *n* at runtime. This is made precise by the code shown in Figure 3.5, which computes this information and stores the result in a table called `ValidM`.

In practice, implementing CHA for large, complex applications like browsers poses a serious challenge, primarily due to subtle interactions between the many C++

inheritance mechanisms, *e.g.,* access modifiers, templates, virtual vs. non-virtual method properties, overloading, and multiple inheritance. To manage this complexity, we build on top of the Clang++ module responsible for constructing C++ vtables at compile time. Clang++ is an industrial strength compiler, capable of handling the tremendous complexity that arises in real-world C++ applications.

Precision and Scalability. SAFEDISPATCH uses CHA to determine, at compile time, which program locations a runtime method call may legitimately jump to. As a type-based analysis, CHA is relatively lightweight and scales up to large, complex applications. However, type-based analyses scale because they are generally coarse-grained and therefore less precise. It is possible that an object x stored in a variable of static type c only ever has runtime type c' where c' is a subclass of c. In such instances, CHA will overestimate the set of valid implementations x may invoke, including the implementation for c and all implementations in other subclasses of c, while in reality only the implementation in c' is called at runtime.

Such sources of imprecision could be remedied by using a more powerful static analysis. The additional precision would provide stronger security guarantees by further restricting an attacker's ability to invoke method implementations that should never arise during legitimate program execution. However, accurately tracking which classes flow to a particular variable x at compile time would require a precise whole program dataflow analysis. While such analyses exist, they often don't scale to the kinds of programs we aim to protect, leading to unacceptable increases in compile time. Those analyses that *can* scale in fact do so by giving up on precision, which would bring us back to square one. As a result, we feel that our type-based approach in CHA presents the best tradeoff by being precise enough to prevent real world attacks without dramatically increasing compile times.

We do note that CHA is fundamentally a whole program analysis, and thus

requires all an application's code to be available at compile time. Unfortunately, this currently precludes the use of separate compilation in our prototype implementation. However, our SAFEDISPATCH implementation is a research prototype and we feel that future work can address this limitation by annotating compiled object files with partial analysis results and composing those results to complete SAFEDISPATCH's program instrumentation at linktime.

3.3.2 SAFEDISPATCH Method Checking Instrumentation

After SAFEDISPATCH computes the CHA results, it can instrument the program with checks to ensure that whenever an object calls a virtual method, control jumps to one of the method implementations statically determined to be valid. Figure 3.6 shows how SAFEDISPATCH instruments each source level method call. For now, consider the basic strategy illustrated in part (A) of Figure 3.6. In the generated code for `o->x(args)`, after the implementation `m` for method name `"x"` has been looked up in the vtable dereferenced from `o`'s vtable pointer, SAFEDISPATCH inserts a call to `check(`*static_typeof(o)*`, "x", m)` *before* invoking `m`. This call to `check` consults the CHA results in `ValidM` to ensure that `m` is one of the valid implementations for `"x"` when called by an object which has `o`'s static type. Note that expressions in *italics* are evaluated *at compile time* as they require source-level information available only to the compiler. As shown in part (B) of Figure 3.6, SAFEDISPATCH also reduces runtime overhead by partially inlining calls to the `check` function, which we discuss in greater detail below.

Data Structures for Checking. The operation for checking method validity, `ValidM[c][n].contains(m)`, is critical for performance since it is inserted at every virtual method call site. Broadly speaking, SAFEDISPATCH uses an array of sets of valid method implementations to perform this validity checking. More specifically, for each pair (c,n) where c is a class and n is a method name, SAFEDISPATCH generates

```
// source level method call
o->x(args);
```

– – – – – – – – – – – – – – – – – – – –

```
// (A) generated code without check inlining
vtable t = *((vtable *)o);
method m = t[vtable_position(x)];
check(static_typeof(o), "x", m);
m(o, args);

// (B) generated code with check partially inlined
vtable t = *((vtable *)o);
method m = t[vtable_position(x)];
if (m != m1 && m != m2 && m != m3)
  check(static_typeof(o), "x", m);
m(o, args);
```

– – – – – – – – – – – – – – – – – – – –

```
void check(type c, string n, method m) {
  if (!ValidM[c][n].contains(m)) {
    error("bogus method implementation!");
  }
}
```

Figure 3.6. SAFEDISPATCH Instrumentation. At each method call site, SAFEDISPATCH inserts a check in the generated code to ensure that objects only invoke methods allowed by the static C++ type system. As shown in (A), the basic SAFEDISPATCH instrumentation simply adds a call to the `check()` function immediately before the jump to a method implementation. `check(`c,n,m`)` consults the `ValidM` table to ensure that function pointer m is a valid implementation of the method named n for objects with static type c. To avoid an extra function call at every method invocation, SAFEDISPATCH actually uses profiling information to partially inline `check()`. As shown in (B), SAFEDISPATCH inserts a branch to test if the function pointer looked up from the calling object's vtable is one of the most common valid implementations of the method used at this call site. If it is, SAFEDISPATCH safely skips the call to `check()`, thus avoiding the overhead of an additional function call in the common case. Note that all expressions in *italics* in the code above are evaluated *at compile time* as they require source-level information available only to the compiler.

at compile time a unique natural number $i_{(c,n)}$ which is used to index into a large array of sets. The set at position $i_{(c,n)}$, which contains the possible implementations for method n of class c, is represented as an unordered array of pointers to method addresses. Therefore `ValidM[c][n].contains(m)` involves an array lookup to retrieve `ValidM[c][n]`, followed by a linear scan through the resulting set. In our experiments we found that the average set size was very small (1.44 for method checking) and as result we do not expect that using a more elaborate data structure for representing these sets (e.g. a hash-set) would reduce the overhead significantly. Instead, we focus on other aggressive optimizations, for example the inlining of common checks, as explained in Section 3.3.3.

Externalizing Linktime Symbols. One subtlety of the method checking instrumentation is that the compiler does not statically know the concrete address where method implementations will be placed at linktime. It may seem that the SAFEDISPATCH compiler can handle this issue by simply referring to the linktime symbols for each method implementation. However, many modern C++ compilers restrict the linktime symbols for method implementations to only *internal* symbols, meaning that they cannot be referred to outside of code for their class. This poses a problem for SAFEDISPATCH as we need to check method implementation addresses wherever they may be called, not just in the class where they're defined. To address this issue, we *externalize* all linktime symbols for method implementations, allowing us to refer to them outside of their defining class. It would be straightforward to add an additional pass to check that these externalized symbols are only used in (1) internally by the defining class or (2) in SAFEDISPATCH instrumentation, together providing a guarantee equivalent to that of the unmodified C++ compiler.

3.3.3 SAFEDISPATCH Optimizations

To minimize SAFEDISPATCH's runtime overhead, we developed a handful of optimizations to reduce the cost of each check. Most importantly, we profile applications and partially inline the checks performed by the `check` function as shown in part (B) of Figure 3.6. This partial inlining compares the function pointer retrieved from an object's vtable against the concrete addresses of the `N` most common implementations of the method being called in profiling. In Figure 3.6 we limit `N` to just the three most common implementations, but in practice we can choose a value that balances the performance improvement of inlining against the increase in code size, which, in the worst case, could negatively impact instruction cache performance. In our actual experiments, discussed in Section 3.6, we inline all checks observed during profiling, which increases codesize, but did not present significant performance overhead for our benchmarks.

SAFEDISPATCH also performs *devirtualization*: in the case that CHA is able to statically determine there is a single valid method implementation at a given method call site, we rewrite the call to forgo vtable lookup and directly call the unique valid implementation. This avoids unnecessary memory operations to load the vtable and other computations to set up a virtual method call.

Now that we have inlined frequently executed checks, the high-level code in part (B) of Figure 3.6 still needs to be translated into low-level code. A direct naïve translation leaves room for two important optmizations, which we now describe. Consider again the code in part (B) of Figure 3.6, and let's look at a direct unoptimized translation to low-level code, as shown in part (A) of Figure 3.7. One source of overhead in this low-level code is that there are *two* opportunities for branch mis-prediction: one is to mis-predict which of the `if (..) goto L1` statements will fire; the second is to mis-predict where the indirect call through `m` will go (note that `m` is a function pointer). Our first low-level

```
// source level method call
o->x(args);
```

– – – – – – – – – – – – – – – – – – – –

```
// (A) direct unoptimized translation
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    if (m == m1) goto L;
    if (m == m2) goto L;
    if (m == m3) goto L;
    check(static_typeof(o), "x", m);
L:  setup_call_args(o, args);
    indirect_call m;
    ...

// (B) eliminate indirect calls
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    if (m == m1) goto L1;
    if (m == m2) goto L2;
    if (m == m3) goto L3;
    check(static_typeof(o), "x", m);
    setup_call_args(o, args);
    indirect_call m;
    goto LR;
L1: setup_call_args(o, args);
    direct_call m1;
    goto LR;
L2: setup_call_args(o, args);
    direct_call m2;
    goto LR;
L3: setup_call_args(o, args);
    direct_call m3;
    goto LR;
LR: ...

// (C) eliminate duplicate code
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    setup_call_args(o, args);
    if (m == m1) goto L1;
    if (m == m2) goto L2;
    if (m == m3) goto L3;
    check(static_typeof(o), "x", m);
    indirect_call m;
    goto LR;
L1: direct_call m1;
    goto LR;
L2: direct_call m2;
    goto LR;
L3: direct_call m3;
    goto LR;
LR: ...
```

Figure 3.7. Low-level SAFEDISPATCH Optimization. The code above illustrates low-level optimizations used in SAFEDISPATCH to eliminate branch misprediction for frequently called methods and to eliminate duplicate code for setting up method invocations. As in Figure 3.6, all expressions in *italics* above are evaluated *at compile time*.

optimization is that we can remove the second mis-prediction opportunity by placing a direct call once we know which of the three conditional has fired. This is shown in part (B) of Figure 3.7, where we now have direct calls for all checks that have been inlined. However, this code now has a lot of code duplication – namely all the setup for parameters. While this doesn't affect the number of instructions executed at run-time, it creates code bloat, which can have adverse effects on instruction-cache performance. Our second low-level optimization is that we hoist the duplicate code from inside the conditionals and use a single copy right before the conditionals, as shown in part (C) of Figure 3.7.

With all of the above optimizations, namely profile-based inlined checks and low-level optimizations, we were able to reduce the runtime overhead of SAFEDISPATCH to 2.1% and the codesize overhead to 7.5%. Section 3.6 will provide a more detailed empirical evaluation of the overheads of SAFEDISPATCH.

3.4 An Alternate Approach: Vtable Checking

The previous section showed how SAFEDISPATCH checks the control flow transfer at virtual method call sites. In this section, we present an alternate technique which establishes the same control-flow guarantee, but provides additional data integrity guarantees in the face of multiple inheritance, at the expense of additional runtime overhead. Later, in Section 3.6, we evaluate and compare the overhead of both approaches.

3.4.1 Pointer Offsets for Multiple Inheritance

To better explain this alternate approach, we first review vtables in more detail. In practice, vtables store more than just function pointers; they also contain offset values that are used to adjust the `this` pointer appropriately in the face of multiple inheritance.

For example, consider a class `C` that *virtually* inherits from both `A` and `B`. The

data layout of C objects will first include the fields from A, followed by the fields from B. Inherited methods from A will work unmodified on objects of type C because the offset of A's data fields are the same in A as in C. However, methods inherited from B will not work, because B's methods assume that B's fields start at the beginning of the object, whereas in C these fields are located *after* A's fields.

To address this problem, the compiler creates wrappers in C for methods inherited from B. Before calling B's original implementation of the method, the wrapper adjusts the calling object's `this` pointer by an appropriate offset so that it points to the B part of the C object. The situation is further complicated if C is subclassed again using additional multiple inheritance, in which case the layout for the fields inherited from A and B could change in the subclass of C. To address this problem, pointer offsets for `this` are stored in the vtable, so that the correct offset can be used at run-time depending on what class is being used to make the method call.

While our approach from Section 3.3 always protects against malicious control flow at virtual method call sites, it does not defend against an attacker counterfeiting a vtable with incorrect `this` pointer offsets. If an attacker successfully mounts such an attack, our previously described approach would still protect the control flow at virtual method calls, but the attacker could corrupt the `this` offset on entry to a method, potentially leading to further data corruption.

3.4.2 vtable Checking

To additionally protect `this` pointer offsets at method calls, we implemented an alternate vtable hijacking defense called *vtable checking*. Instead of checking the validity of the function pointer looked up from an object's vtable, we check the vtable pointer itself to ensure that it is valid given the static type of the calling object. In this way, we not only guarantee valid control flow at method calls, but also ensure that the offset value

of `this` is computed appropriately.

Figure 3.8 shows how each source level method call is instrumented in the vtable checking approach. As in Figure 3.6, expressions in *italics* are evaluated *at compile time* as they require source-level information available only to the compiler. We insert a check similar to the method checking instrumentation shown in Figure 3.6, but move the instrumentation earlier to check the *vtable* itself instead of the function pointer retrieved from it. In general, for code generated for method call `o->x(args)`, we insert a call to the `vt_check(`*`static_typeof(o)`*`, t)` after vtable `t` has been loaded from `o`'s vtable pointer. This call to `vt_check` consults the results of a modified CHA analysis to ensure that `t` is one of the valid vtables for an object of `o`'s static type. The computation for `ValidVT` is a modified, simpler version of the computation for `ValidM` described in the previous section, since the compiler already computes vtables. In particular, for each class `c` we collect the vtables for `c` and all of its subclasses, and store this entire set in `ValidVT[c]`. Similarly to method checking, the operation `ValidVT[c].contains(t)` is performed in two steps: `ValidVT[c]` is implemented as an array lookup and `contains(t)` is implemented using linear search. Here again, the average size of `ValidVT[c]` in our experiments was very small (2.58) and we reduce runtime overhead by selectively inlining calls to the `vt_check` function, taking advantage of profiling information as discussed in the previous section.

3.4.3 Performance Implications

The vtable checking approach described above provides a stronger security guarantee than the method checking approach described in the previous section, as it also ensures the integrity of `this` pointer offsets. Unfortunately, this stronger guarantee also incurs higher runtime overhead: since subclasses frequently inherit method implementations from their parent classes, at any virtual method call site, the number of valid vtables

```
// source level method call
o->x(args);
```

– – – – – – – – – – – – – – – – – – –

```
// generated code with vtable check partially inlined
vtable t = *((vtable *)o);
if (t != t1 && t != t2 && t != t3)
vt_check(static_typeof(o), t);
method m = t[vtable_position(x)];
m(o, args);
```

– – – – – – – – – – – – – – – – – – –

```
void vt_check(type c, vtable t) {
  if (!ValidVT[c].contains(t)) {
    error("bogus vtable!");
  }
}
```

Figure 3.8. Alternate SAFEDISPATCH vtable Checking. The instrumentation above illustrates an alternate vtable hijacking defense: checking the vtable pointer itself *before* using it to look up a method implementation. Similar to the approach shown in Figure 3.6, the SAFEDISPATCH instrumentation for this alternate strategy inserts a check in the generated code at each method call site, but in this case the check ensures that the calling object's vtable pointer agrees with the static C++ type system. The `vt_check(`c,t`)` function (analogous to the `check()` function discussed earlier) consults the `ValidVT` table (constructed from a modified CHA) to ensure that vtable t is a valid vtable for objects of c's static type. As in Figure 3.6, we partially inline this check using profiling information to avoid the overhead of an extra function call at most method invocations. Again, note that all expressions in *italics* in the code above are evaluated *at compile time* as they require source-level information available only to the compiler. This alternate has higher overhead in certain situations, but provides stronger data integrity guarantees in the face of multiple inheritance.

is *always* greater than or equal to the number of valid method implementations that can be invoked.

To better understand why this is the case, consider an example in which a class `A` declares method `foo`, and suppose there are many subclasses of `A`, none of which override `foo`. Now for any method call `x->foo()` where the static type of `x` is `A`, method checking just needs to compare against `A::foo`, since it is the only valid implementation of `foo`. On the other hand, vtable checking must compare against each vtable of the many subclasses of `A`, since each subclass has its own vtable. We explore the performance implications of this difference further in Section 3.6.

3.5 A Hybrid Approach for Method Pointers

In previous sections we described two vtable hijacking defenses, method checking and vtable checking, each presenting different tradeoffs. To best choose between these tradeoffs, we must consider additional subtleties arising from yet another C++ feature: *method pointers*. Conceptually, C++ method pointers are similar to traditional function pointers, except that pointers to virtual methods are *invoked by dynamic dispatch*, which means they could be exploited by vtable hijacking attacks and thus SAFEDISPATCH must also protect virtual calls through method pointers.

