FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ZigZag

Hardening Web Applications against CSV Attacks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Weissbacher
Matrikelnummer 0553406

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Engin Kirda
Privatdoz. Dipl.-Ing. Dr.techn. Christopher Kruegel

Wien, 18.08.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ZigZag

Hardening Web Applications against CSV Attacks

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Weissbacher
Registration Number 0553406

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Engin Kirda
Privatdoz. Dipl.-Ing. Dr.techn. Christopher Kruegel

Vienna, 18.08.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Weissbacher
284 Harvard St. Apt 76, 02139 Cambridge, MA, USA

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I want to thank my advisors Christopher Kruegel and Engin Kirda for their guidance and patience while carrying out this project. Special thanks go out to my family and friends for continued support, including the Security Groups at Northeastern University, UC Santa Barbara, and TU Vienna where I received plenty of valuable input and opportunity. I would also like to thank the Marshall Plan Foundation and the Office of Naval Research (ONR) for providing partial support.

Abstract

Modern web applications are increasingly moving program code to the client in the form of JavaScript. With the growing adoption of HTML5 APIs such as `postMessage`, client-side validation (CSV) vulnerabilities are consequently becoming increasingly important to address. However, while detecting and preventing attacks against web applications is a well-studied topic on the server, considerably less work has been performed for the client. Exacerbating this issue is the problem that defenses against CSVs must, in the general case, fundamentally exist in the browser, rendering current server-side defenses inadequate.

In this master's thesis, we present ZigZag, a system for hardening JavaScript-based web applications against client-side validation attacks. ZigZag transparently instruments client-side code to perform dynamic invariant detection on security-sensitive code, generating models that describe how – and with whom – client-side components interact. ZigZag is capable of handling templated JavaScript, avoiding full re-instrumentation when JavaScript programs are structurally similar. Learned invariants are then enforced through a subsequent instrumentation step. Our evaluation demonstrates that ZigZag is capable of automatically hardening client-side code against both known and previously-unknown vulnerabilities. Finally, we show that ZigZag introduces acceptable overhead in many cases, and is compatible with popular websites drawn from the Alexa Top 20 without developer or user intervention.

Kurzfassung

Im Gegensatz zu klassischen Serverseitigen Programmen zeichnet sich bei modernen Web Applikationen ein Trend ab, mehr Programmlogik auf die Clientseite auszulagern. Da die Popularität von HTML5 APIs wie etwa `postMessage` steigt, spielen Schwachstellen in clientseitigem Programmcode, im speziellen Client-Side Validation (CSV) Vulnerabilities eine immer wichtiger werdende Rolle. Bisher wurde vor allem die Erkennung und Abwehr von Angriffen gegen die Serverseite von Web Applikationen erforscht. Angriffe und Sicherungsmassnahmen für die Clientseite erfuhren weniger Aufmerksamkeit. Um in Zukunft Web Anwendungen besser zu schützen, müssen Clientseitige Angriffe wie CSV direkt im Browser erkannt werden, da serverseitige Schutzmechanismen dafür nicht eingesetzt werden können.

In dieser Masterarbeit wird ZigZag vorgestellt, ein System zur Härtung von JavaScript Web Applikationen gegen CSV Schwachstellen. ZigZag erlaubt es transparent clientseitigen Code zu instrumentieren um dynamische Erkennung von Invarianten für sicherheitsrelevanten Code durchzuführen. Es werden durch das System Modelle generiert die beschreiben in welcher Art Programmkomponenten auf der Clientseite interagieren können. ZigZag unterstüzt JavaScript welches serverseitig durch Templates generiert wurde, und vermeidet dadurch wiederholte Instrumentierung wenn JavaScript Programme mit ähnlicher Struktur bereits Instrumentiert wurden. Sobald genug Daten gesammelt wurden um ein Modell für ein Programm zu generieren, wird in den Enforcement-mode geschaltet um wahlweise Angriffe zu Erkennen oder Abzuwehren. Durch Evaluierung des Systems haben wir festgestellt, dass ZigZag fähig ist automatisch clientseitigen Code zu härten, um Angriffe sowohl gegen bekannte als auch bisher unbekannte Schwachstellen abzuwehren. Abschließend demonstrieren wir, dass der Overhead von ZigZag großteils akzeptabel ist. Weiters zeigen wir, dass unser System kompatibel ist mit Websites der Alexa Top 20, ohne Eingreifen von Entwicklern.

Contents

CHAPTER 1

Introduction

Most of the over 2 billion Internet users [27] regularly access the World Wide Web, performing a wide variety of tasks that range from searching for information to the purchase of goods and online banking transactions. Unfortunately, the popularity of web-based services and the fact that the web is used for business transactions has also attracted a large number of malicious actors. These actors compromise both web servers and end-user machines to steal sensitive information, to violate user privacy by spying on browsing habits and accessing confidential data, or simply to turn them into "zombie" hosts as part of a botnet.