Figure 3.9 illustrates the behavior of C++ method pointers with two simple classes, `A` and `B`, where `A` contains a single method `foo` and `B` extends `A` and overrides `foo`. The method pointer `f` is declared to point to a method of an object of type `A` or one of `A`'s subclasses, and then `f` is assigned to point to `A::foo`. Next an `A` object is allocated and `A::foo` is called through the method pointer `f`. Afterward a `B` object is allocated and the *same method pointer*, `f`, is used to call one of the object's methods. However, in this case, control jumps to `B::foo` instead of `A::foo` since method pointers are *invoked by dynamic dispatch* : a method pointer is essentially an index into vtables of a class or the

```
class A {
  public: virtual void foo(int) { ... }
};

class B: A {
  public: virtual void foo(int) { ... }
};

void (A::*f)(int); // declare f as ptr to some method of A
f = &A::foo;       // f now points to the foo method

A* a = new A();
(a->*f)(5);        // method call via f ptr, invokes A::foo

a = new B();
(a->*f)(5);        // method call via f ptr, invokes B::foo
```

Figure 3.9. Method Pointer Example. Because C++ method pointers are invoked via dynamic dispatch, even though `f` is only assigned once, the first call above jumps to `A::foo` while the second jumps to `B::foo`.

class's subclasses.

To implement method pointer semantics, C++ compilers generate code which stores a vtable index in method pointers instead of the concrete address of a method's implementation. For example, if `foo` is placed at index `0` in the vtables of `A` and `B`, then the statement `f = &A::foo` will store the value `0` in `f`. When a call is made through a method pointer, the method pointer's value is used to index into the calling object's vtable to retrieve the appropriate method implementation to invoke.

3.5.1 Revisiting Previous Approaches

We now evaluate our previous two approaches, method checking and vtable checking, in the face of method pointers. First, consider our vtable checking technique from Section 3.4. Fortunately, vtable checking correctly handles method pointers with only a slight modification: since a method pointer is simply a vtable index and vtable checking guarantees the validity of vtables at runtime, SAFEDISPATCH simply checks that vtable indices from method pointers are within the valid range of methods for the given class, thus ensuring that method implementations retrieved by indexing into valid

vtables with a method pointer will also be valid. While simple, this modification is essential for preventing hijacking attacks through method pointers: if an attacker could arbitrarily set the method index to be out of range for the given class's vtable, they could cause a virtual method pointer call to jump to malicious code.

Second, consider our method checking technique from Section 3.3. In particular, consider a call through a method pointer of the form `(x->*f)(...)`, where the class used in the declaration of method pointer `f` is `C`. We must modify our method checking approach so that for such calls, the instrumentation checks, at runtime, that the function pointer extracted from the calling object's vtable is one of the implementations for *any* method of `C` or its subclasses, that satisfies the method pointer's argument type signature. This conservative approach can lead to a blow up in the number of required checks for large class hierarchies with many methods, like those found in modern web browsers. This effect is seen in Section 3.6 where we evaluate and further compare our different defenses. Unfortunately, improving on this approach would require a precise whole program dataflow analysis to compute which method implementations a pointer may point to. Despite decades of research, such analyses are difficult to scale to the large, complex applications most frequently targeted by vtable hijacking attacks.

3.5.2 Hybrid Approach

Comparing method checking and vtable checking in the face of method pointers leads to a key observation: at method pointer call sites, vtable checking typically requires many fewer comparisons than method pointer checking, since method pointer checking must compare against all method implementations from several classes. This situation is exactly the opposite from traditional method calls where vtable checking always demands at least as many comparisons as method checking, as discussed at the end of Section 3.4.

This observation suggests a hybrid approach: perform vtable checking (enhanced

with vtable index range checks) at method pointer call sites and method checking at traditional method call sites. We implemented this hybrid approach in SAFEDISPATCH and found that it incurs less runtime overhead than all other techniques, while providing the same strong security guarantees against vtable hijacking. We further discuss the performance implications of our hybrid approach in Section 3.6. At a member function call site, the numbers of method/vtable checks are compared, and vtable checks are used only when the number of the vtable checks is stricly less than the number of the method checks.

3.6 Evaluation

In this section we evaluate SAFEDISPATCH along three primary dimensions: (A) runtime and code size overhead, (B) effort to develop our prototype, and (C) compatibility with existing applications and programming practice.

3.6.1 SAFEDISPATCH Overhead

To evaluate the overhead of our SAFEDISPATCH defense, we used our enhanced C++ compiler to build a vtable-safe version of Google Chromium [15], a full-featured, open source web browser which forms the core of the popular Google Chrome browser [87]. Google Chromium is extremely large and complex, far larger than any SPEC benchmark; it contains millions of lines of production code, in diverse components (HTML renderer, JPEG decoder, Javascript JIT, IPC library, etc.) developed across multiple organizations (Google and various open source groups). Chromium serves as an ideal test case for SAFEDISPATCH: not only is it a complex, high performance C++ application with millions of users, but has also been targeted by several vtable hijacking attacks [35, 40].

Benchmarks. We measured SAFEDISPATCH overhead on Chromium over six de-

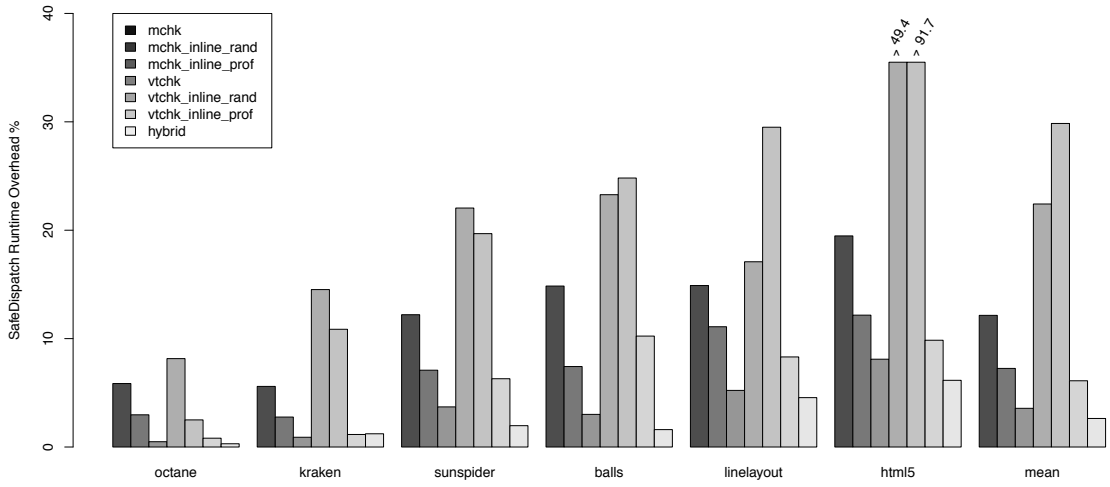


Figure 3.10. SAFEDISPATCH Overhead. We measured the overhead of SAFEDISPATCH on the Google Chromium browser over six demanding benchmarks: three industry standard JavaScript performance suites (octane, kraken, and sunspider) and three HTML rendering performance tests (balls, linelayout, and html5). All results are reported from the average of five runs, using percentage overhead compared to a baseline with no instrumentation. "mchk" is the unoptimized method pointer checking from Section 3.3, "vtchk" is the unoptimized vtable checking from Section 3.4. "inline_rand" indicates that we inline all checks that our Class Hierarchy Analysis tells us are needed for safety, but we inline them in a random order (*i.e.,* no profile information). "inline_prof" indicates that we inline the checks observed during profiling in order of how frequently they occur. "hybrid" is the hybrid approach from Section 3.5, which does profile-based inlining, but also combines method pointer checking and vtable checking. Note that two bars did not fit in the graph with the scale we chose for the y axis, namely "vtchk" and "vtchk_inline_rand" for html5; we shortened those bars, and show their values right on top of the bars (rather than change the scale and make all the other bars more difficult to read).

Table 3.1. SAFEDISPATCH Benchmarking Results and Code Size Overhead. The table above shows our benchmarking measurements for SAFEDISPATCH which correspond to the runtime overhead graph in Figure 3.10: "sp" stands for *sunspider*, for times reported in milliseconds, smaller is better, for other reported quantities (score, fps, and runs), larger is better. We additionally measured average code size increase due to SAFEDISPATCH data structures and instrumentation, and observed overheads typically well under 10%.

Instrumentation	**octane (score)**	**krkn (ms)**	**sp (ms)**	**balls (fps)**	**lnlayout (runs)**	**html5 (ms)**	**Code Size %**
none	15353	1556	254	14.95	106.26	3543	-
mchk	14454	1643	285	12.73	90.43	4233	7.20
mchk_inline_rand	14897	1599	272	13.84	94.47	3974	14.11
mchk_inline_prof	15278	1570	263	14.50	100.71	3830	7.48
vtchk	14101	1782	310	11.47	88.10	5294	7.31
vtchk_inline_rand	14969	1725	304	11.24	74.91	6793	44.18
vtchk_inline_prof	15228	1574	270	13.42	97.43	3892	7.85
hybrid	15299	1570	256	14.71	102.39	3721	7.48

manding benchmarks: three industry-standard JavaScript performance suites (*octane* [41], *sunspider* [11], and *kraken* [74]), and three HTML rendering performance tests (*balls*, *linelayout*, and *html5*). The three HTML rendering benchmarks are drawn from the WebKit performance test suite [111], the engine underlying several major web browsers including Google Chrome, Apple Safari, and Opera. We selected these benchmarks from the suite as three of the most important for performance and rendering correctness. We briefly describe the benchmarks below:

Our JavaScript benchmarks, *octane*, *kraken*, and *sunspider* are performance benchmarking suites from the Google Chrome, Mozilla Firefox, and Apple WebKit teams respectively. These benchmarks strive to measure real-world workloads and exercise many important browser functionality accessable to JavaScript. For octane we report the benchmark score where higher is better and for kraken and sunspider we measure running time in milliseconds where smaller is better.

balls creates thousands of small ball-shaped DOM elements, moves them around on the screen, measures how many of them can be moved in a fixed amount of time, and reports frames per second as its output. We report frames per second (fps); higher is better.

linelayout creates multiple DOM objects containing copious text. The renderer must draw many text lines, automatically inserting line breaks and allocating DOM objects efficiently on the screen, ensuring the renderer correctly handles the layout of DOM elements on the screen. We report number of complete runs in a fixed period; higher is better.

html5 performs millions of DOM manipulations to test numerous HTML5 features and is one of the most demanding WebKit performance tests. Each complex rendering is compared to an industry-standard reference rendering, thus ensuring optimizations have not introduced incorrect behavior. We report timing results in milliseconds; smaller is

better.

Runtime Overhead. Figure 3.10 presents the runtime overhead percentage of SAFEDISPATCH on benchmarks using a number of different approaches and optimizations, whereas Table 3.1 presents the raw numbers, including memory overhead. See the caption of Figure 3.10 for what each configuration of SAFEDISPATCH corresponds to (e.g., "mchk_inline_rand"). All our results are the average of five runs on an otherwise quiescent system running Ubuntu 12.04 on an Intel i7 Quad Core machine with 8GB of RAM.

From Figure 3.10, we can see that in general, all the "mchk" overheads are smaller than the "vtchk" overheads. This is consistent with the fact that, as described in Section 3.8, the number of valid vtables at a given method callsite is often 2x greater than the number of valid method implementations. Figure 3.10 shows the effectiveness of partial inlining of checks not just using profile information, but also using a random order. The random order is meant to capture the situation where we perform inlining, but we don't have profile information. We can see that inlining alone, without profile information ("mchk_inline_rand" and "vtchk_inline_rand") improves performance compared to the unoptimized instrumentation, but only for method checking. For vtable checking, the random-order inlining causes a slowdown because there were too many checks to inline, which affected performance negatively (this is confirmed by the memory overhead shown in Table 3.1. Inlining with profile information ("mchk_inline_prof" and "vtchk_inline_prof") provides a significant reduction in percentage overhead compared to the unoptimized instrumentation. Finally, Figure 3.10 also shows that that the hybrid approach from Section 3.5 has the lowest overhead by far, about 2% on average.

Cross Profiling. As shown above, profiling information can significantly reduce SAFEDISPATCH overhead. However, once deployed, applications are often run on inputs that were not profiled. To measure the effectiveness of profiling on one application and

Table 3.2. Cross Profiling. To evaluate the effect of profiling across benchmarks, we measured the overhead of running each binary optimized for one JavaScript performance suite on the other suites. The numbers reported are percentage overhead for the hybrid approach.

	Benchmark Overhead %		
Profile	**octane**	**kraken**	**sunspider**
octane	0.30	2.51	6.30
kraken	0.79	1.22	6.69
sunspider	1.15	2.25	1.97

running on another, we used each of the binaries optimized for each JavaScript benchmark and ran it on the others. We focused on JavaScript benchmarks for this cross-profiling evaluation because the rendering benchmarks each evaluate a different kind of rendering (*e.g.,* text, graphics, html rendering), and it would be unlikely that one of them would be a good predictor for others (in essence we would have to profile all three rendering benchmarks to get a representative set, but then this would not evaluate cross-profiling). Table 3.2 shows the results of cross-profiling for the hybrid approach. Each row and each column is a benchmark, and at row *y* and column *x*, we show the percentage overhead of running the *x* benchmark using the binary optimized for *y*'s profile information. While we can see that in some cases the overhead jumps to 6%, if we profile with *sunspider*, the overhead still remains in the vicinity of 2%. This may indicate that *sunspider* is a more representative Javascript benchmark, which is better suited for generating good profile information.

Code Size Overhead. We also measured the increase to code size resulting from SAFEDISPATCH data structures and instrumentation in the generated executable, shown in the final column of the table from Table 3.1. For the hybrid approach, the generated executable size was within 10% of the corresponding unprotected executable. Note that

Table 3.3. SAFEDISPATCH Prototype LOC. The table above characterizes the major components in our SAFEDISPATCH implementation. The basic instrumentation module is implemented as a Clang++ compiler pass and inserts calls to the `check()` fucntion as described in Section 3.3 function at each method call site, additionally logging some type data. These logs are used by the CHA module, written in Python, to build the `ValidM` and `ValidVT` used during checking at runtime. The final module is implemented as a set of low-level LLVM passes to inline checks based on profiling information.

Component	Framework	Language	LOC
Basic Instrumentation	Clang++	C++	177
Class Hierarchy Analysis	-	Python	691
Inlining Optimizations	LLVM	C++	381
Total			1249

the memory overhead for "vtchk_inline_rand" is substantial, which is consistent with the run-time overhead for "vtchk_inline_rand" from Figure 3.10.

3.6.2 Development Effort

Our prototype implementation of SAFEDISPATCH has three major components: (1) the basic instrumentation compiler pass, (2) CHA analysis to generate the `ValidM` and `ValidVT` internal SAFEDISPATCH checking data structures, and (3) inlining optimizations. The size of each component is listed in Table 3.3.

The basic instrumentation pass is implemented as a pass in Clang++ while the compiler has access to source-level type information which is erased once a program is translated into the lower level LLVM representation. This pass also produces information used in our second major component, the CHA analysis, which we implemented in a set of Python scripts to build the intermediate `ValidM` and `ValidVT` tables. Finally, we implemented our inlining passes as an optimization in LLVM which can take advantage of profiling information to order checking branches by how frequently they were taken in profile runs.

3.6.3 Compatibility

In principle, SAFEDISPATCH only incurs minimal compile time overhead to build the `ValidM` and `ValidVT` tables and instrument virtual method call sites as described in Sections 3.3, 3.4 and 3.5. Thus, in principle, the programmer should be able to use SAFEDISPATCH on every compilation without disrupting the typical edit, compile, test workflow. However, in our current prototype implementation, SAFEDISPATCH performs two full compilations to gather necessary analysis results before instrumenting the code, leading to a roughly 2x increase in compile time. As mentioned above, this is an artifact of our prototype implementation which can easily be fixed and is not an inherit limitation of SAFEDISPATCH.

The SAFEDISPATCH prototype also requires a whole-program CHA to perform instrumentation, and does not currently support separate compilation. There are two main challenges in supporting separate compilation. The first challenge is to make CHA modular. In particular, the compiler would have to generate CHA information per-compilation unit, which the linker would then combine into whole-program information. This approach to CHA is very similar to the approach taken in GCC's vtable verification branch [104, 103]. The second challenge is to inline checks in a modular way. In particular, editing code in one file could require additional checks in *another* file. To address this challenge, the compiler could insert calls to `check` at compile time, and then replace these calls with inserted inlined checks at link-time (similarly to link-time inlining of function calls). Finally, profiling data for inlining optimizations can be collected using a profile build in which the `check` function collects the required function/vtable pointers. This profile build can easily support separate compilation, as it does not require inlining or CHA.

3.7 SAFEDISPATCH Security Analysis

In this section we consider the security implications of SAFEDISPATCH including the class of attacks SAFEDISPATCH prevents and some limitations of our approach.

3.7.1 SAFEDISPATCH Guarantee

The checks inserted by the SAFEDISPATCH compiler guarantees that each virtual method call made at runtime jumps to a valid implementation of that method according to C++ dynamic dispatch rules. This guarantee immediately eliminates the attacker's ability to arbitrarily compromise the control flow of an application using a vtable hijacking attack. Our defense would prevent crucial steps in many recent, high profile vtable hijacking attacks, *e.g.,* Pinkie Pie's 2012 Zero-day exploit of Google Chrome which escaped the tab sandbox and allowed an adversary to compromise the underlying system. In addition to preventing many attacks, SAFEDISPATCH provides an intuitive guarantee in terms of the C++ type system, which is easy to understand for programmers who are familiar with the type system. Furthermore, the programmer cannot inadvertently nullify the SAFEDISPATCH guarantee through a programming mistake; the checks inserted by SAFEDISPATCH will detect errors such as incorrect type casts which would otherwise lead to a method call invoking an invalid method implementation.

The SAFEDISPATCH guarantee provides strong defense against vtable hijacking attacks, regardless of how the attack is mounted, *e.g.,* use-after-free error, heap based buffer overflow, type confusion, etc. As discussed further in the next section on related work, other defenses only focus on particular styles of attack (for example mitigating use-after-free errors by reference counting), or incur non-trivial overhead (for example using a custom allocator to ensure the memory safety properties necessary to prevent vtable hijacking via dangling pointers). Furthermore, SAFEDISPATCH protection is

always safe to apply: all programs should already satisfy the SAFEDISPATCH guarantee according to the C++ type system – we are simply enforcing it at runtime.

SAFEDISPATCH also defends against potentially exploitable, invalid typecasts made by the programmer [33]. If a programmer incorrectly casts an object of static type c to another type c' and at runtime the object does not have type c', then methods invoked on the object will not be valid implementation and SAFEDISPATCH will signal an error.

The astute reader may wonder why the checks inserted by SAFEDISPATCH instrumentation are any more secure than the vtable pointer stored in a runtime object. Unlike such heap pointers, the checks inserted by SAFEDISPATCH and their associated data structures are embedded in the generated executable which resides in *read-only memory*, ensuring that an attacker will not be able to corrupt SAFEDISPATCH inserted checks at runtime whereas vtable pointers stored in C++ objects reside in writable memory. Of course, this assumes the attacker will not be able to remap the program's text segment, or portion of memory containing the application's executable code, to be writable.

3.7.2 SAFEDISPATCH Limitations

SAFEDISPATCH guarantees that *one of* the valid method implementations for a given call site will be invoked at runtime, *not* that the correct method will be called. For example, an attacker could still corrupt an object's vtable pointer to point to the vtable of a child class, causing an object to invoke a child class's implementation of a method instead of it's own. While this call would technically satisfy the static C++ dynamic dispatch rules, it could lead to further memory corruption or other undesirable effects.

SAFEDISPATCH detects vtable pointer corruption precisely when it would result in an invalid method invocation. This does not prevent other memory corruption attacks, such as overwriting the return address stored in a function's activation record on the stack.

SAFEDISPATCH also does not currently prevent corrupting arbitrary (non-object) function pointer values. Such function pointers are important in systems making extensive use of callbacks or continuations. SAFEDISPATCH could be extended to protect such calls through function pointers by conceptually treating them as method invocations of a special ghost class introduced by the compiler. This change, which we will explore in future work, would also be transparent to the programmer and would further strengthen our guarantee.

SAFEDISPATCH only protects the code it compiles. Thus, if an application dynamically loads *unprotected* system libraries, an attacker may be able to compromise control flow within the library code via vtable hijacking. While such libraries can be compiled with SAFEDISPATCH to prevent such attacks, it's important to note that SAFEDISPATCH requires performing a whole program Class Hierarchy Analysis on the *entire program*, including all application libraries *and* all system libraries. Unfortunately, it is well known that such whole program analyses present challenges in the face of separate compilation, dynamically linked libraries, and shared libraries. As a result, our current SAFEDISPATCH prototype protects the entire application code, including all application libraries, but it does not protect shared system libraries such as the C++ standard library.

Dynamically linked libraries are also a possible source of incompatibility with the current SAFEDISPATCH prototype. For example, consider an application that uses a subclass implemented in an external, dynamically linked library. Since the subclass information is not statically available to SAFEDISPATCH's CHA, any such dynamically loaded subclass method implementations will be reported as invalid by `check` at runtime. To overcome this limitation, SAFEDISPATCH would be required to dynamically update its `ValidM` and `ValidVT` tables as dynamic libraries are loaded at runtime by instrumentation of certain system calls (e.g., `dlopen`). In future work, we hope to address this limitation

by developing better techniques for performing our CHA analysis in the face of separate compilation and dynamically linked libraries.

3.7.3 Performance and Security Tradeoffs

As discussed in previous sections, there are multiple strategies for enforcing the SAFEDISPATCH guarantee which lead to different security and performance tradeoffs. Vtable checking provides additional data integrity guarantees over method checking, in particular for `this` pointer offsets in the face of multiple inheritance, but at the cost of additional runtime overhead. Our hybrid approach adopts vtable checking at method pointer call sites to reduce runtime overhead, but uses method checking at non-method-pointer call sites, and so does not provide the same data integrity guarantees as vtable checking. Although the additional data integrity guarantee provided by vtable checking may mitigate some attacks, we feel that the significantly reduced overhead of our method checking and hybrid approaches offer a more realistic tradeoff for complex, high performance applications like web browsers.

Acknowledgements

Chapter 3, in full, is a reprint of the material as it appears in Lujo Bauer, editor, *Proceedings of NDSS 2014.* Dongseok Jang, Zachary Tatlock, and Sorin Lerner, Internet Society, February 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Formal Verification for Kernel-based Browsers

Despite the critical security role of web browsers, attackers routinely exploit implementation bugs in browsers to exfiltrate private data and take over the underlying system. In this chapter, we present QUARK, a browser structured with a centralized kernel that mediates security-critical resources access. QUARK's kernel implementation has been *formally verified* in Coq: we give a specification of our kernel, show that the implementation satisfies the specification, and finally show that the specification implies several security properties, including tab non-interference, cookie integrity and confidentiality, and address bar correctness.

The QUARK project was a joint project with another PhD student, Zachary Tatlock. The disseration author's primary contribution in QUARK was the design of QUARK's architecture, and the design and implementation of all the non-kernel components that interact with the kernel. Zachary Tatlock's primary contribution was the implementation and formalization of the Kernel in Coq.

4.1 Motivation

Security experts consistently discover vulnerabilities stemming from implementation bugs in all popular browsers, leading to data loss and remote exploitation. In the annual Pwn2Own competition, part of the CanSecWest security conference [4], security experts demonstrate new attacks on *up-to-date* browsers, allowing them to subvert a user's machine through the click of a single link. These vulnerabilities represent realistic, zero-day exploits and thus are quickly patched by browser vendors. Exploits are also regularly found in the wild; Google maintains a Vulnerability Reward Program, publishing its most notorious bugs and rewarding the cash to their reporters [2].

Researchers have responded to the problems of browser security with a diverse range of techniques, from novel browser architectures [15, 110, 42, 99, 69] and defenses against specific attacks [57, 45, 48, 14, 86] to alternative security policies [55, 96, 47, 14, 95, 8] and improved JavaScript safety [23, 52, 92, 10, 114]. While all these techniques improve browser security, the intricate subtleties of web security make it very difficult to know with full certainty whether a given technique works as intended. Often, a solution only "works" until an attacker finds a bug in the technique or its implementation. Even in work that attempts to provide strong guarantees (for example [42, 20, 99, 19]) the guarantees come from analyzing a model of the browser, not the actual implementation. Reasoning about such a simplified model eases the verification burden by omitting the gritty details and corner cases present in real systems. Unfortunately, attackers exploit precisely such corner cases. Thus, these approaches still leave a *formality gap* between the theory and implementation of a technique.

There is one promising technique that could minimize this formality gap: *fully formal verification of the browser implementation*, carried out in the demanding and foundational context of a mechanical proof assistant. This strict discipline forces the

programmer to specify precisely how their code should behave and then provides the tools to formally guarantee that it does, all in fully formal logic, building from basic axioms up. For their trouble, the programmer is rewarded with a *machine checkable proof* that the implementation satisfies the specification. With this proof in hand, we can avoid further reasoning about the large, complex implementation, and instead consider only the substantially smaller, simpler specification. In order to believe that such a browser truly satisfies its specification, one needs only trust a very small, extensively tested proof checker. By reasoning about the actual implementation directly, we can guarantee that any security properties implied by the specification will hold in every case, on every run of the actual browser.

Unfortunately, formal verification in a proof assistant is tremendously difficult. Often, those systems which we can formally verify are severely restricted, "toy" versions of the programs we actually have in mind. Thus, many researchers still consider full formal verification of realistic, browser-scale systems an unrealistic fantasy. Fortunately, recent advances in fully formal verification allow us to begin challenging this pessimistic outlook.

We demonstrate how our approach presented in this chapter, *formal shim verification*, radically reduces the verification burden for large systems to the degree that we were able to formally verify the implementation of a web browser, QUARK, within the demanding and foundational context of the mechanical proof assistant Coq.

At its core, formal shim verification addresses the challenge of formally verifying a large system by cleverly reducing the amount of code that must be considered; our approach restructure an existing system in a way that all components communicate through a small, lightweight shim which ensures the components are restricted to only exhibit allowed behaviors. Formal shim verification only requires one to reason about the shim, thus eliminating the tremendously expensive or infeasible task of verifying large,

complex components in a proof assistant.

Our web browser, QUARK, exploits formal shim verification and enables us to verify security properties for a *million* lines of code while reasoning about only a *few hundred*. To achieve this goal, QUARK is structured similarly to Google Chrome [15] or OP [42]. It consists of a small browser kernel which mediates access to system resources for all other browser components. These other components run in sandboxes which only allow the component to communicate with the kernel. In this way, QUARK is able to make strong guarantees about a million lines of code (*e.g.,* the renderer, JavaScript implementation, JPEG decoders, etc.) while only using a proof assistant to reason about a few hundred lines of code (the kernel). Because the underlying system is protected from QUARK's untrusted components (*i.e.,* everything other than the kernel) we were free to adopt state-of-the-art implementations and thus QUARK is able to run popular, complex websites like Facebook and GMail.

By applying formal shim verification to only reason about a small core of the browser, we formally establish the following security properties in QUARK, all within a proof assistant:

1. **Tab Non-Interference**: no tab can ever affect how the kernel interacts with another tab

2. **Cookie Same Domain**: cookies on a domain can only be accessed/modified by tabs of that domain

3. **Address Bar Integrity and Correctness**: the address bar cannot be modified by a tab without the user being involved, and always displays the correct address bar.

To summarize, our contributions are as follows:

- We demonstrate how formal shim verification enabled us to formally verify the implementation of a web browser. We discuss the techniques, tools, and design decisions required to formally verify QUARK in detail.
- We identify and formally prove key security properties for a realistic web browser.
- We provide a framework that can be used to further investigate and prove more complex policies within a working, formally verified browser.

The rest of the chapter is organized as follows. Section 4.2 presents an overview of the QUARK browser. Section 4.3 details the design of the QUARK kernel and its implementation. Section 4.4 explains the tools and techniques we used to formally verify the implementation of the QUARK kernel. Section 4.5 evaluates QUARK along several dimensions while Section 4.6 discusses lessons learned from our endeavor.

4.2 QUARK Architecture and Design

Figure 4.1 diagrams QUARK's architecture. Similar to Chrome [15] and OP [42], QUARK isolates complex and vulnerability-ridden components in sandboxes, forcing them to access sensitive system resources through a small browser kernel. Our kernel, written in Coq, runs in its own process and mediates access to resources including the keyboard, disk, and network. Each tab runs a modified version of WebKit in its own process. WebKit is the open source HTML rendering engine used in Chrome and Safari. It provides various callbacks for clients as Python bindings which we use to implement tab components. Since tab components should not directly access any system resources, we hook into these callbacks to re-route WebKit's network, screen, and cookie access through our kernel written in Coq. QUARK also uses separate components for displaying to the screen, storing and accessing cookies, as well reading input from the user. Each

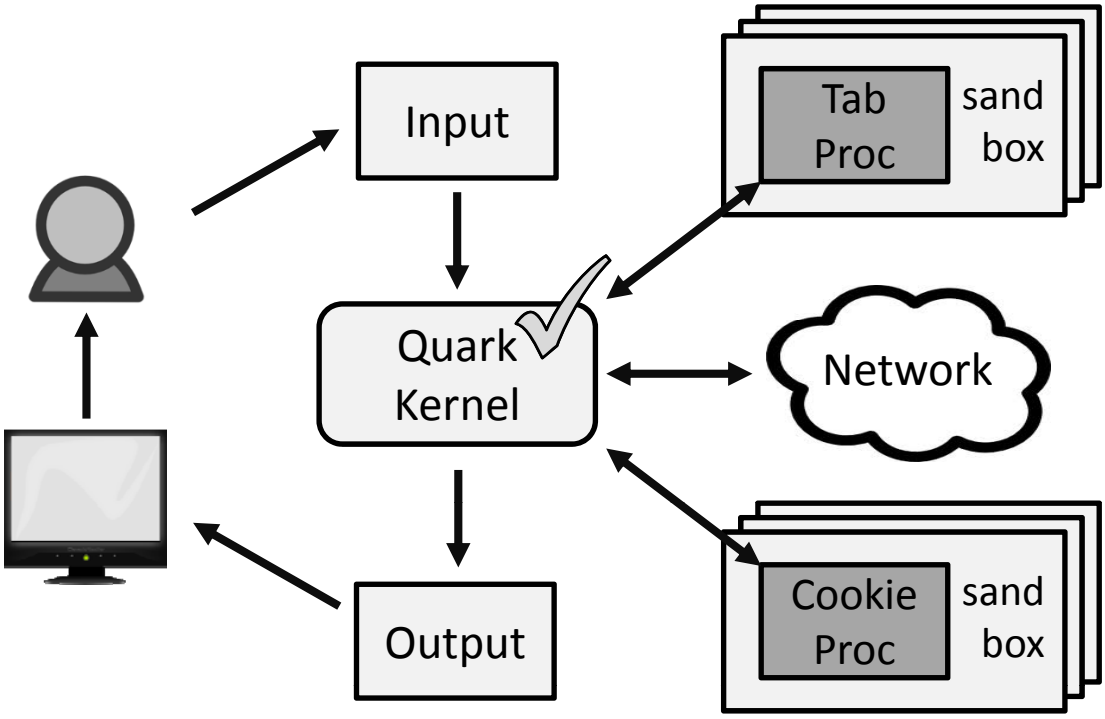


Figure 4.1. QUARK Architecture. This diagram shows how QUARK factors a modern browser into distinct components which run in separate processes; arrows indicate information flow. We guarantee our security properties by formally verifying the QUARK Kernel in the Coq proof assistant, which allows us to avoid reasoning about the intricate details of other components.

instance of the components are spawned as a separate, sandboxed process that is only allowed to send a message to the kernel.

Throughout this chapter, we assume that an attacker can compromise any QUARK component which is exposed to content from the Internet, except for the kernel which we formally verified. This includes all tab components, cookie components, and the graphical output component. Thus, we provide strong formal guarantees about tab and cookie isolation, even when some components have been completely taken over (*e.g.,* by a buffer overflow attack in the rendering or JavaScript engine of WebKit).

4.2.1 Graphical User Interface

The traditional GUI for web browsers manages several key responsibilities: reading mouse and keyboard input, showing rendered graphical output, and displaying the current URL. Unfortunately, such a monolithic component cannot be made to satisfy our security goals. If compromised, such a GUI component could spoof the current URL or send arbitrary user inputs to the kernel, which, if coordinated with a compromised tab, would violate tab isolation. Thus QUARK must carefully separate GUI responsibilities to maximize our security guarantees while still providing a realistic browser.

QUARK divides GUI responsibilities into several components which the kernel orchestrates to provide a traditional GUI for the user. The most complex component displays rendered bitmaps on the screen. QUARK puts this component in a separate process to which the kernel directs rendered bitmaps from the currently selected tab. Because the kernel never reads input from this graphical output process, any vulnerabilities it may have cannot subvert the kernel or impact any other component in QUARK. Furthermore, treating the graphical output component as a separate process simplifies the kernel and proofs because it allows the kernel to employ a uniform mechanism for interacting with the outside world: messages over channels.