As a consequence, significant effort has been invested to either produce more secure web applications, or to defend existing web applications against attacks. Examples of these approaches include applying static and dynamic program analyses to discover vulnerabilities or prove the absence of vulnerabilities in programs [16, 29, 40, 41], language-based approaches to render the introduction of certain classes of vulnerabilities impossible [34, 36, 39], sandboxing of potentially vulnerable code, and signature- and anomaly-based schemes to detect attacks against legacy programs.

However, despite the large amount of research that has been performed into preventing attacks against web applications, vulnerabilities persist. This is due to a combination of factors, including the difficulty of training developers to make use of more secure development frameworks or sandboxes, as well as the continuing evolution of the web platform itself.

The programming language that is used for most client-side components is JavaScript. Although it has been introduced in 1994, until recently there was a lack of big, complex applications. JavaScript comes from a background of small scripts that enhance the appearance and user experience of websites, whereas modern applications can exceed a MiB of code. While

```
1 var messagesURL = "http://csv-example.com";
2 popURL = "messages_popup.html";
3 var popup;
4 ...
5 function receiveMessages(msg) {
6    var message = getMessage();
7    popup.postMessage(message,messagesURL);
8 }
9 ...
10 popup = window.open(popURL);
11 for(;;) {
12    ...
13    receiveMessages(msg);
14    ...
15 }
```

Figure 1.1

security implications of small, isolated scripts were studied in the past, interacting and complex components, which are typical for new Web 2.0 applications, pose a different threat.

Advances in browser JavaScript engines and the adoption of HTML5 APIs have led to an explosion of highly complex web applications where the majority of application code has been pushed to the client. Client-side JavaScript components from different origins often co-exist within the same browser, and make use of HTML5 APIs such as `postMessage` to interact with each other in highly dynamic ways.

`postMessage` enables applications to communicate with each other purely within the browser, and are not subject to the classical same-origin policy (SOP) that defines how code from mutually untrusted principals are separated. While SOP automatically prevents client-side code from distinct origins from interfering with each others' code and data, code that makes use of `postMessage` is expected to define and enforce their own security policy. While this provides much greater flexibility to application developers, it also opens the door for vulnerabilities to be introduced into web applications due to insufficient origin checks.

`postMessage` is only one example of the more general problem of insufficient client-side validation (CSV) vulnerabilities. These vulnerabilities can be exploited by input from untrusted sources – e.g., the cross-window communication interface, referrer data, and others. An important property of these vulnerabilities is that attacks cannot be detected on the server side, and therefore any framework for defending against them at runtime must execute within the browser.

As a typical example of a CSV vulnerability, consider the simple code in Figure 1.1. The program opens a popup window (Line 10) and then uses this window to display messages received from a backend server at `csv-example.com` (Lines 11 to 15).

```
1 function displayMessage (evt) {
2   var message;
3   message = "I␣received␣" + evt.data;
4   ...
5 }
```

Figure 1.2

```
1 if (evt.origin !== "http://csv-example.com") {
2    message = "You␣are␣not␣worthy";
3    ...
4 }
5 else {
6     ...
7 }
```

Figure 1.3

The problem here is that it is the programmer's responsibility to validate that the messages that the popup window receives are indeed coming from the legitimate domain `csv-example.com`. The popup window might use code to receive and display messages, similar to what is depicted in Figure 1.2.

Unfortunately, the developer does not validate the origin, and hence, this code is vulnerable. Any malicious website could send a message to the popup window that is then displayed to the user. To fix this vulnerability, the developer would have to insert a piece of code that validates the origin, similarly to Figure 1.3

Although the example we provided is simple, similar to cross-site scripting (XSS) attacks, the possibilities of exploitation of a CSV by an attacker are only limited by her imagination. Among the possible attacks are origin misattribution, code injection, command injection, and cookie-sink attacks [41].

Given the strong trend towards web applications, and the increasing support for complex, client-side components by new HTML standards and browser improvements, we expect to see more and more critical applications developed following the Web 2.0 paradigm. Of course, this means that client-side vulnerabilities will become more widespread, and the severity of exploits will increase. This makes the development of new techniques to protect client-side code critical. ZigZag represents the first step towards this goal.

1.1 Motivation

To contextualize ZigZag and further motivate the problem of CSV vulnerabilities, we consider a hypothetical webmail service. This application is composed of code and resources belonging both to the application itself as well as advertisements from multiple origins. Since these origins are distinct, the SOP applies, and code from each of these origins cannot interfere with the others. This type of origin-based separation is typical for modern web applications.