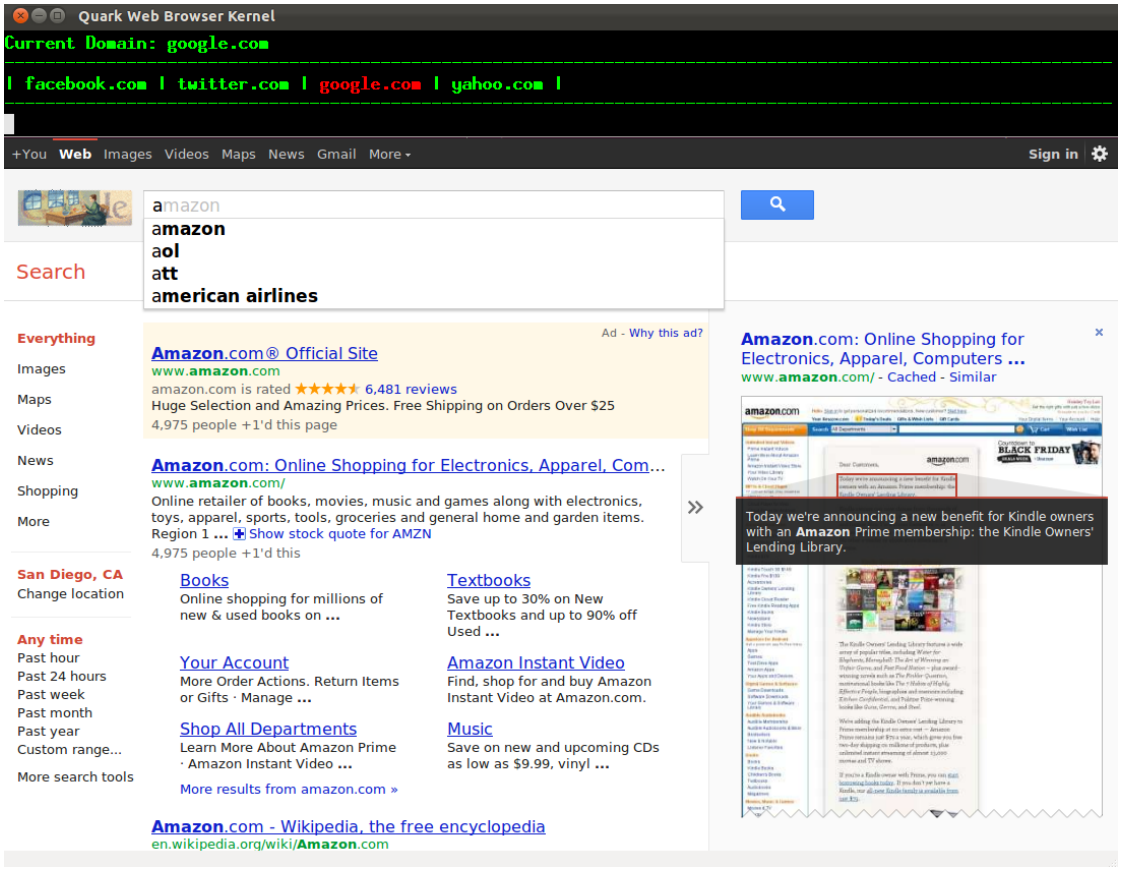


Figure 4.2. QUARK Screenshot. This screenshot shows QUARK running a Google search, including an interactive drop-down suggesting query completions and an initial set of search results from a JavaScript event handler dispatching an "instant search" as well as a page preview from a search result link.

To formally reason about the address bar, we designed our kernel so that the current URL is written directly to the kernel's `stdout`. This gives rise to a hybrid graphical/text output as shown in Figure 4.2 where the kernel has complete control over the address bar: we only trust that the simple terminal process correctly prints out the character stream from the kernel's `stdout` to correctly show the address to the user. With this design, the graphical output process is never able to spoof the address bar.

QUARK uses a separate input process to support richer inputs, *e.g.,* the mouse. The input process is a simple Python script which grabs keyboard and mouse events from the user, encodes them as user input messages, and forwards them on to the kernel. For keystrokes, the input process simply sends the ASCII code of the typed key to the kernel. Mouse clicks are sent to the kernel through un-printable ASCII values, interpreted as different mouse click events by the kernel. Because the input process only reads from the keyboard and mouse, and never from the kernel or any other QUARK components, it cannot be exposed to any attacks originating from the network. Furthermore, the user input is not mistakenly leaked to other components since the user input is only taken by the input process and directly sent to the kernel.

4.2.2 Example of Message Exchanges

To illustrate how the kernel orchestrates all the components in QUARK, we detail the steps from startup to a tab finishing loading http://www.google.com. The user opens QUARK by starting the kernel which in turn starts three processes: the input process, the graphical output process, and a tab process. The kernel establishes a two-way communication channel with each process it starts. Next, the kernel creates a tab process on the security domain "google.com" (see Section 4.2.4 and 4.2.5 for details of the tab security domain), and then sends a `(Go "http://www.google.com")` message to the tab indicating it should load the given URL (for now, assume this is normal behavior for

all new tabs).

The tab process comprises our modified version of WebKit wrapped by a thin layer of Python to handle messaging with the kernel. After recieving the `Go` message, the Python wrapper tells WebKit to start processing http://www.google.com. Since the tab process is running in a sandbox, WebKit cannot directly access the network. When it attempts to, our Python wrapper intervenes and sends a `GetURL` request to the kernel. As long as the request is valid under its security domain "google.com", the kernel responds with a `ResDoc` message containing the HTML document the tab requested.

Once the tab process has received the necessary resources from the kernel and rendered the Web pages, it sends a `Display` message to the kernel which contains a bitmap to display. When the kernel receives a `Display` message from the currently focused tab, it forwards the message to the graphical output process, which in turn displays the bitmap on the screen.

When the kernel reads a printable character `c` from standard input, it sends a `(KeyPress c)` message to the currently selected tab. Upon receiving such a message, the tab calls the appropriate input handler in WebKit. For example, if a user types "a" on Google, the "a" character is read by the kernel, passed to the tab, and then passed to WebKit, at which point WebKit adds the "a" character to Google's search box. This in turn causes WebKit's JavaScript engine to run an event handler that Google has installed on their search box. The event handler performs an "instant search", which initiates further communication with the QUARK kernel to access additional network resources, followed by another `Display` message to repaint the screen. Note that to ease verification, QUARK currently handles all requests synchronously.

4.2.3 Efficiency

With a few simple optimizations, we achieve performance comparable to WebKit on average (see Section 4.5 for measurements). Following Chrome, we adopt two optimizations critical for good graphics performance. First, QUARK uses shared memory to pass bitmaps from the tab process through the kernel to the output process, so that the `Display` message only passes a shared memory ID instead of a bitmap. This drastically reduces the communication cost of sending bitmaps. To prevent a malicious tab from accessing another tab's shared memory, we run each tab as a different user, and set access controls so that a tab's shared memory can only be accessed by the output process. Second, QUARK uses *rectangle-based* rendering: instead of sending a large bitmap of the entire screen each time the display changes, the tab process determines which part of the display has changed, and sends bitmaps only for the rectangular regions that need to be updated. This drastically reduces the size of the bitmaps being transferred, and the amount of redrawing on the screen.

The Ynot library [77] QUARK uses only has single-character read/write routines, imposing significant overhead since every single character read/write needs to be separately handled by the operating system kernel. For better I/O performance, we defined a new I/O library which uses size `n` reads/writes. This reduced reading an `n` byte message from `n` I/O calls to just three: reading a 1 byte tag, followed by a 4 byte payload size, and then a single read for the entire payload.

We also optimized socket connections in QUARK. Our original prototype opened a new `TCP` connection for each HTTP `GET` request, imposing significant overhead. Modern Web servers and browsers use *persistent connections* to improve the efficiency of page loading and the responsiveness of Web 2.0 applications. These connections are maintained anywhere from a few seconds to several minutes, allowing the client and

server can exchange multiple request/responses on a single connection. Services like Google Chat make use of very long-lived HTTP connections to support responsive interaction with the user.

We support such persistent HTTP connections via Unix domain sockets which allow processes to send open file descriptors over channels using the `sendmsg` and `recvmsg` system calls. When a tab needs to open a socket, it sends a `GetSoc` message to the kernel with the host and port. If the request is valid, the kernel opens and connects the socket, and then sends an *open* socket file descriptor to the tab. Once the tab gets the socket file descriptor, it can read/write on the socket, but it cannot re-bind the socket to another host/port. In this way, the kernel controls all socket connections.

Even though we formally verify our browser kernel in a proof assistant, we were still able to implement and reason about these low-level optimizations.

4.2.4 Socket Security Policy

The `GetSoc` message brings up an interesting security issue. If the kernel allowed for all `GetSoc` requests no matter what destination they are connected to, then a compromised tab could open sockets to any server and exchange arbitrary amounts of information. This arbitrary communication completely violates the same origin policy and can be exploited to bypass the local firewall. Therefore, the kernel must prevent this scenario by restricting socket connections.

To implement this restriction, we introduce the idea of a *domain suffix* for a tab which the user enters when the tab starts. A tab's domain suffix controls several security features in QUARK, including which socket connections are allowed and how cookies are handled (see Section 4.2.5). In fact, our address bar, located at the very top of the browser (see Figure 4.2), displays the domain suffix, not just the tab's URL. We therefore refer to it as the "domain bar".

For simplicity, our current domain suffixes build on the notion of a *public domain suffix*, which is a top-level domain under which Internet users can directly register names, for example .com, .co.uk, or .edu – Mozilla maintains an exhaustive list of such suffixes [3]. In particular, we require the domain suffix for a tab to be exactly one level down under a public suffix, *e.g.,* google.com, amazon.com, etc. In the current QUARK prototype the user provides a tab's domain suffix separately from its initial URL, but one could easily compute the former from the later. Note that, once set, a tab's domain suffix never changes. In particular, any frames a tab loads do not affect its domain suffix.

We considered using the tab's origin (which includes the URL, scheme, and port) to restrict socket creation in an attempt to enforce the same origin policy in the kernel, but such a policy is too restrictive for many useful sites. For example, a single GMail tab uses frames from domains such as static.google.com and mail.google.com. However, our actual domain suffix checks are modularized within QUARK, which will allow us to experiment with finer grained policies in future work.

To enforce our current socket creation policy, we first define a subdomain relation $\leq$ as follows: given domain d_1 and domain suffix d_2, we use $d_1 \leq d_2$ to denote that d_1 is a subdomain of d_2. For example www.google.com $\leq$ google.com. If a tab with domain suffix t requests to open a connection to a host h, then the kernel allows the connection if $h \leq t$. To load URLs that are not a subdomain of the tab suffix, the tab must send a `GetURL` message to the kernel – in response, the kernel does not open a socket but, if the request is valid, may provide the content of the URL. Since the kernel does not attach any cookies to the HTTP request for a `GetURL` message, a tab can only access publicly available data using `GetURL`. In addition, `GetURL` requests only provide the response body, not HTTP headers.

Note that an exploited tab could leak cookies by encoding information within the URL parameter of `GetURL` requests, but only cookies for that tab's domain suffix could

be leaked : a tab cannot get cookies via GetURL since we do not provide any access to HTTP headers with GetURL. In addition, a tab whose domain suffix is google.com can only access cookies for *.google.com under our cookie policy 4.2.5. For exmaple, an exploited tab on google.com can steal cookies for www.google.com, but not for www.facebook.com.

We also slightly enhanced our socket policy to improve performance. Sites with large data sets often use content distribution networks whose domains will not satisfy our domain suffix checks. For example facebook.com uses fbcdn.net to load much of its data. Unfortunately, the simple socket policy described above will force all this data to be loaded using slow `GetURL` requests through the kernel. To address this issue, we associate *whitelists* with the most popular sites so that tabs for those domains can open sockets to the associated content distribution network. The tab domain suffix remains a single string, e.g. facebook.com, but behind the scenes, it gets expanded into a list depending on the domain, *e.g.,* [facebook.com, fbcdn.net]. When deciding whether to satisfy a given socket request, QUARK considers this list as a disjunction of allowed domain suffixes. Currently, we provide these whitelists manually.

4.2.5 Cookies and Cookie Policy

QUARK maintains a set of cookie processes to handle cookie accesses from tabs. This set of cookie processes will contain a cookie process for domain suffix *S* (see Section 4.2.4) if *S* is the domain suffix of a running tab. By restricting messages to and from cookie processes, the QUARK kernel guarantees that tab processes will only be able to access cookies appropriate for their domain suffix.

The kernel receives cookie store/retrieve requests from tabs and directs the requests to the appropriate cookie process based on their domain suffix. If a tab with domain suffix *t* asks to store a cookie string with domain *c*, then our kernel allows the

operation if $c \leq t$, in which case it sends the store request to the cookie process for domain suffix t. Similarly, if a tab with domain suffix t wants to retrieve a cookie for domain c, then our kernel allows the operation if $c \leq t$, in which case it sends the request to the cookie process for domain suffix t and forwards any response to the requesting tab.

The above policy prevents cross-domain-suffix cookie reads from a compromised tab, and it prevents a compromised cookie process from leaking information about its cookies to another domain suffix; yet it also allows different tabs with the same domain suffix (but different URLs) to communicate through cookies (for example, mail.google.com and calendar.google.com).

4.2.6 Security Properties of QUARK

We provide intuitive descriptions of the security properties we proved for the kernel of QUARK; formal definitions appear later in Section 4.3. A tab in the kernel is a pair, containing the tab's public domain suffix as a string and the tab's communication channel as a file descriptor. A cookie process is also a pair, containing the public domain suffix that this cookie process manages and its communication channel to the kernel. We define the state of the kernel as the currently selected tab, the list of tabs, and the list of cookie processes. Note that the kernel state only consists of strings and file descriptors.

We prove the following main theorems in Coq:

1. **Response Integrity**: The way the kernel responds to any request only depends on past user "control keys" for tab creation/switching (keys F1-F12 in our prototype). This ensures that one browser component (*e.g.,* a tab or cookie process) can never influence how the kernel responds to another component, and that the kernel never allows untrusted input (*e.g.,* data from the web) to influence how the kernel responds to a request.

2. **Tab Non-Interference**: The kernel's response to a tab's request is the same no matter how other tabs of a different domain suffix interact with the kernel. This ensures that the kernel never provides a direct way for one tab to attack another tab or steal private information from another tab across their domain suffix.

3. **Same Domain Suffix Socket Creation**: The kernel disallows any cross-domain socket creation (as described in Section 4.2.4).

4. **Same Domain Suffix Cookie Access**: The kernel disallows any cross-domain-suffix cookie stores or retrieves (as described in Section 4.2.5).

5. **Domain Bar Integrity and Correctness**: The domain bar cannot be compromised by any browser component, and is always equal to the public domain suffix of the currently focused tab.

4.3 Kernel Implementation in Coq

QUARK's most distinguishing feature is its kernel implemented and proved correct in Coq. In this section we present the implementation of the main event handling loop of the kernel. In the next section we explain how we formally verified the kernel.

Coq enables users to write programs in a small, simple functional language and then reason formally about them using a powerful logic, the Calculus of Constructions. This language is essentially an effect-free (pure) subset of popular functional languages like OCaml with the additional restriction that programs must always terminate. Unfortunately, these limitations make Coq's default implementation language ill-suited for writing system programs like servers or browsers which must be effectful to perform I/O and by design may not terminate.

```
Definition kstep(ctab, ctabs) :=
  chan <- iselect(stdin, tabs);
  match chan with
  | Stdin =>
      c <- read(stdin);
      match c with
      | F12 =>
          t <- mktab();
          write_msg(t, Render);
          return (t, t::tabs)
      | ...
      end
  | Tab t =>
      msg <- read_msg(t);
      match msg with
      | GetSoc(host, port) =>
          if(safe_soc(host, domain_suffix(t)) then
            send_soc(t, host, port);
            return (ctab, tabs)
          else
            write_msg(t, Error);
            return (ctab, tabs)
      | ...
      end
  end
```

Figure 4.3. Body for Main Kernel Loop. This Coq code shows how our QUARK kernel receives and responds to requests from other browser components. It first uses a Unix-style select to choose a ready input channel, reads a request from that channel, and responds to the message appropriately. For example, if the user enters function key F12, the kernel creates a new tab and sends it the `Render` message. In each case, the code returns the new kernel state resulting from handling this request.

To address the limitations of Coq's implementation language, we use Ynot [77]. Ynot is a Coq library which provides monadic types that allow us to write effectful, non-terminating programs in Coq while retaining the strong guarantees and reasoning capabilities Coq normally provides. Equipped with Ynot, we can write our browser kernel in a fairly straightforward style whose essence is shown in Figure 4.3.

Single Step of Kernel QUARK's kernel is essentially an infinite loop whose body reactively responds to a request from the user or tabs. In each iteration, the kernel calls `kstep` which takes the current kernel state, handles a single request, and returns the new

kernel state as shown in Figure 4.3. The kernel state is a tuple of the currently focused tab (`ctab`), the list of tabs (`tabs`), and a few other components which we omit here (*e.g.,* the list of cookie processes). For details regarding the loop and kernel initialization code please see [54].

The function `kstep` starts by calling `iselect` (the "i" stands for input) which performs a Unix-style select over `stdin` and all tab input channels, returning `Stdin` if `stdin` is ready for reading or `Tab t` if the input channel of tab `t` is ready. `iselect` is implemented in Coq using a `select` primitive which is ultimately just a thin wrapper over the Unix select system call. The Coq extraction process, which converts Coq into OCaml for execution, can be customized to link our Coq code with OCaml implementations of primitives like `select`. Thus `select` is exposed to Coq essentially as a primitive of the appropriate monadic type. We have similar primitives for reading/writing on channels, and opening sockets.

Request from User If `stdin` is ready for reading, the kernel reads one character `c` using the `read` primitive, and then responds based on the value of `c`. If `c` is `F12`, a keycode for function key F12, the kernel adds a new tab to the browser. To achieve this, it first calls `mktab` to start a tab process (another primitive implemented in OCaml). `mktab` returns a tab object, which contains an input and output channels to communicate with the tab process. Once the tab `t` is created, the kernel sends it a `Render` message using the `write_msg` function – this tells `t` to render itself, which will later cause the tab to send a `Display` message to the kernel. Finally, we return an updated kernel state `(t, t::tabs)`, which sets the newly created tab `t` as the current tab, and adds `t` to the list of tabs.

In addition to `F12` the kernel handles several other cases for user input, which we omit in Figure 4.3. For example, when the kernel reads keys F1 through F10, it switches to tabs 1 through 10, respectively, if the tab exists. To switch tabs, the kernel updates the

currently selected tab and sends it a `Render` message. The kernel also processes mouse events delivered by the input process to the kernel's `stdin`. For now, we only handle mouse clicks, which are delivered by the input process using a single un-printable ASCII character (adding richer mouse events would not fundamentally change our kernel or proofs). The kernel in this case calls a primitive implemented in OCaml which gets the location of the mouse, and it sends a `MouseClick` message using the returned coordinates to the currently selected tab. We use this two-step approach for mouse clicks (un-printable character from the input process, followed by primitive in OCaml), so that the kernel only needs to processes a single character at a time from `stdin`, which simplifies the kernel and proofs.

Request from Tab If a tab `t` is ready for reading, the kernel reads a message `m` from the tab using `read_msg`, and then sends a response which depends on the message. If the message is `GetSoc(host, port)`, then the tab is requesting that a socket be opened to the given host/port. We apply the socket policy described in Section 4.2.4, where `domain_suffix t` returns the domain suffix of a tab `t`, and `safe_soc(host, domsuf)` applies the policy (which basically checks that `host` is a sub-domain of `domsuf`). If the policy allows the socket to be opened, the kernel uses the `send_socket` to open a socket to the host, and sends the socket over the channel to the tab (recall that we use Unix domain sockets to send open file descriptors from one process to another). Otherwise, it returns an `Error` message.

In addition to `GetSoc` the kernel handles several other cases for tab requests, which we omit in Figure 4.3. For example, the kernel responds to `GetURL` by retrieving a URL and returning the result. It responds to cookie store and retrieve messages by checking the security policy from Section 4.2.5 and forwarding the message to the appropriate cookie process (note that for simplicity, we did not show the cookie processes in Figure 4.3). The kernel also responds to cookie processes that are sending cookie

results back to a tab, by forwarding the cookie results to the appropriate tab. The kernel responds to `Display` messages by forwarding them to the output process.

Monads in Ynot The code in Figure 4.3 shows how Ynot supports an imperative programming style in Coq. This is achieved via *monads* which allow one to encode effectful, non-terminating computations in pure languages like Haskell or Coq. Here we briefly show how monads enable this encoding. In the next section we extend our discussion to show how Ynot's monads also enable reasoning about the kernel using pre- and post-conditions as in Hoare logic.

We use Ynot's `ST` monad which is a parameterized type where `ST T` denotes computations which may perform some I/O and then return a value of type `T`. To use `ST`, Ynot provides a `bind` primitive which has the following dependent type:

```
bind : forall T1 T2,
  ST T1 -> (T1 -> ST T2) -> ST T2
```

This type indicates that, for any types `T1` and `T2`, `bind` will take two parameters: (1) a monad of type `ST T1` and (2) a function that takes a value of type `T1` and returns a monad of type `ST T2`; then `bind` will produce a value in the `ST T2` monad. The type parameters `T1` and `T2` are inferred automatically by Coq. Thus, the expression `bind X Y` returns a monad which represents the computation: run `X` to get a value `v`; run `(Y v)` to get a value `v'`; return `v'`.

To make using `bind` more convenient, Ynot also defines Haskell-style "do" syntactic sugar using Coq's Notation mechanism, so that `x <- a;b` is translated to `bind a (fun x => b)`, and `a;b` is translated to `bind a (fun _ => b)`. Finally, the Ynot library provides a `return` primitive of type `forall T (v: T), ST T` (where again `T` is inferred by Coq). Given a value `v`, the monad `return v` represents the computation that does no I/O and simply returns `v`.

4.4 Kernel Verification

In this section we explain how we verified that QUARK's kernel implementation actually satisfies our security properties. First, we specify correct behavior of the kernel in terms of *traces*. Second, we prove the kernel satisfies this specification using the full power of Ynot's monads. Finally, we prove that our kernel specification implies our target security properties.

4.4.1 Actions and Traces

We verify our kernel by reasoning about the sequences of calls to primitives (*i.e.,* system calls) it can make. We call such a sequence a *trace*; our kernel specification (henceforth "spec") defines which traces are allowed for a correct implementation as in [67].

We use a list of *actions* to represent the trace the kernel produces by calling primitives. Each action in a trace corresponds to the kernel invoking a particular primitive. Figure 4.4 shows a partial definition of the `Action` datatype. For example: `ReadN f n l` is an `Action` indicating that the `n` bytes in list `l` were read from input channel `f`; `MkTab t` indicates that tab `t` was created; `SentSoc t host port` indicates a socket was connected to `host`/`port` and passed to tab `t`.

Although traces and actions are part of our spec, we can manipulate them like any other values in Coq. For example, we can define a function `Read c b` defined in terms of `ReadN` to encode the special case that a single byte `b` was read on input channel `c`. Though not shown here, we also define similar helper functions to build up trace fragments which correspond to having read or written a particular message to a given component. For example, `ReadMsg t (GetSoc host port)` corresponds to the trace fragment that results from reading a `GetSoc` request from tab `t`.

```
Definition Trace := list Action.

Inductive Action :=
  | ReadN   : chan -> positive -> list ascii -> Action
  | WriteN  : chan -> positive -> list ascii -> Action
  | MkTab   : tab -> Action
  | SentSoc : tab -> list ascii -> list ascii -> Action
  | ...

Definition Read c b := ReadN c 1 [c].
```

Figure 4.4. Traces and Actions. This Coq code defines the type of externally visible actions our kernel can take. A *trace* is simply a list of such actions. We reason about our kernel by proving properties of the traces it can have. Traces are like other Coq values; in particular, we can write functions that return traces. `Read` is a helper function to construct a trace fragment corresponding to reading a single byte.

4.4.2 Kernel Specification

Figure 4.5 shows a simplified snippet of our kernel spec. The spec is an inductively defined predicate `tcorrect` over traces with two constructors, stating the two ways in which `tcorrect` can be established: (1) `tcorrect_nil` states that the empty trace satisfies `tcorrect` (2) `tcorrect_step` states that if `tr` satisfies `tcorrect` and the kernel takes a single step, meaning that after `tr` it gets a request `req`, and responds with `rsp`, then the trace `rsp ++ req ++ tr` (where ++ is list concatenation) also satisfies `tcorrect`. By convention the first action in a trace is the most recent.

The predicate `step_correct` defines correctness for a single iteration of the kernel's main loop: `step_correct tr req rsp` holds if given the past trace `tr` and a request `req`, the response of the kernel should be `rsp`. The predicate has several constructors (not all shown) enumerating the ways `step_correct` can be established. For example, `step_correct_add_tab` states that typing `F12` on `stdin` leads to the creation of a tab and sending the `Render` message. The `step_correct_socket_true` case captures the successful socket creation case, whereas `step_correct_socket_false` captures the error case.

```
Inductive tcorrect : Trace -> Prop :=
  | tcorrect_nil:
      tcorrect nil
  | tcorrect_step: forall tr req rsp,
      tcorrect tr ->
      step_correct tr req rsp ->
      tcorrect (rsp ++ req ++ tr).

Inductive step_correct :
  Trace -> Trace -> Trace -> Prop :=
  | step_correct_add_tab: forall tr t,
      step_correct tr
        (MkTab t :: Read stdin F12 :: nil)
        (WroteMsg t Render)
  | step_correct_socket_true: forall tr t host port,
      is_safe_soc host (domain_suffix t) = true ->
      step_correct tr
        (ReadMsg t (GetSoc host port))
        (SentSoc t host port)
  | step_correct_socket_false: forall tr t host port,
      is_safe_soc host (domain_suffix t) <> true ->
      step_correct tr
        (ReadMsg t (GetSoc host port) ++ tr)
        (WroteMsg t Error ++ tr)
  | ...
```

Figure 4.5. Kernel Specification. `step_correct` is a predicate over triples containing a past trace, a request trace, and a response trace; it holds when the response is valid for the given request in the context of the past trace. `tcorrect` defines a correct trace for our kernel to be a sequence of correct steps, *i.e.,* the concatenation of valid request and response trace fragments.

```
Axiom readn:
 forall (f: chan) (n: positive) {tr: Trace},
 ST (list ascii)
    {traced tr * open f}
    {fun l =>
      traced (ReadN f n l :: tr) *
      [len l = n] * open f }.

Definition read_msg:
 forall (t: tab) {tr: Trace},
 ST msg
    {traced tr * open (tchan t)}
    {fun m =>
      traced (ReadMsg t m ++ tr) * open (tchan t)} :=
  ...
```

Figure 4.6. Example Monadic Types. This Coq code shows the monadic types for the `readn` primitive and for the `read_msg` function which is implemented in terms of `readn`. In both cases, the first expression between curly braces represents a pre-condition and the second represents a post-condition. The asterisk (`*`) may be read as normal conjunction in this context.

4.4.3 Monads in Ynot Revisited

In the previous section, we explained Ynot's `ST` monad as being parameterized over a single type `T`. In reality, `ST` takes two additional parameters representing pre- and post-conditions for the computation encoded by the monad. Thus, `ST T P Q` represents a computation which, if started in a state where `P` holds, may perform some I/O and then return a value of type `T` in a state where `Q` holds. For technical reasons, these pre- and post-conditions are expressed using separation logic, but we defer details to a tech report [54].

Following the approach of Malecha et al. [67], we define an *opaque* predicate `(traced tr)` to represent the fact that at a given point during execution, `tr` captures all the past activities; and `(open f)` to represent the fact that channel `f` is currently open. An opaque predicate cannot be proven directly. This property allows us to ensure that no part of the kernel can forge a proof of `(traced tr)` for any trace it independently constructs. Thus `(traced tr)` can only be true for the current trace `tr`.

Figure 4.6 shows the full monadic type for the `readn` primitive, which reads `n` bytes of data and returns it. The `*` connective represents the separating conjunction from separation logic. For our purposes, consider it as a regular conjunction. The precondition of `(readn f n tr)` states that `tr` is the current trace and that `f` is open. The post-condition states that the trace after `readn` will be the same as the original, but with an additional `(ReadN f n l)` action at the beginning, where the length of `l` is equal to `n` (`len l = n` is a regular predicate, which is lifted using square brackets into a separation logic predicate). After the call, the channel `f` is still open.

The full type of the Ynot `bind` operation makes sure that when two monads are sequenced, the post-condition of the first monad implies the pre-condition of the second. This is achieved by having `bind` take an additional third argument, which is a proof of this implication. The syntactic sugar for `x <- a;b` is updated to pass the wildcard "`_`" for the additional argument. When processing the definition of our kernel, Coq will enter into an interactive mode that allows the user to construct proofs to fill in these wildcards. This allows us to prove that the post-condition of each monad implies the pre-condition of the immediately following monad in Coq's interactive proof environment.

4.4.4 Back to the Kernel

We now return to our kernel from Figure 4.3 and show how we prove that it satisfies the spec from Figure 4.5. We augment the kernel state to additionally include the trace of the kernel so far, and we update our kernel code to maintain this `tr` field. By using a special encoding in Ynot for this trace, the `tr` field is not realized at run-time, it is only used for proof purposes.

We define the `kcorrect` predicate as follows (`s.tr` projects the current trace out of kernel state `s`):

```
Definition kcorrect (s: kstate) := traced s.tr * [tcorrect s.tr]
```

Now we want to show that `kcorrect` is an invariant that holds throughout execution of the kernel. Essentially we must show that `(kcorrect s)` is a loop invariant on the kernel state `s` for the main kernel loop, which boils down to showing that `(kcorrect s)` is valid as both the pre- and post-condition for the loop body, `kstep` as shown in Figure 4.3.

As mentioned previously, Coq will ask us to prove implications between the post-condition of one monad and the pre-condition of the next. While these proofs are ultimately spelled out in full formal detail, Coq provides facilities to automate a substantial portion of the proof process. Ynot further provides a handful of sophisticated tactics which helped automatically dispatch tedious, repeatedly occurring proof obligations. We had to manually prove the cases which were not handled automatically. While we have only shown the key kernel invariant here, in the full implementation there are many additional Hoare predicates for the intermediate goals between program points. We defer details of these predicates and the manual proof process to [54], but discuss proof effort in Section 3.6.

4.4.5 Security Properties

Our security properties are phrased as theorems about the spec. We now prove that our spec implies these key security properties, which we intend to hold in QUARK. Figure 4.7 shows these key theorems, which correspond precisely to the security properties outlined in Section 4.2.6.

State Integrity The first security property, `kstate_dep_user`, ensures that the kernel state only changes in response to the user pressing a "control key" (e.g. switching to the third tab by pressing F3). The theorem establishes this property by showing its contrapositive: if the kernel steps by responding with `rsp` to request `req` after trace `tr` *and* no "control keys" were read from the user, then the kernel state remains unchanged by this step. The function `proj_user_control`, not shown here, simply projects from

```
Theorem kstate_dep_user: forall tr req rsp,
  step_correct tr req rsp ->
  proj_user_control tr = proj_user_control (rsp ++ req ++ tr) ->
  kernel_state tr = kernel_state (rsp ++ req ++ tr).

Theorem kresponse_dep_kstate: forall tr1 tr2 req rsp,
  kernel_state tr1 = kernel_state tr2 ->
  step_correct tr1 req rsp ->
  step_correct tr2 req rsp.

Theorem tab_NI: forall tr1 tr2 t req rsp1 rsp2,
  tcorrect tr1 -> tcorrect tr2 ->
  from_tab t req ->
  (cur_tab tr1 = Some t <-> cur_tab tr2 = Some t) ->
  step_correct tr1 req rsp1 ->
  step_correct tr2 req rsp2 ->
  rsp1 = rsp2 \/
  (exists m, rsp1 = WroteCMsg (cproc_for t tr1) m /\
             rsp2 = WroteCMsg (cproc_for t tr2) m).

Theorem no_xdom_sockets: forall tr t,
  tcorrect tr -> In (SendSocket t host s) tr ->
  is_safe_soc host (domain_suffic t).

Theorem no_xdom_cookie_set: forall tr1 tr2 cproc,
  tcorrect (tr1 ++ SetCookie key value cproc :: tr2) ->
     exists tr t,
       (tr2 = (SetCookieRequest t key value :: tr) /\
        is_safe_cook (domain cproc) (domain_suffix t))

Theorem dom_bar_correct: forall tr,
  tcorrect tr -> dom_bar tr = domain_suffix (cur_tab tr).
```

Figure 4.7. Kernel Security Properties. This Coq code shows how traces allow us to formalize QUARK's security properties.

the trace all actions of the form `(Read c stdin)` where `c` is a control key. The function `kernel_state`, not shown here, just computes the kernel state from a trace. We also prove that at the beginning of any invocation to `kloop` in Figure 4.3, all fields of `s` aside from `tr` are equal to the corresponding field in `(kernel_state s.tr)`.

Response Integrity The second security property, `kresponse_dep_kstate`, ensures that every kernel response depends solely on the request and the kernel state. This delineates which parts of a trace can affect the kernel's behavior: for a given request `req`, the kernel will produce the same response `rsp`, for any two traces that *induce the same*

kernel state, even if the two traces have completely different sets of requests/responses (recall that the kernel state only includes the current tab and the set of tabs, and most request responses don't change these). Since the kernel state depends only the user's control key inputs, this theorem immediately establishes the fact that *our browser will never allow one component to influence how the kernel treats another component unless the user intervenes*.