However, in this example, the webmail component communicates with the advertising network via `postMessage` to request targeted ads given a profile it has generated for its users. The ad network can respond that it has successfully placed ads, or else request further information in the case that a suitable ad could not be found. Figure 1.4 shows one side of this communication channel, where the advertising component both registers an `onMessage` event listener to receive messages from the webmail component, as well as sends responses using the `postMessage` method. In this case, because the ad network does not verify the origin of the messages it receives, it is vulnerable to a CSV attack [43].

To tamper with the ad network, an attacker must be able to invoke `postMessage` in the same context. This can be achieved by exploiting XSS vulnerabilities from user content, or framing the webmail service. Hence, the attacker has to send an email to a victim user that contains XSS code, or lure the victim to a site that will frame the webmail service.

Despite the fact that the ad network component is vulnerable, ZigZag prevents successful exploitation of the vulnerability. With ZigZag, the webmail service is used through a transparent proxy that instruments the JavaScript code, augmenting each component with monitoring code. The webmail service then runs in a training phase where execution traces of the JavaScript programs are collected. Collected data points include function parameters, caller/callee pairs, and return values. Once enough execution traces have been collected, ZigZag uses invariant detection to establish a model of normal behavior. Next, the original program is extended with enforcement code that detects deviations from the baseline established during training. Execution is compared against this baseline, and violations are treated as attacks.

In this example, ZigZag would recognize that messages received by the ad network must originate from the webmail component's origin, and would terminate execution if a message is received from another origin – for instance, from the user content origin. We stress that this protection requires no changes to the browser or application on the server, and is therefore transparent to both developers and users alike.

We expand upon this example service with more vulnerabilities and learned invariants in following sections.

```
1  // Handle a received message
2  var receiveMessage = function(e) {
3      // Missing check on e.origin!
4  }
5
6  var sendMessage = function(e) {
7      ...
8      w.postMessage(data, '*');
9  }
10
11 window.addEventListener("message", receiveMessage, false);
```

Figure 1.4: Insecure usage of the `postMessage` API in a hypothetical webmail client-side component.

1.2 Threat Model

The threat model we assume for this work is as follows. ZigZag aims to defend benign-but-buggy JavaScript applications against attacks targeting client-side validation vulnerabilities, where CSV vulnerabilities represent bugs in JavaScript programs that allow for unauthorized actions via untrusted input.

The attacker can provide input to JavaScript programs through cross-window communication (e.g., `postMessage`), or window/frame cross-domain properties. This can be performed by operating in an otherwise isolated JavaScript context within the same browser. However, the attacker cannot run arbitrary code in a ZigZag-protected context without first bypassing ZigZag, an eventuality we aim to prevent. Therefore, we assume that attackers cannot directly tamper with ZigZag invariant learning and enforcement by, for instance, overwriting these functions in the JavaScript context without first evading the system.

Because ZigZag depends on a training set to learn dynamic invariants, we assume that the training data is trusted and, in particular, attack-free. This is a general limitation of anomaly-based detection schemes, though one that also has partial solutions [14].

1.3 Contributions

In this paper, we propose ZigZag, a system for hardening JavaScript-based web applications against client-side validation attacks. ZigZag transparently instruments client-side code to perform dynamic invariant detection over live browser executions. From this, it derives models of the normal behavior of client-side code that capture essential properties of how – and with whom – client-side web application components interact, as well as properties related to control flows

and data values within the browser. Using these models, ZigZag can then detect deviations that are highly correlated with client-side validation attacks.

We describe an implementation of ZigZag as a proxy, and demonstrate that it can effectively defend against vulnerabilities found in the wild against real web applications without modifications to the browser or application itself, aside from automated instrumentation. In addition, we show that ZigZag is efficient, and can be deployed in realistic environments without a significant impact on the user experience.

In summary, this master's thesis makes the following contributions:

- We present a novel in-browser anomaly detection system based on dynamic invariant detection that defends clients against previously unknown client-side validation attacks.
- We present a new technique we term *invariant patching* for extending dynamic invariant detection to server-side JavaScript templates, a very common technique for lightweight parameterization of client-side code.
- We extensively evaluate both the performance and security benefits of ZigZag, and show that it can be effectively deployed in several real scenarios, including as a transparent proxy or through direct application integration by developers.

1.4 Methodological Approach

To obtain a model of normal execution we let the userbase run the instrumented program in their browsers. Specifics of the program execution will be reported to our server. Those datapoints include function arguments, the callgraph, and others. Once we obtain enough data on a JavaScript program we can derive likely program invariants using our modified version Daikon [18]. Program executions in accordance with our defined model will be considered as benign, violations are marked as attacks. To secure the user from CSV attacks enforcement code is generated in program instrumentation, it will ensure that depending on the configuration either merely the user will be alerted of a possible attack, or the execution will halt.