Note that `kresponse_dep_kstate` shows that the kernel will produce the same response given the same request after any two traces that induce the same kernel state. This may seem surprising since many of the kernel's operations produce nondeterministic results, *e.g.,* there is no way to guarantee that two web fetches of the same URL will produce the same document. However, such nondeterminism is captured in the request, which is consistent with our notion of requests as inputs and responses as outputs.

Tab Non-Interference The second security property, `tab_NI`, states that the kernel's response to a tab is not affected by any other tab. In particular, `tab_NI` shows that if in the context of a valid trace, `tr1`, the kernel responds to a request `req` from tab `t` with `rsp1`, then the kernel will respond to the same request `req` with an equivalent response in the context of any other valid trace `tr2` which also contains tab `t`, irrespective of what other tabs are present in `tr2` or what actions they take. Note that this property holds in particular for the case where trace `tr2` contains *only* tab `t`, which leads to the following corollary: the kernel's response to a tab will be the same even if all other tabs did not exist.

The formal statement of the theorem in Figure 4.7 is made slightly more complicated because of two issues. First, we must assume that the focused tab at the end of `tr1` (denoted by `cur_tab tr1`) is `t` if and only if the focused tab at the end of `tr2` is also `t`. This additional assumption is needed because the kernel responds differently based on whether a tab is focused or not. For example, when the kernel receives a `Display`

message from a tab (indicating that the tab wants to display its rendered page to the user), the kernel only forwards the message to the output process if the tab is currently focused.

The second complication is that the communication channel underlying the cookie process for `t`'s domain may not be the same between `tr1` and `tr2`. Thus, in the case that kernel responds by forwarding a valid request from `t` to its cookie process, we guarantee that the kernel sends the same payload to the cookie process corresponding to `t`'s domain.

Note that, unlike `kresponse_dep_kstate`, `tab_NI` does not require `tr1` and `tr2` to induce the same kernel state. Instead, it merely requires the request `req` to be from a tab `t`, and `tr1` and `tr2` to be valid traces that both contain `t` (indeed, `t` must be on both traces otherwise the `step_correct` assumptions would not hold). Other than these restrictions, `tr1` and `tr2` may be arbitrarily different. They could contain different tabs from different domains, have different tabs focused, different cookie processes, etc.

Response Integrity and Tab Non-Interference provide different, complimentary guarantees. Response Integrity ensures the response to *any* request `req` is only affected by control keys and `req`, while Tab Non-Interference guarantees that the response to a tab request does not *directly* leak information to another tab. Note that Response Integrity could still hold for a kernel which mistakenly sends responses to the wrong tab, but Tab Non-Interference prevents this. Similarly, Tab Non-Interference could hold for a kernel which allows a tab to affect how the kernel responds to a cookie process, but Response Integrity precludes such behavior.

It is also important to understand that `tab_NI` proves the absence of interference as caused by the *kernel*, not by other components, such as the network or cookie processes. In particular, it is still possible for two websites to communicate with each other through the network, causing one tab to affect another tab's view of the web. Similarly, it is possible for one tab to set a cookie which is read by another tab, which again causes a tab to affect another one. For the cookie case, however, we have a separate theorem

about cookie integrity and confidentiality which states that cookie access control is done correctly.

Note that this property is an adaptation of the traditional non-interference property. In traditional non-interference, the program has "high" and "low" inputs and outputs; a program is non-interfering if high inputs never affect low outputs. Intuitively, this constrains the program to never reveal secret information to untrusted principles.

We found that this traditional approach to non-interference fits poorly with our trace-based verification approach. In particular, because the browser is a non-terminating, reactive program, the "inputs" and "outputs" are infinite streams of data.

Previous research [17] has adapted the notion of non-interference to the setting of reactive programs like browsers. They provide a formal definition of non-interference in terms of possibly infinite input and output streams. A program at a particular state is non-interfering if it produces *similar* outputs from *similar* inputs. The notion of similarity is parameterized in their definition; they explore several options and examine the consequences of each definition for similarity.

Our tab non-interference theorem can be viewed in terms of the definition from [17], where requests are “inputs” and responses are “outputs”; essentially, our theorem shows the inductive case for potentially infinite streams. Adapting our definition to fit directly in the framework from [17] is complicated by the fact that we deal with a unified trace of input and output events in the sequence they occur instead of having one trace of input events and a separate trace of output events. In future work, we hope to refine our notion of non-interference to be between domains instead of tabs, and we believe that applying the formalism from [17] will be useful in achieving this goal. Unlike [17], we prove a version of non-interference for a particular program, the QUARK browser kernel, directly in Coq.

Same Domain Suffix Socket Creation The third security property, `no_xdom_sockets`,

ensures that the kernel never delivers a socket bound to domain d to a tab whose domain does not match d. This involves checking URL suffixes in a style very similar to the cookie policy as discussed earlier. This property forces a tab to use `GetURL` when accessing websites that do not match its domain suffix, thus restricting the tab to only access publicly available data from other domains.

Same Domain Suffix Cookie Access The fourth security property states cookie integrity and confidentiality. As an example of how cookies are processed, consider the following trace when a cookie is set:

```
SetCookie key value cproc ::
SetCookieRequest tab key value :: ...
```

First, the `SetCookieRequest` action occurs, stating that a given `tab` just requested a cookie (in fact, `SetCookieRequest` is just defined in terms of a `ReadMsg` action of the appropriate message). The kernel responds with a `SetCookie` action (defined in terms of `WroteMsg`), which represents the fact that the kernel sent the cookie to the cookie process `cproc`. The kernel implementation is meant to find a `cproc` whose domain suffix corresponds to the `tab`. This requirement is given in the theorem `no_xdom_cookie_set`, which encodes cookie integrity. It requires that, within a correct trace, if a cookie process is ever asked to set a cookie, then it is in immediate response to a cookie set request for the same exact cookie from a tab whose domain matches that of the cookie process. There is a similar theorem `no_xdom_cookie_get`, not shown here, which encodes cookie confidentiality.

Domain Bar Integrity and Correctness The fifth property states that the domain bar is equal to the domain suffix of the currently selected tab, which encodes the correctness of the address bar.

Table 4.1. QUARK Components by Language and Size.

Component	Language	Lines of code
Kernel Code	Coq	859
Kernel Security Properties	Coq	142
Kernel Proofs	Coq	4,383
Kernel Primitive Specification	Coq	143
Kernel Primitives	Ocaml/C	538
Tab Process	Python	229
Input Process	Python	60
Output Process	Python	83
Cookie Process	Python	135
Python Message Lib	Python	334
WebKit Modifications	C	250
WebKit	C/C++	969,109

4.5 Evaluation

In this section we evaluate QUARK in terms of proof effort, trusted computing base, performance, and security.

Proof Effort and Component Sizes QUARK comprises several components written in various languages; we summarize their sizes in Table 4.1. All Python components share the "Python Message Lib" for messaging with the kernel. Implementing QUARK took about 6 person months, which includes several iterations redesigning the kernel, proofs, and interfaces between components. Formal shim verification saved substantial effort: we guaranteed our security properties for a million lines of code by reasoning just 859.

Trusted Computing Base The trusted computing base (TCB) consists of all system components we assume to be correct. A bug in the TCB could invalidate our security guarantees. QUARK's TCB includes:

- Coq's core calculus and type checker
- Our formal statement of the security properties
- Several primitives used in Ynot
- Several primitives unique to QUARK

- The OCaml compiler and runtime
- The underlying Operating System kernel
- Our chroot sandbox

Because Coq exploits the Curry-Howard Isomorphism, its type checker is actually the "proof checker" we have mentioned throughout the paper. We assume that our formal statement of the security properties correctly reflects how we understand them intuitively. We also assume that the primitives from Ynot and those we added in QUARK correctly implement the monadic type they are axiomatically assigned. We trust the OCaml compiler and runtime since our kernel is extracted from Coq and run as an OCaml program. We also trust the operating system kernel and our traditional chroot sandbox to provide process isolation, specifically, our design assumes the sandboxing mechanism restricts tabs to only access resources provided by the kernel, thus preventing compromised tabs from commuting over covert channels.

Our TCB does *not* include WebKit's large code base or the Python implementation. This is because a compromised tab or cookie process can not affect the security guarantees provided by kernel. Furthermore, the TCB does not include the browser kernel code, since it has been proved correct.

Ideally, QUARK will take advantage of previous formally verified infrastructure to minimize its TCB. For example, by running QUARK in seL4 [60], compiling QUARK's ML-like browser kernel with the MLCompCert compiler [1], and sandboxing other QUARK components with RockSalt [73], we could drastically reduce our TCB by eliminating its largest components. In this light, our work shows how to build yet another piece of the puzzle (namely a verified browser) needed to for a fully verified software stack. However, these other verified building blocks are themselves research prototypes which, for now, makes them very difficult to stitch together as a foundation for a realistic

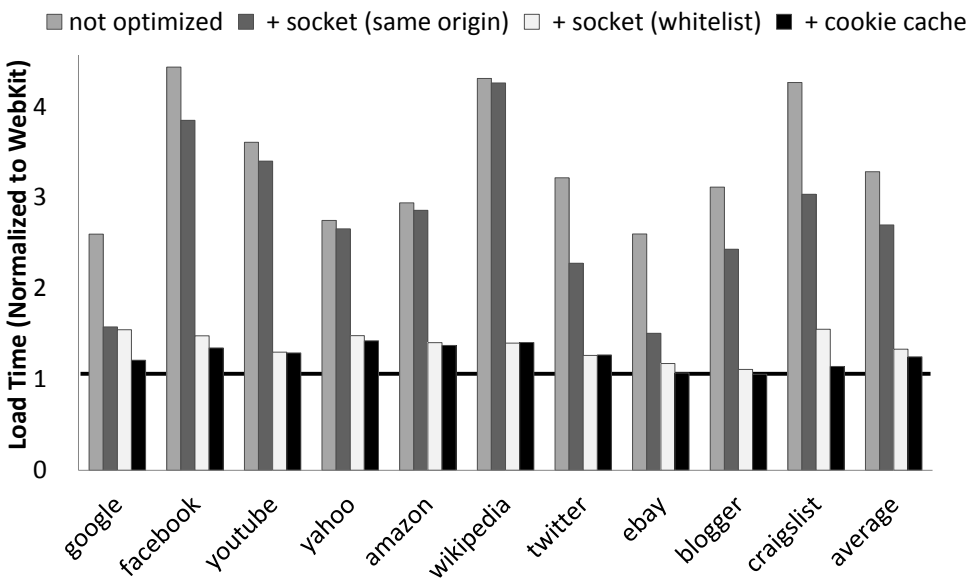


Figure 4.8. QUARK Performance. This graph shows QUARK load times for the Alexa Top 10 Websites, normalized to stock WebKit's load times. In each group, the leftmost bar shows the unoptimized load time, the rightmost bar shows the load time in the final, optimized version of QUARK, and intermediate bars show how additional optimizations improve performance. Smaller is better.

browser.

Performance We evaluate our approach's performance impact by comparing QUARK's load times to stock WebKit. Figure 4.8 shows QUARK load times for the top 10 Alexa Web sites, normalized to stock WebKit. QUARK's overhead is due to factoring the browser into distinct components which run in separate processes and explicitly communicate through a formally verified browser kernel.

By performing a few simple optimizations, the final version of QUARK loads large, sophisticated websites with only 24% overhead. This is a substantial improvement over a naïve implementation of our architecture, shown by the left-most "not-optimized" bars in Figure 4.8. Passing bound sockets to tabs, whitelisting content distribution networks for major websites, and caching cookie accesses, improves performance by 62% on average.

The WebKit baseline in Figure 4.8 is a full-featured browser based on the Python

bindings to WebKit. These bindings are simply a thin layer around WebKit's C/C++ implementation which provide easy access to key callbacks. We measure 10 loads of each page and take the average. Over all 10 sites, the average slowdown in load-time is 24% (with a minimum of 5% for `blogger` and a maximum of 42% for `yahoo`).

We also measured load-time for the previous version of QUARK, just before rectangle-based rendering was added. In this previous version, the average load-time was only 12% versus 24% for the current version. The increase in overhead is due to additional communication with the kernel during incremental rendering. Despite this additional overhead in load time, incremental rendering is preferable because it allows QUARK to display content to the user as it becomes available instead of waiting until an entire page is loaded.

Security Analysis QUARK provides strong, formal guarantees for security policies which are not fully compatible with traditional web security policies, but still provide some of the assurances popular web browsers seek to provide.

For the policies we have not formally verified, QUARK offers exactly the same level of traditional, unverified enforcement WebKit provides. Thus, QUARK actually provides security far beyond the handful policies we formally verified. Below we discuss the gap between the subset of policies we verified and the full set of common browser security policies.

The *same origin policy* [90] (SOP) dictates which resources a tab may access. For example, a tab is allowed to load cross-domain images using an `img` tag, but not using an `XMLHttpRequest`.

Unfortunately, we cannot easily verify this policy since restricting how a resource may be used after it has been loaded (*e.g.,* in an `img` tag vs. as a JavaScript value) requires reasoning across abstraction boundaries, *i.e.,* analyzing the large, complex tab implementation instead of treating it as a black box.

The SOP also restricts how JavaScript running in different frames on the same page may access the DOM. We could formally reason about this aspect of the SOP by making frames the basic protection domains in QUARK instead of tabs. To support this refined architecture, frames would own a rectangle of screen real estate which they could subdivide and delegate to sub-frames. Communication between frames would be coordinated by the kernel, which would allow us to formally guarantee that all frame access to the DOM conforms with the SOP.

We only formally prove inter-domain cookie isolation. Even this coarse guarantee prohibits a broad class of attacks, *e.g.,* it protects all Google cookies from any non-Google tab. QUARK does enforce restrictions on cookie access between subdomains; it just does so using WebKit as unverified cookie handling code within our cookie processes. Formally proving finer-grained cookie policies in Coq would be possible and would not require significant changes to the kernel or proofs.

Unfortunately, Quark does not prevent all cookie exfiltration attacks. If a subframe is able to exploit the entire tab, then it could steal the cookies of its top-level parent tab, and leak the stolen cookies by encoding the information within the URL parameter of `GetURL` requests. This limitation arises because tabs are principles in Quark instead of frames. This problem can be prevented by refining Quark so that frames themselves are the principles.

Our socket security policy prevents an important subset of cross-site request forgery attacks [14]. Quark guarantees that a tab uses a `GetURL` message when requesting a resource from sites whose domain suffix doesn't match with the tab's one. Because our implementation of `GetURL` does not send cookies, the resources requested by a `GetURL` message are guaranteed to be publicly available ones which do not trigger any privileged actions on the server side. This guarantee prohibits a large class of attacks, *e.g.,* cross-site request forgery attacks against Amazon domains from non-Amazon domains. However,

this policy cannot prevent cross-site request forgery attacks against sites sharing the same domain suffix with the tab, *e.g.,* attacks from a tab on www.ucsd.edu against cse.ucsd.edu since the tab on www.ucsd.edu can directly connect to cse.ucsd.edu using a socket and cookies on cse.ucsd.edu are also available to the tab.

Compatibility Issues QUARK enforces non-standard security policies which break compatibility with some web applications. For example, Mashups do not work properly because a tab can only access cookies for its domain and subdomains, *e.g.,* a subframe in a tab cannot properly access any page that needs user credentials identified by cookies if the subframe's domain suffix does not match with the tab's one. This limitation arises because tabs are the principles of Quark as opposed to subframes inside tabs. Unfortunately, tabs are too coarse grained to properly support mashups and retain our strong guarantees.

For the same reason as above, Quark cannot currently support third-party cookies. It is worth noting that third-party cookies have been considered a privacy-violating feature of the web, and there are even popular browser extensions to suppress them. However, many websites depend on third party cookies for full functionality, and our current Quark browser does not allow such cookies since they would violate our non-interference guarantees.

Finally, Quark does not support communications like "postMessage" between tabs; again, this would violate our tab non-interference guarantees.

Despite these incompatibilities, Quark works well on a variety of important sites such as Google Maps, Amazon, and Facebook since they mostly comply with Quarks' security policies. More importantly, our hope is that in the future Quark will provide a foundation to explore all of the above features within a formally verified setting.

In particular, adding the above features will require future work in two broad directions. First, frames need to become the principles in Quark instead of tabs. This change will require the kernel to support parent frames delegating resources like screen

region to child frames. Second, finer grained policies will be required to retain appropriate non-interference results in the face of these new features, e.g. to support interaction between tabs via "postMessage". Together, these changes would provide a form of "controlled" interference, where frames are allowed to communicate directly, but only in a sanctioned manner. Even more aggressively, one may attempt to re-implement other research prototypes like MashupOS [44] within Quark, going beyond the web standards of today, and prove properties of its implementation.