1.5 Structure of the thesis

This chapter, the Introduction, gave an overview of the thesis and described the problems which ZigZag is addressing. Chapter 2 discusses Related Work and explains which shortcomings of the State of the Art will be improved. Chapter 3 describes the methodological approach taken to solve the discussed challenges and explains the architecture of the system in detail. Furthermore, in Chapter 4 an evaluation of the system will demonstrate how ZigZag can mitigate CSV

attacks. Chapter 5 concludes with a summary and discussion of limitations along with how those limitations can be overcome in future work.

1.6 Terminology

AJAX - Asynchronous JavaScript and XML A bundle of programming techniques used to make websites more interactive by transferring data in the background, hence - without having to reload the whole website. In practice JSON is often used instead of XML.

CSV - Client-side validation vulnerabilities A type of vulnerability that can be found in JavaScript programs which leverage HTML5 communication primitives. Exploitation of these vulnerabilities cannot be detected on the server-side.

ICAP - Internet Content Adaptation Protocol A protocol used to communicate between HTTP(S) proxies and content adaption programs.

JSON - JavaScript Object Notation A subset of JavaScript used for light-weight data exchange [15].

SOP - Same-Origin Policy A security concept that restricts data access depending on creator and accessor origin. Browsers are enforcing this policy on program code, cookies and others.

Web Workers Provides support for long running code execution in the background for web applications.

XHR - XMLHttpRequest An API to send HTTP requests from client programs. Modern websites use it for background data transfers that make the website more responsive. This technology is used by AJAX, other than XML, JSON is often used to encode data. By default, XHR [3] is restricted by the same-origin policy.

CHAPTER 2

Related Work

Securing applications against CSV, or more generally—securing web applications—is a well known problem, and an extensive body of work exists. This chapter provides an overview on the state of the art of existing techniques and compares them to ZigZag. Work on CSV, more general web security, program instrumentation as well as dynamic and static analysis are covered. Each section discusses related approaches, their advantages and disadvantages, and how ZigZag is addressing their shortcomings.

2.1 CSV detection

FLAX: Systematic Discovery of Client-side Validation Vulnerabilitiesin Rich Web Applications

CSV vulnerabilities were first highlighted by Saxena et al. [41]. In their work, the authors propose FLAX, a framework for CSV vulnerability discovery that combines dynamic taint analysis and fuzzing into taint-enhanced blackbox fuzzing. The system operates in multiple steps. First, JavaScript programs are translated into a simplified intermediate language called JASIL. Then, the JavaScript application under test is executed to dynamically identify all data flows from untrusted sources to critical sinks such as cookie writes, `eval`, or `XMLHttpRequest` invocations. This flow information is processed into small executable programs called acceptor slices. These programs accept the same inputs as the original program but are reduced in size. Next, the acceptor slices are fuzzed using an input-aware technique to find inputs to the original program that can be used to exploit a bug. A program is considered to be vulnerable when a data flow from an untrusted source to a critical sink can be established.

A Symbolic Execution Framework for JavaScript

Later, the same authors improved FLAX by replacing the dynamic taint analysis component with a static analyzer. Kudzu [40] is a tool for finding client-side code injection vulnerabilities in JavaScript applications by leveraging dynamic symbolic execution. The input space is divided into event space and value space, and is explored by employing both dynamic and static techniques. The event space can be mouse hovering, clicks, keyboard presses, and others. The value space is composed of data in form fields, cross-window communication data, request headers, and others. The application event space is first explored with an automatic random GUI exploration tool that is used as input to the dynamic symbolic interpreter that operates on a combination of concrete values (integers, strings, etc.) and symbolic values. Out of each recorded execution trace the symbolic values are recorded into a full path constraint. The authors considered off the shelf constraint languages as not expressive enough and created Kaluza, a specifically designed constraint language. The language is at least PSPACE hard and NP-complete for unbounded string length. Constraints are translated to a bit-vector and handed to STP, an SMT solver. Satisfiable assignments to constraints represent valid attacks.

The advantage of Kudzu is that it is an end-to-end system that will automatically generate and test exploits without user interaction. While it still relies on initial dynamic input like FLAX, it does not use random fuzzing in later stages. While bound-k completeness is provided, the required k could be too high to provide a feasible result. Also, the approach will only find bugs but it will not secure applications.

The main difference between ZigZag, Kudzu, and FLAX is that the latter two focus on detecting vulnerabilities in applications, while ZigZag is intended to defend unknown vulnerabilities against attacks.

2.2 Anomaly Detection

Daikon

Anomaly detection has found wide application in security research. For instance, Daikon [18] is a system that can infer likely invariants. An invariant is an observation over a property at a program point, where the system will infer the property always holds. Examples of invariants are that a number is always even, greater than a certain value, or a string always having one out of three values. These invariants can be translated into program assertions that can be checked at runtime. The system applies machine learning to make observations at runtime.

Daikon supports multiple programming languages, but can also be used over arbitrary data as comma-separate value files. In ZigZag, we extended Daikon with new invariants specific to JavaScript applications for runtime enforcement.