There are also several other features that Quark does not currently support, and would be useful to add, including local storage, file upload, browser cache, browser history, etc. However, we believe that these are not fundamental limitations of our approach or Quark's current design. Indeed, most of these features don't involve inter-tab communication. For the cases where they do (for example history information is passed between tabs if visited links are to be correctly rendered), one would again have to refine the non-interference definition and theorems to allow for controlled flow of information.

4.6 Discussion

In this section we discuss lessons learned while developing QUARK and verifying its kernel in Coq. In hindsight, these guidelines could have substantially eased our efforts. We hope they prove useful for future endeavors.

Formal Shim Verification Our most essential technique was *formal shim verification*. For us, it reduced the verification burden to proving a small browser kernel. Previous browsers like Chrome, OP, and Gazelle clearly demonstrate the value of kernel-based architectures. OP further shows how this approach enables reasoning about a model of the browser. We take the next step and formally prove the actual browser implementation correct.

Modularity through Trace-based Specification We ultimately specified the browser's correct behavior in terms of traces and proved both that (1) the implementation satisfies the spec and (2) the spec implies our security properties. Splitting our verification into these two phases improved modularity by separating concerns. The first proof phase reasons using monads in Ynot to show that the trace-based specification correctly abstracts the implementation. The second proof phase is no longer bound to reasoning in terms of monads – it only needs to reason about traces, substantially simplifying proofs.

This modularity aided us late in development when we proved address bar correctness (Theorem `dom_bar_correct` in Figure 4.7). To prove this theorem, we only had to reason about the trace-based specification, not the implementation. As a result, the proof of `dom_bar_correct` was only about 300 lines of code, tiny in comparison to the total proof effort. Thus, proving additional properties can be done with relatively little effort over the trace-based specification, without having to reason about monads or other implementation details.

Implement Non-verified Prototype First Another approach we found effective was to write a non-verified version of the kernel code before verifying it. This allowed us to carefully design and debug the interfaces between components and to enable the right browsing functionality before starting the verification task.

Iterative Development After failing to build and verify the browser in a single shot, we found that an iterative approach was much more effective. We started with a text-based browser, where the tab used `lynx` to generate a text-based version of QUARK. We then evolved this browser into a GUI-based version based on WebKit, but with no sockets or cookies. Then we added sockets and finally cookies. When combined with our philosophy of "write the non-verified version first", this meant that we kept a working version of the kernel written in Python throughout the various iterations. Just for comparison, the Python kernel which is equivalent to the Coq version listed in Figure 4.1 is 305 lines of

code.

Favor Ease of Reasoning When forced to choose between adding complexity to the browser kernel or to the untrusted tab implementation, it was *always* better keep the kernel as simple as possible. This helped manage the verification burden which was the ultimate bottleneck in developing QUARK. Similarly, when faced with a choice between flexibility/extensibility of code and ease of reasoning, we found it best to aim for ease of reasoning.

Acknowledgements

Chapter 4, in full, is a reprint of the material as it appears in Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium.* Dongseok Jang, Zachary Tatlock, and Sorin Lerner, USENIX Association, August 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Related Work

5.1 Securing JavaScript via Dynamic Information Flow Tracking

Information Flow. Information flow [32] and non-interference [39] have been used to formalize fine-grained isolation for nearly three decades. Several *static* techniques guarantee that certain kinds of inputs do no flow into certain outputs. These include type systems [108, 82], model checking [102], Hoare-logics [9], and dataflow analyses [61, 94]. Of these, the most expressive policies are captured by the dependent type system of [75], which allows the specification and (mostly) static enforcement of rich flow and access control policies including the dynamic creation of principals and declassification of high-security information. Unfortunately, fully static techniques are not applicable in the setting of JavaScript, as parts of the code only become available (for analysis) at run time, and as they often rely on the presence of underlying program structure (*e.g.,* a static type system).

Dynamic Taint Tracking. Several authors have investigated the use of dynamic taint propagation and checking, using specialized hardware [97, 105], virtual machines [21], binary rewriting [80], and source-level rewriting [22, 65]. In the late nineties, the JavaScript engine in Netscape 3.0 implemented a Data Tainting module [38], that tracked

a single taint bit on different pieces of data. The module was abandoned in favor of signed scripts (which today are rarely used in Web 2.0 applications), in part because it led to too many alerts. Our results show that, due to the prevalence of privacy-violating flows in popular Web 2.0 applications, the question of designing efficient, flexible and usable flow control mechanisms should be revisited. Recently, Vogt et al. [107] modified Firefox's JavaScript engine to track a taint bit that determines whether a piece of data is sensitive and report an XSS attack if this data is sent to a domain other than the page's domain, and Dhawan and Ganapathy [34] used similar techniques to analyze confidentiality properties of JavaScript browser extensions for Firefox. Our approach provides a different point in the design space. In particular, our policies are more expressive, in that our framework can handle both integrity and confidentiality policies, and more fine-grained, in that our framework can carry multiple taints from different sources at the same time, rather than just a single bit of taint. On the downside, our approach is implemented using a JavaScript rewriting strategy rather than modifying the JavaScript run-time, which results in a larger performance overhead. Dynamic rewriting approaches for client-side JavaScript information flow have also been investigated in a theoretical setting [22, 65]. Our work distinguishes itself from these more theoretical advances in terms of experimental evaluation: we have focused on implementing a rewriting-based approach that works on a large number of popular sites, and on evaluating the prevalence of privacy-violating flows on these websites.

JavaScript Security. One way to ensure safety on the *client* is to disallow unknown scripts from executing [57]. However, this will likely make it hard to use dynamic third-party content. Finally, Yu et al. [114] present a formal semantics of the interaction between JavaScript and browsers and builds on it a proxy-based rewriting framework for dynamically enforcing automata-based security policies [59]. These policies are quite different from information flow in that they require sparser instrumentation, and cannot

enforce fine-grained isolation.

History Sniffing. The possibility of history sniffing was first raised in the academic community a decade ago [37]. The original form of history sniffing used timing difference between retrieving a resource that is cached (because it has previously been retrieved) and one that is not. In general, many other forms of history sniffing are possible based on CSS link decoration, some of which (for example, setting the background property of a visited link to `url(...)`) work even when JavaScript is disabled. This, together with the genuine user-interface utility that visited link decoration provides, is the reason that history sniffing is so difficult to address comprehensively in browsers (cf. [50] for a proposed server-side solution, [49] for a proposed client-side solution and [13] for the fix recently deployed by the Firefox browser.) The potential of history sniffing has been recently proven to be enormous [51]. However, since to date there has been no public disclosure regarding the use of history sniffing, and no publicly available tools for detecting it, we expect that, today, many malicious sites will prefer the simple, robust approach of querying and exfiltrating link computed style. Accordingly, it is this data flow that we focus on; if there are sites that use other approaches, we will not have detected them. Our goal in this paper is to draw attention to the use of clandestine history sniffing at popular, high-traffic sites, which means that false negatives are acceptable.

5.2 Securing C++ Virtual Function Calls

The research community has developed numerous defenses to increase the cost of mounting low-level attacks that corrupt control data, steadily driving attackers to discover new classes of exploitable programming errors like vtable hijacking. In this section we survey the existing defenses most relevant to vtable hijacking, consider their effectiveness at mitigating such attacks, and compare them to SAFEDISPATCH.

Reference Counting. Reference counting [25, 89, 64] is a memory management

technique used in garbage collectors and complex applications to track how many references point to an object during program execution. When the number of references reaches zero, the object may safely be freed. Use-after-free errors can be avoided using reference counting by checking that an object has a non-zero number of references before calling any methods with the object. While this may help increase the attack complexity of vtable hijacking attacks mounted by exploiting use-after-free bugs, reference counting can have a non-trivial run-time overhead, and it also makes reclaiming cyclic data-structures complicated. Most importantly, however, reference counting cannot fundamentally prevent such attacks. In reference counting, the number of references to an object is stored in the heap, and thus an adversary capable of corrupting vtable pointers would also be able to corrupt reference counts, thereby circumventing any reference counting based defense. In contrast, SAFEDISPATCH instrumentation is placed in the program binary which resides in read-only memory and thus is not susceptible to corruption by an attacker.

Memory Safety. Programs written in *memory safe* languages are guaranteed, by construction, to be free of exploitable, low-level memory errors. This kind of memory safety guarantee is clearly stronger than the guarantee that SAFEDISPATCH provides. However, unfortunately programs written in such languages often suffer significant performance overhead from runtime checking to ensure that all memory operations are safe. This overhead is sufficient to preclude the use of memory safe languages in many performance critical applications. In contrast, SAFEDISPATCH provides strong security guarantees without any assumptions about memory safety and incurs only minimal overhead.

There has also been extensive research on C compilers which insert additional checks or modify language features to ensure memory safety, for example CCured [78, 27], Cyclone [56], Purify [112], and Deputy [26]. While these techniques can help

prevent vtable hijacking, they often require some amount of user annotations, and even if they don't, their run-time overheads are bigger than SAFEDISPATCH, especially on large-scale applications like Chrome.

Control Flow Integrity. Control flow integrity (CFI) is a technique that inserts sufficient checks in a program to ensure that every control flow transfer jumps to a valid program location [7]. Recent advances have greatly reduced the overhead of CFI, in some cases to as low as 5%, by adapting efficient checks for indirect targets [117], using static analysis [116], harnessing further compiler optimizations from a trustworthy high-level inline-reference monitor representation [115], or incorporating low-level optimizations [118]. The main difference between our work and these previous CFI approaches lies in the particular point in design space that we chose to explore. Broadly speaking, previous CFI approaches are designed to secure all indirect jumps whereas we focus specifically on protecting C++ dynamic dispatch, which has become a popular target for exploits. In this more specific setting, we provide stronger guarantees than recent CFI approaches while incurring very low performance overhead.

VTable Hijacking Prevention. The GCC compiler has recently been extended with a promising new "vtable verification" feature developed by Google [104, 103], concurrently and indenpendently from SAFEDISPATCH. The GCC approach compiles each C++ source file to an object file extended with local vtable checking data, and the local checking data is combined at load-time into a program-wide checking table. Each virtual method call site is then instrumented with a call to a checking function which uses the program-wide table to determine if the control-flow transfer should be allowed. In many respects, the GCC approach is roughly equivalent to our unoptimized vtable checking approach. In this light, our work extends GCC's approach in the following ways: (1) we explore and empirically evaluate not only vtable checking, but also method checking (2) through this evaluation, we discover and propose a new optimization

opportunity in the form of a hybrid approach and (3) we inline common checks. In our implementation, vtable checking without inlining (which is roughly what GCC does) leads to an overhead of about 25%. Through optimizations 2 and 3 above, we reduce the overhead to only 2%. On the other hand, the GCC approach supports separate compilation much more easily than our approach, which requires whole program analysis and profiling.

Another technique for preventing vtable hijacking is VTGuard [85], a feature of the Visual Studio C++ compiler. This approach inserts a secret cookie into each vtable and checks the cookie before the vtable is used at runtime. While this approach has very low performance overhead, it is less secure than ours: the attacker can still overwrite a vtable pointer to make it point to *any* vtable generated by the compiler, something we prevent. Moreover, if the secret cookie is revealed through an information disclosure attack, then the VTGuard protection mechanism can be circumvented.

Memory Allocators and Dynamic Heap Monitoring. Dynamic heap monitoring, like that used in Undangle [18] and Valgrind [79], can help discover memory errors during testing, but are not suitable for deployment as they can impose up to 25x performance overhead, which is unacceptable for the applications we aim to protect. The DieHard [16, 81] custom memory manager has proven effective at providing probabilistic guarantees against several classes of memory errors, including heap-based buffer overflows and use-after-free errors by randomizing and spreading out the heap. While DieHard overhead is often as low at 8%, it demands a heap *at least* 2x larger than what the protected application would normally require, which is unacceptable for the applications we aim to protect. Furthermore, large applications like a browser often use multiple custom memory allocators for performance, whereas DieHard requires the entire application to use a single allocator.

Data Execution Prevention (DEP). After an adversary has compromised pro-

gram control flow, they must arrange for their attack code to be executed. DEP [70] seeks to prevent an attacker from writing malicious shellcode directly to memory and then jumping to that code. Conceptually every memory page is *either* writable *or* executable, but *never* both. DEP can mitigate vtable hijacking after the attack has been mounted by preventing the attacker from executing code they've allocated somewhere in memory. However, attackers can still employ techniques like Return Oriented Programming [93] (ROP) to circumvent DEP after control flow has been compromised from a vtable hijacking attack. DEP is also often disabled for JIT. While DEP tries to mitigate the damage an attacker can do after compromising control flow, SAFEDISPATCH seeks to prevent a class of control flow compromises (those due to vtable hijacking) from arising in the first place.

Address Space Layout Randomization (ASLR). Like DEP, ASLR [100] seeks to severely limit an attackers ability to execute their attack code *after* control flow has been compromised. It does this by randomly laying out pages in memory so that program and library code will not reside at predictable addresses, making it difficult to mount ROP and other attacks. Unfortunately, for compatibility, many prevalent, complex applications are still forced to load key libraries at predictable addresses, limiting the effectiveness for ASLR in these applications. SAFEDISPATCH helps secure such applications by preventing vtable-hijacking-based control flow compromises from arising in the first place.

5.3 Formal Verification for Kernel-based Browsers

This section briefly discusses both previous efforts to improve browser security and verification techniques to ensure programs behave as specified.

Kernel-based Browser Architecture. As mentioned in the Introduction, there is a rich literature on techniques to improve browser security [15, 110, 42, 99, 69, 20, 19]. We distinguish ourselves from all previous techniques by verifying the actual implementation of a modern Web browser and formally proving that it satisfies our security properties, all in the context of a mechanical proof assistant. Below, we survey the most closely related work.

Previous browsers like Google Chrome [15], Gazelle [110], and OP [42] have been designed using *privilege separation* [83], where the browser is divided into components which are then limited to only those privileges they absolutely require, thus minimizing the damage an attacker can cause by exploiting any one component. We follow this design strategy.

Chrome's design compromises the principles of privilege separation for the sake of performance and compatibility. Unfortunately, its design does not protect the user's data from a compromised tab which is free to leak all cookies for every domain. Gazelle [110] adopts a more principled approach, implementing the browser as a multi-principal OS, where the kernel has exclusive control over resource management across various Web principals. This allows Gazelle to enforce richer policies than those found in Chrome. However, neither Chrome nor Gazelle apply any formal methods to make guarantees about their browser.

The OP [42] browser goes beyond privilege separation. Its authors additionally construct a model of their browser kernel and apply the Maude model checker to ensure that this model satisfies important security properties such as the same origin policy and address bar correctness. As such, the OP browser applies insight similar to our work, in that OP focuses its formal reasoning on a small kernel. However, unlike our work, OP does not make any formal guarantees about the actual browser implementation, which means there is still a formality gap between the model and the code that runs. Our formal

shim verification closes this formality gap by conducting all proofs in full formal detail using a proof assistant.

Formal Verification. Recently, researchers have begun using proof assistants to fully formally verify implementations for foundational software including Operating Systems [60], Compilers [63, 1], Database Management Systems [66], Web Servers [67], and Sandboxes [73]. Some of these results have even *experimentally* been shown to to drastically improve software reliability: Yang et al. [113] show through random testing that the CompCert verified C compiler is substantially more robust and reliable than its non-verified competitors like GCC and LLVM.

As researchers verify more of the software stack, the frontier is being pushed toward higher level platforms like the browser. Unfortunately, previous verification results have only been achieved at staggering cost; in the case of seL4, verification took over 13 person years of effort. Based on these results, verifying a browser-scale platform seemed truly infeasible.

Our formal verification of QUARK was radically cheaper than previous efforts. Previous efforts were tremendously expensive because researchers proved nearly every line of code correct. We avoid these costs in QUARK by applying *formal shim verification*: we structure our browser so that all our target security properties can be ensured by a very small browser kernel and then reason only about that single, tiny component. Leveraging this technique enabled us to make strong guarantees about the behavior of a million of lines of code while reasoning about only a few hundred in the mechanical proof assistant Coq.

We use the Ynot library [77] extensively to reason about imperative programming features, *e.g.,* impure functions like `fopen`, which are otherwise unavailable in Coq's pure implementation language. Ynot also provides features which allow us to verify QUARK

in a familiar style: invariants expressed as pre- and post-conditions over program states, essentially a variant of Hoare Type Theory [76]. Specifically, Ynot enables *trace-based verification*, used extensively in [67] to prove properties of servers. This technique entails reasoning about the sequence of externally visible actions a program may perform on any input, also known as *traces*. Essentially, our specification delineates which sequences of system calls the QUARK kernel can make and our verification consists of proving that the implementation is restricted to only making such sequences of system calls. We go on to formally prove that satisfying this specification implies higher level security properties like tab isolation, cookie integrity and confidentiality, and address bar integrity and correctness. Building QUARK with a different proof assistant like Isabelle/HOL would have required essentially the same approach for encoding imperative programming features, but we chose Coq since Ynot is available and has been well vetted.