SRI IDES

An early example of anomaly detection in intrusion detection is the IDES Statistical Anomaly Detector by SRI International [28]. The system detects intrusion or misuse by authorized users in real-time. For each user historical audit data is used to create a statistical model that combines all audit variables into a profile that represents normal behavior. This multivariate model takes into account the relationships between audit variables and not simply deviations based on single variables. IDES is adaptive to slow changes in user behavior by using a decay model for old audit data. This technique provides a moving time window that will value more recent data over older data-points. User behavior that deviates from the expected behavior at a level that is higher than the defined threshold is considered a potential attack.

A Sense of Self for Unix Processes

Forrest et al. [19] presented an anomaly detection system observing exclusively the sequence of system calls. Analogies from an artificial immune system are drawn to distinguish self from other. Normal behavior is defined as a sequence of system calls with the length of 5, 6 and 11. The system operates in two stages. In stage one a database of benign behavior is trained, sliding windows over the sequence of system calls are recorded and stored in a database. The database depends on the configuration of the system and has to be trained for each program for each individual system. In stage two traces of calls can be checked against the database to test for anomalous behavior by comparing for the percentage of mismatches to expected behavior. However, anomaly detection based on system calls can be defeated with mimicry attacks [48].

Swaddler

Attacks on the workflow of PHP applications have been addressed by Swaddler [12]. Not all attacks on systems produce requests or, more generally, external behavior that can be detected as anomalous. These attacks can be detected by instrumenting the execution environment and generating models that are representative of benign runs. Swaddler can be operated in three modes: training, detection, and prevention. In training mode, the system is trained on benign user behavior. In detection mode attacks are reported, while in prevention mode execution is halted after an attack is detected. To model program execution, profiles for each basic block are generated, and both univariate and multivariate models are used. Univariate models describe

properties of single variables – for example, discovery of enums, length of attributes, character distribution in strings, and data types. Multivariate models describe the relationships between variables. These include likely invariants and variable presence or absence. During training, probability values are assigned to each profile by storing the most anomalous score for benign data, a level of "normality" is established. In detection and prevention mode, an anomaly score is calculated based on the probability of the execution data being normal using a preset threshold. Violations are assumed to represent attacks. The results suggest that anomaly detection on internal application state allows a finer level of attack detection than exclusively analyzing external behavior.

2.3 JavaScript Program Instrumentation

Our solution requires that client-side JavaScript is instrumented. Previous research has examined various ways in which JavaScript instrumentation can be implemented. JavaScript can be considered as self-modifying code since a running program can generate input code for its own execution. This renders complete instrumentation prior to execution impossible since writes to code cannot be covered. Hence, programs must be instrumented before execution and all subsequent writes to program code must be processed by separate instrumentation steps. A proxy-based approach to JavaScript instrumentation with enforcement of manually written policies was discussed by Yu et al. [31, 50]. Newly introduced program code is sent back to the server for instrumentation.

2.4 Static and Dynamic Analysis

Several approaches to analyze the behavior of programs exist. One distinction can be made between static and dynamic analysis, both have their advantages and disadvantages. Results of static analysis hold for all program executions, however, programs are generally too complex to allow for meaningful results due to complexity of the approach. Dynamic analysis on the other hand provides full truth for single executions. However, the main disadvantage of dynamic analysis is generalization to other executions. It is generally unclear whether the results deducted from the observed executions will hold for any other future execution. Also, dynamic analysis systems can be defeated in various ways. For example, if a malware author knows that the analysis system will terminate after three minutes she can wait for the payload to be executed after five minutes. The analysis system will miss the malware.

An example for a dynamic analysis system is Anubis [26]. To detect Windows malware the system runs PE-executables in an emulated environment that is based on Qemu. By inspecting

network traffic, file activity, registry activity and others programs can be classified as potentially malicious or benign. Wepawet [13] is an analysis system for detection of drive-by downloads triggered by malicious JavaScript code. The system uses anomaly detection to generate a profile of normal JavaScript executions. JavaScript code is emulated in the sandbox and compared against that profile, deviations are labeled as suspicious.

An example for static analysis of JavaScript programs is JSLint [16] by Douglas Crockford. This tool is mainly in use for quality checking purposes and can catch involuntarily introduced bugs. Static analysis approaches for CSV are discussed in Section 2.1.

2.5 White-box and Black-box Testing

Other than dynamic and static analysis, there are two different approaches [20] to test software applications for the presence of bugs and vulnerabilities: white-box testing and black-box testing. In white-box testing, the source code of an application is analyzed to find flaws. In contrast, in black-box testing, input is fed into a running application and the generated output is analyzed for unexpected behavior that may indicate errors.

When analyzing web applications for vulnerabilities, black-box testing tools [6, 10, 30, 47] are the most popular. Some of these tools claim to be generic enough to identify a wide range of vulnerabilities in web applications [6]. However, recent studies [9, 17] have shown that scanning solutions that claim to be generic have serious limitations, and that they are not as comprehensive in practice as they pretend to be.

Two well-known, older web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [42] and Huang et al.'s [24] vulnerability detection tool that automatically executes SQL injection attacks. Scott and Sharp's solution allows users to define fine-grained policies manually in order to prevent attacks such as parameter tampering and cross-site scripting.

With respect to white-box testing of web applications, a large number of static source code analysis tools [29, 45, 49] that aim to identify vulnerabilities have been proposed. These approaches typically employ taint tracking to help discover if tainted user input reaches a critical function without being validated. Previous research has shown that the sanitization process can still be faulty if the developer does not understand a certain class of vulnerability [7].

Note that there also exists a large body of more general vulnerability detection and security assessment tools, for example, Nikto [37], and Nessus [46]. Such tools typically rely on a repository of known vulnerabilities and test for the existence of these flaws.

With respect to scanning, there also exist network-level tools such as nmap [25]. Tools like nmap can determine the availability of hosts and accessible services. However, they cannot

detect higher-level application vulnerabilities or CSV vulnerabilities.

2.6 JavaScript Subsetting Techniques

Various methods have been developed to restrain JavaScript execution in advertising, mashups, or widgets by removing unsafe operations or reducing the functionality otherwise. Subsetting is relevant for ad display, as it provides the ad networks with the advantages of JavaScript while protecting the confidentiality of the web client.

AdSafe

One prominent example of subsetting is AdSafe [1]. It was developed to allow a safe subset of JavaScript to be used by untrusted parties. AdSafe prohibits accessing global variables or direct access to the DOM, this is a very coarse grained restriction. The AdSafe properties can be statically verified with JSLint during an approval step by ad networks. Applications that we target for instrumentation with ZigZag often require access to the DOM.

AdSentry

Another example for safe JavaScript in ad display is AdSentry which uses a shadow JavaScript engine for sandboxing. Access to the DOM is restricted based on policies that can be defined by both the publisher or client. Only a virtual DOM is available in the shadow environment that interfaces with the real DOM but will only allow interactions that are within the granted permissions. While this approach is a safe method to embed advertisements it is not suitable for detection of CSV. Also the requirement for developers to write additional annotations renders AdSentry infeasible in practice.

2.7 Client-Side Policy Enforcement

ICESHIELD

Based on analysis of malicious JavaScript programs, Heiderich et al. implemented ICESHIELD [23], to enforce policies on programs. By adding JavaScript code before all other content, ICESHIELD is invoked by the browser before any other code is executed. By employing ECMAScript 5 features, DOM properties are frozen to maintain the integrity of the detection code. The system protects users from drive-by downloads and exploit websites. In contrast, ZigZag performs online invariant detection and prevents previously unknown attacks. Furthermore, the goal of ZigZag is to harden benign-but-buggy programs as opposed to defend against ostensibly malicious code.

ConScript

ConScript [35] allows developers to create fine-grained security policies that specify the actions a script is allowed to perform and what data it is allowed to access or modify. Conscript can generate rules from static analysis performed on the server, as well as by inspecting dynamic behavior on the client. However, it requires modifications to the JavaScript engine, which ZigZag aims to avoid.

BrowserShield

The authors of BrowserShield [38] observed that the time between gaining knowledge of a vulnerability and the application of the patch—i.e., the window of vulnerability – is too high, and advocate for a more dynamic approach. BrowserShield rewrites JavaScript on the server to replace unsafe functions with safe equivalents. The instrumented code contains runtime checks that inspect generated code for vulnerabilities in the browser. However, the system depends on signatures that describe these vulnerabilities. Therefore, policies must be written manually and and defending against unknown vulnerabilities is not possible in general. In contrast, ZigZag uses an automated, anomaly-based approach that addresses both of these limitations.

Gatekeeper

In Gatekeeper [22], a static policy enforcement framework for JavaScript widgets is presented. To analyze the data flow of JavaScript one has to reason about the program heap. An inclusion-based Andersen-style flow—and context-sensitive analysis is used, Gatekeeper proposes the first points-to analysis for JavaScript. Policies are defined as Datalog queries. Datalog is a declarative language, a syntactic subset of Prolog. It is used in various query languages where SQL is inconvenient.

Gatekeeper translates the JavaScript program to a Datalog fact database and applies inference rules. The result of applying these rules is a CFG with points-to relations that is aware of heap allocation. Finally the bddbddb solver is used for detection of policy violations. An example of inference rules would be that an object property can be looked up either locally in the object or recursively through property chains.

The provided policies support prohibiting access to certain functions, code injection, browser redirection and others. Only a safe subset of JavaScript is supported, for example the *with* statement and features for dynamic code execution like `eval` and `new Function()` have been removed. The reason for removing code loading is obvious: dynamically loaded code can not be statically checked by the system ahead of time. However, due to its popularity writes to `document.innerHTML` are allowed. To allow for this run-time checks are added to the code.