Our approach is fundamentally different from previous verification tools like ESP [30], SLAM [12], BLAST [43] and Terminator [28], which work on existing code bases. In our approach, instead of trying to prove properties about a large existing code base expressed in difficult-to-reason-about languages like C or C++, we rewrite the browser inside of a theorem prover. This provides much stronger reasoning capabilities.

Chapter 6

Conclusions and Future Work

In this dissertation, we proposed techniques for retrofitting programming language runtimes for JavaScript and C++, and restructuring a browser with a browser kernel formally verified in Coq.

First, we proposed a rewriting-based information flow framework for JavaScript and evaluated the performance of an instantiation of the framework. Our evaluation showed that the performance of our rewriting-based information flow control is acceptable given our engineering and optimization efforts, but it still imposes a perceptible running-time overhead. We also presented an extensive empirical study of the prevalence of privacy-violation information flows: cookie stealing, location hijacking, history sniffing, and behavior tracking. Our JavaScript information flow framework found many interesting privacy-violating information flows including 46 cases of real history sniffing over the Alexa global top 50,000 websites, despite some incompleteness.

Second, we addressed the growing threat of vtable hijacking with SAFEDISPATCH, an enhanced C++ compiler to ensure that control flow transfers at method invocations via dynamic dispatch are valid at runtime according to the static C++ semantics. SAFEDISPATCH first performs class hierarchy analysis (CHA) to determine, for each class c in the program, the set of valid method implementations that may be invoked by an object of static type c, according to C++ semantics. SAFEDISPATCH then uses the information

produced by CHA to instrument the program with checks, ensuring that, at runtime, all method calls invoke a valid method implementation according to C++ dynamic dispatch rules. To minimize performance overhead, SAFEDISPATCH performs optimizations to inline and order checks based on profiling data and adopts a hybrid approach which combines method checking and vtable checking. We were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5% in the first vtable-safe version of the Google Chromium browser which we built with the SAFEDISPATCH compiler.

Third, we demonstrated how *formal shim verification* can be used to achieve strong security guarantees for a modern Web browser using a mechanical proof assistant. We formally proved that our browser provides tab noninterference, cookie integrity and confidentiality, and address bar integrity and correctness. We detailed our design and verification techniques and showed that the resulting browser, QUARK, provides a modern browsing experience with performance comparable to the default WebKit browser. For future research, QUARK furnishes a framework to easily experiment with additional web policies without re-engineering an entire browser or formalizing all the details of its behavior from scratch.

We believe that these results are solid first steps towards securing browsers by hardening each layer of browsers ranging from JavaScript into core security logic implementation of browsers. Our work provides a good foundation for future exploration as we explain them in the rest of the chapter.

Dynamic Information Flow for JavaScript. Recently, browsers have been competing with each other to achieve a better performance to accommodate the demanding workload of HTML5 JavaScript web applications. Thus the most important direction for future work is to optimize the performance of dynamic information tracking frameworks. Although our rewriting-based approach makes our flow policy language very flexible, information flow tracking logic must be deeply incorporated in JIT compilation that all

modern JavaScript have adopted for realistic performance. We may attempt to achieve a better performance by incorporating off-line static analysis of each JavaScript function. Each JavaScript function without `eval` could be statically and precisely analyzed and then a JIT-based JavaScript information flow framework can harness the result to apply a more aggressive compile-time optimization to emit more efficient code.

Another direction for future work is to explore flow policies capturing the malicious behavior of various JavaScript attacks in the wild. In many web applications, there are other sources of private information such as user credentials, social security numbers, and credit card numbers. Finding out many private information sources and securing them can be challenging due to complicated interactions between many websites.

Securing C++ Virtual Function Calls. The current prototype of SAFEDISPATCH does not support either separate compilation nor dynamic-link libraries. For future work, SAFEDISPATCH will be able to support separate compilation by emitting partial CHA information into for each compilation unit into an object file and a linker can gather them up at linking time later. This approach also allows to support dynamic-link libraries if we also incorporate the CHA information stored in them.

Another direction for future work is to perform dynamic profiling of virtual calls and *dynamically* apply the result for inlining optimization. Dynamic profiling will free developers from obtaining representative profiling data offline. However, dynamically applying inlining optimization will require the code section to be temporarily writable, which can be exploited by the attacker. One possible direction to resolve this security concern would be to adapt the secure self-modifying code technique introduced by the Google Native Client [10]. For each virtual call site, SAFEDISPATCH will emit a placeholder initially filled with checks that always fail, leading the execution to the general `check` function call. At runtime, `check` function call will gather profile information and fill out those checks with frequently used method pointers. When applying inlining

dynamically, `check` can first guard the placeholder with a halt instruction, and *atomically* update only inlined method pointers at a virtual callsite as proposed in Native Client.

Formally Verifying Browser Kernel. One important direction for future work is to refine the principles of QUARK to resolve the current compatibility issues and provide stronger security guarantees. For future work, frames must become the principles in QUARK instead of tabs. This change will support Mashups with multiple cross-domain subframes in one tab. The kernel will be required to support parent frames delegating resources like screen region to child frames. Finer grained policies will also be required to retain appropriate non-interference results in the face of these new features, e.g. to support interaction between tabs via "postMessage". Together, these changes would provide a form of "controlled" interference, where frames are allowed to communicate directly, but only in a sanctioned manner.

Even more aggressively, one may attempt to provide formal guarantees for the same origin policy. Enforcing SOP will require the QUARK kernel to control how cross-domain resources (*e.g.,* image resources) are used in an HTML renderer, and this seems quite challenging in terms of engineering efforts.

Another direction for future work is to support various features of the modern browsers for QUARK. These might include local storage, file upload, browser cache, browser history, etc. However, we believe that these are not fundamental limitations of our approach or QUARK's current design. Indeed, most of these features don't involve communications between tabs. For the cases where they do (*e.g.,* history information is passed between tabs if visited links are to be correctly rendered), one would again have to refine the non-interference definition and theorems to allow for controlled flow of information.

Bibliography

[1] http://gallium.inria.fr/~dargaye/mlcompcert.html.

[2] Chrome security hall of fame. http://dev.chromium.org/Home/chromium-security/hall-of-fame.

[3] Public suffix list. http://publicsuffix.org/.

[4] Pwn2own. http://en.wikipedia.org/wiki/Pwn2Own.

[5] Mitre. cve-2013-1701, 2006.

[6] Mitre. cve-2013-1713, 2006.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[8] Devdatta Akhawe, Adam Barth, Peifung E. Lamy, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In Michael Backes and Andrew Myers, editors, *Proceedings of CSF 2010*, pages 290–304. IEEE Computer Society, July 2010.

[9] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Proceedings of SAS 2004*, volume 3148 of *LNCS*, pages 100–15. Springer-Verlag, August 2004.

[10] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. *ACM SIGPLAN Notices*, 46(6):355–366, 2011.

[11] Apple. Sunspider 1.0 javascript benchmark suite. https://www.webkit.org/perf/sunspider/sunspider.html, 2013.

[12] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[13] L. David Baron. Preventing attacks on a user's history through CSS :visited selectors, April 2010. Online: http://dbaron.org/mozilla/visited-privacy.

[14] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security*, pages 75–88, 2008.

[15] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of the Chromium browser. Technical report, Google, 2008.

[16] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[17] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[18] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[19] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[20] Shuo Chen, José Meseguer, Ralf Sasse, Helen J. Wang, and Yi min Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, 2007.

[21] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In Matt Blaze, editor, *Proceedings of USENIX Security 2004*, pages 321–36. USENIX, August 2004.

[22] Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In Michael Backes and Andrew Myers, editors, *Proceedings of CSF 2010*. IEEE Computer Society, July 2010.

[23] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of PLDI 2009*, pages 50–62, 2009.

[24] A. Clover. Timing attacks on Web privacy. Online: http://www.securiteam.com/securityreviews/5GP020A6LG.html, February 2002.

[25] George E. Collins. A method for overlapping and erasure of lists. In *CACM*, 1960.

[26] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[27] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *PLDI*, 2003.

[28] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV*, 2006.

[29] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[30] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[31] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, 1995.

[32] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[33] David Dewey and Jonathon T. Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *NDSS*, 2012.

[34] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In Charlie Payne and Michael Franz, editors, *Proceedings of ACSAC 2009*, pages 382–91. IEEE Computer Society, December 2009.

[35] Chris Evans. Exploiting 64-bit Linux like a boss. http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html, 2013.

[36] Adrienne Felt. Defacing facebook: A security case study. *White paper, UC Berkeley*, 2007.

[37] Edward W. Felten and Michael A. Schneider. Timing attacks on Web privacy. In Sushil Jajodia, editor, *Proceedings of CCS 2000*, pages 25–32. ACM Press, November 2000.

[38] David Flannagan. *JavaScript: The Definitive Guide*. O'Reilly, fifth edition, August 2006.

[39] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of IEEE Security and Privacy ("Oakland") 1982*, pages 11–20. IEEE Computer Society, April 1982.

[40] Google. Heap-use-after-free in WebCore (exploitable). https://code.google.com/p/chromium/issues/detail?id=162835, 2012.

[41] Google. Octane javascript benchmark suite. https://developers.google.com/octane/, 2013.

[42] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *IEEE Security and Privacy*, 2008.

[43] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, 2002.

[44] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: operating system abstractions for client mashups. In *HotOS*, 2007.

[45] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, pages 619–629, 2010.

[46] WhiteHat Security Inc. Whitehat website security statistics report, May 2013. https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf.

[47] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *Proceedings of Web 2.0 & Privacty*, 2008.

[48] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *ACM Conference on Computer and Communications Security*, pages 421–431, 2007.

[49] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from Web privacy attacks. In Carole Goble and Mike Dahlin, editors, *Proceedings of WWW 2006*, pages 737–44. ACM Press, May 2006.

[50] Markus Jakobsson and Sid Stamm. Invasive browser sniffing and countermeasures. In Carole Goble and Mike Dahlin, editors, *Proceedings of WWW 2006*, pages 523–32. ACM Press, May 2006.

[51] Artur Janc and Lukasz Olejnik. Feasibility and real-world implications of Web browser history detection. In Collin Jackson, editor, *Proceedings of W2SP 2010*. IEEE Computer Society, May 2010.

[52] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of the ACM Conference Computer and Communications Security (CCS)*, 2010.

[53] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. Rewriting-based dynamic information flow for JavaScript. Technical report, University of California, San Diego, January 2010. Online: http://pho.ucsd.edu/rjhala/dif.pdf.

[54] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. Technical report, UC San Diego, 2012.

[55] Dongseok Jang, Aishwarya Venkataraman, G. Michael Sawka, and Hovav Shacham. Analyzing the cross-domain policies of flash applications. In *In Web 2.0 Security and Privacy (W2SP 2011)*, May 2011.

[56] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX ATEC*, 2002.

[57] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In Peter Patel-Schneider and Prashant Shenoy, editors, *Proceedings of WWW 2007*, pages 601–10. ACM Press, May 2007.

[58] Nils JÃijnemann. Cross-site-scripting in google mail, June 2012. http://www.nilsjuenemann.de/2012/06/cross-site-scripting-in-google-mail.html.

[59] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript instrumentation in practice. In G. Ramalingam, editor, *Proceedings of APLAS 2008*, volume 5356 of *LNCS*, pages 326–41. Springer-Verlag, December 2008.

[60] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[61] Monica S. Lam, Michael Martin, V. Benjamin Livshits, and John Whaley. Securing Web applications with static and dynamic information flow tracking. In Robert Glück and Oege de Moor, editors, *Proceedings of PEPM 2008*, pages 3–12. ACM Press, January 2008.

[62] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[63] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *PLDI*, 2006.

[64] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. In *TOPLAS*, 2006.

[65] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. In Kai Rannenberg and Vijay Varadharajan, editors, *Proceedings of SEC 2010*, September 2010.

[66] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.

[67] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46:95–118, February 2011.

[68] Leo A. Meyerovich and V. Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of IEEE Security and Privacy ("Oakland") 2010*, pages 481–496. IEEE Computer Society, 2010.

[69] James Mickens and Mohan Dhawan. Atlantis: robust, extensible execution environments for web applications. In *SOSP*, pages 217–231, 2011.

[70] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. http://support.microsoft.com/kb/875352.

[71] Microsoft. Vulnerability in Internet Explorer could allow remote code execution. http://technet.microsoft.com/en-us/security/advisory/961051, 2008.

[72] H. D. Moore. Microsoft Internet Explorer data binding memory corruption. http://packetstormsecurity.com/files/86162/Microsoft-Internet-Explorer-Data-Binding-Memory-Corruption.html, 2010.

[73] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger sfi for the x86. In *PLDI*, 2012.

[74] Mozilla. Kraken 1.1 javascript benchmark suite. http://krakenbenchmark.mozilla.org/, 2013.

[75] Andrew C. Myers. Programming with explicit security policies. In Mooly Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *LNCS*, pages 1–4. Springer-Verlag, April 2005.

[76] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.

[77] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

[78] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. In *TOPLAS*, 2005.

[79] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[80] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Dan Boneh and Dan Simon, editors, *Proceedings of NDSS 2005*. ISOC, February 2005.

[81] Gene Novark and Emery D. Berger. Dieharder: securing the heap. In *CCS*, 2010.

[82] François Pottier and Vincent Simonet. Information flow inference for ML. In John C. Mitchell, editor, *Proceedings of POPL 2002*, pages 319–330. ACM Press, January 2002.

[83] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003.

[84] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, et al. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, 2007.

[85] Matthew R. Miller and Kenneth D.Johnson. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities. http://patentimages.storage.googleapis.com/pdfs/US20120144480.pdf, 2010.

[86] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[87] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from google chrome. In *CACM*, 2009.

[88] rix. Smashing c++ vptrs. http://www.phrack.org/issues.html?issue=56&id=8, 2000.

[89] David J. Roth and David S. Wise. One-bit counts between unique and sticky. In *ISMM*, 1998.

[90] Jesse Ruderman. The same origin policy, 2001. http://www.mozilla.org/projects/security/ components/same-origin.html.

[91] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In Michael Backes and Peng Ning, editors, *Proceedings of ESORICS 2009*, volume 5789 of *LNCS*, pages 86–103. Springer-Verlag, September 2009.

[92] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[93] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, 2007.

[94] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In Dan Wallach, editor, *Proceedings of USENIX Security 2001*, pages 201–17. USENIX, August 2001.

[95] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy*, pages 463–478, 2010.

[96] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, 2010.

[97] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In Kathryn McKinley, editor, *Proceedings of ASPLOS 2004*, pages 85–96. ACM Press, October 2004.

[98] Symantec. Microsoft Internet Explorer virtual function table remote code execution vulnerability. http://www.symantec.com/security_response/vulnerability.jsp?bid=54951, 2012.

[99] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the illinois browser operating system. In *OSDI*, pages 17–32, 2010.

[100] PaX Team. Pax address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[101] Symantec Security Intel Analysis Team. Pwn2own 2010: Lessons learned. http://www.symantec.com/connect/blogs/pwn2own-2010-lessons-learned, 2010.

[102] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin, editor, *Proceedings of SAS 2005*, volume 3672 of *LNCS*, pages 352–67. Springer-Verlag, September 2005.

[103] Caroline Tice. Gcc vtable security hardening proposal. http://gcc.gnu.org/ml/gcc-patches/2012-11/txt00001.txt, 2012.

[104] Caroline Tice. Improving function pointer security for virtual method dispatches. http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf, 2012.

[105] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In Antonio González and John P. Shen, editors, *Proceedings of MICRO 2004*, pages 243–54. IEEE Computer Society, December 2004.

[106] Olli Vertanen. Java type confusion and fault attacks. In *FDTC*, 2006.

[107] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In William Arbaugh and Crispin Cowan, editors, *Proceedings of NDSS 2007*. ISOC, February 2007.

[108] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In Thomas Reps, editor, *Proceedings of POPL 2000*, pages 268–76. ACM Press, January 2000.

[109] VUPEN. Exploitation of Mozilla Firefox use-after-free vulnerability. http://www.vupen.com/blog/20120625.Advanced_Exploitation_of-Mozilla_Firefox_UaF_CVE-2012-0469.php, 2012.

[110] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the gazelle web browser. Technical Report MSR-TR-2009-16, MSR, 2009.

[111] WebKit. Rendering performance tests. https://code.google.com/p/webkit-mirror/source/browse/PerformanceTests/, 2013.

[112] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *SIGSOFT*, 2004.

[113] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[114] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In Matthias Felleisen, editor, *Proceedings of POPL 2007*, pages 237–49. ACM Press, January 2007.

[115] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato-a retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, 2013.

[116] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 29–40. ACM, 2011.

[117] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy, San Francisco, CA*, 2013.

[118] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, 2013.