Advantages of this approach are the fast runtime, 90% of the analyzed samples terminated in less than four seconds. The drawbacks are that Gatekeeper might not be applicable to larger JavaScript programs. It has to be considered that the analyzed widgets are below 300 lines of code on average, it is unclear whether the analysis scales to larger programs. Furthermore, the CFG is an over-approximation of the actual CFG and leads to false positives.

Staged information flow for JavaScript

Since the dynamic nature of JavaScript renders a purely static approach infeasible, the authors of *Staged information flow for JavaScript* [11] propose a staged approach where given a list of disallowed flow policies they first perform an initial analysis of the program and add residual-policy enforcement code to program points that dynamically load code (*holes*). Analysis of dynamically loaded code can be performed at runtime. This procedure is applied recursively to address dynamically loaded code. Policies can enforce integrity and confidentiality properties. Integrity policies would disallow writing to global variables or the DOM. Confidentiality policies would disallow reading data like `document.cookie`. Policies are a list of tuples of disallowed flows (from, to) where from and to are variables or holes. Flows to individual holes can not be specified, only flows to or from all holes can be disallowed. Like most projects that analyze JavaScript, the language is reduced to a subset here as well. Core JavaScript provides the features the authors identified as most commonly used and avoids language features like the *with*-statement, which is known to be hard to analyze.

The constraint generation is executed on the server, static taint propagation is performed based on the provided policies and outputs residual check functions. These residual checks, which are performed by the client before executing `eval`, are purely syntactic.

Alternatively to this approach a fully-dynamic enforcement or a static enforcement that re-analyzes the whole program whenever new code is loaded are possible. However, these approaches would lead to an overhead that is infeasible.

The staged approach allows enforcement of data flow policies without a strong performance hit. The drawbacks of this approach are that developers are forced to annotate code with policies. Also, since this approach requires modifications on the browser side it is unlikely to be used in practice.

The discussed projects address JavaScript security on different levels of granularity. Fine and coarse grained restrictions are possible. Policies need to be fine grained to be effective, which implies a higher complexity on policy creation. An automated approach is required for a technique to be feasible.

2.8 Web Standards

Although Barth et al. [8] made the HTML5 `postMessage` API more secure, analysis of websites suggests that it is nevertheless used in an insecure manner. Authentication weaknesses of popular websites have been discussed by Son et al. [43]. They showed that 84 of the top 10,000 websites were vulnerable to CSV attacks, and moreover these sites often employ broken origin authentication or no authentication at all. Their proposed defenses rely on modifying either the websites or the browser.

Content Security Policy

Content Security Policy (CSP) [5,44] is a framework for restricting JavaScript execution directly in the browser. CSP rules must be provided by the application developers and are not effective unless the application obeys certain programming practices. While automatic generation of these rules is possible in some cases, inline scripting and `eval` must be disabled to effectively contain XSS attacks. Furthermore, all JavaScript sources should be hosted locally to reduce dependencies on untrustworthy third-party servers. Websites often do not follow these standards, and CSP therefore often cannot be applied in a meaningful way to protect clients.

In ZigZag, we aim for a fine-grained, automated, annotation-free approach that dynamically secures applications against unknown CSV attacks in an unmodified browser.

CHAPTER 3

The System

3.1 System Overview

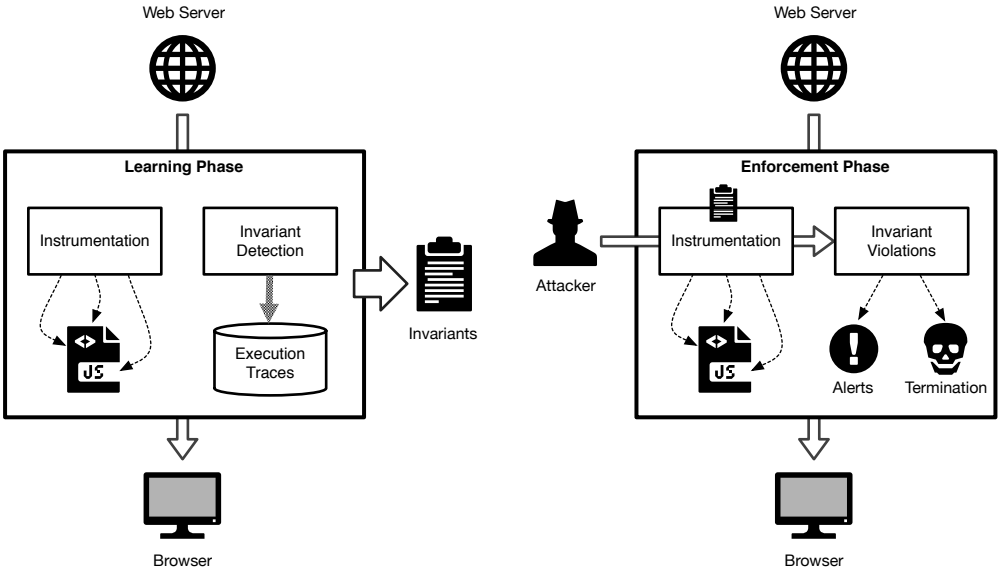
ZigZag is an in-browser anomaly detection system that defends against CSV vulnerabilities in JavaScript applications. ZigZag operates by interposing between web servers and browsers in order to transparently instrument JavaScript programs. This instrumentation process proceeds in two phases.

Learning phase

First, ZigZag rewrites programs with monitoring code to collect execution traces of client-side code. These traces are fed to a dynamic invariant detector that extracts likely invariants, or models. The invariants that ZigZag extracts are learned over data such as function parameters, variable types, and function caller and callee pairs.

Enforcement phase

In the second phase, the invariants that were learned in the initial phase are used to harden the client-side components of the application. The hardened version of the web application preserves the semantics of the original, but also incorporates runtime checks to enforce that execution does not deviate from what was observed during the initial learning phase. If a deviation is detected, the system assumes that an attack has occurred and execution is either aborted or the violation is reported to the user.



(a) Learning phase. A JavaScript program is instrumented in order to collect execution traces. Invariant detection is then performed on the trace collection in order to produce a set of likely invariants.

(b) Enforcement phase. Given a JavaScript program and the invariants previously learned, instrumentation is again used to enforce those invariants.

Figure 3.1: ZigZag overview. Instrumentation is used in both the learning and enforcement phases to produce and enforce likely invariants, respectively. Note that instrumentation is only performed *once* in each case; subsequent loads use a cached instrumented version of the program.

An overview of this system architecture is shown in Figure 3.1. We note that instrumentation for both the learning phase and enforcement phase is performed *once*, and subsequent accesses of an already instrumented program re-use a cached version of that program.

In the following sections, we describe in detail each phase of ZigZag's approach to defending against client-side vulnerabilities (CSVs) in web applications.

3.2 Invariant Detection

In this section, we focus on describing the invariants ZigZag learns, why we selected these invariants for enforcement, and how we extract these invariants from client-side code.

Data Type	Invariants
All	Type inference
Numbers	Equality, inequality, oneOf
String	Length, equality, oneOf, isJSON, isPrintable, isEmail, isURL, isNumber
Boolean	Equality
Objects	All of the above for object properties
Functions	Calling function, return value

Table 3.1: Invariants supported by ZigZag.

Program Invariants

Dynamic program invariants are statistically-likely assertions established by observing multiple program executions. We capture program state at checkpoints and compare subsets of these states for each individual checkpoint. The underlying assumption is that invariants should hold not only for the observed executions, but also for future ones.

However, there is no guarantee that invariants will also hold in the future. Therefore, ZigZag only uses invariants which should hold with a high probability. These invariants are later used to decide whether a program execution is to be considered anomalous. By capturing state dynamically, ZigZag has insight into user behavior, which purely static systems lack.

ZigZag uses program execution traces to generate Daikon [18] dtrace files. These dtrace files are then generalized into likely invariants with a modified version of Daikon we have developed. Daikon is capable of generating both univariate and multivariate invariants. Univariate invariants describe properties of a single variable; examples of this include the length of a string, the percentage of printable characters in a string, and the parity of a number. Multivariate models, on the other hand, describe relations between two or more variables, for example $x == y$, $x + 5 == y$, or $x < y$.

ZigZag analyzes multivariate relationships within function parameters, return values, and invoking functions. In addition, we extended the invariants provided by Daikon with additional ones, including checks on whether a string is a valid JSON object, URL, or email address.

For example, when used on a website with `postMessage`, ZigZag could learn that the origin attribute of the `onMessage` event is both printable and a URL, or equal to a string. Depending on the number of different origins, the system could also learn the legitimate set of sending origins

$$\text{v0.origin} \in \{o_1, \ldots, o_n\}.$$

As another example, since JavaScript is a dynamically typed language, it has no type annotations in function signatures. This language feature can lead to runtime errors or be exploited by

an attacker. By learning likely type invariants over function parameters and return values that are checked during the enforcement phase, ZigZag can often partially retrofit types into JavaScript programs. For example, this can become security relevant when developers use numeric values for input, and do not consider other values during input sanitization. We describe an example of parameter injection, and how the attack is thwarted by ZigZag in Section 4

The full set of invariants supported by ZigZag is shown in Table 3.1.

Program Instrumentation

Trace collection and enforcement code is inserted at program points we refer to as *checkpoints*. The finest supported granularity is to insert checkpoints for every statement. However, while this is possible, statement granularity introduces unacceptable overhead with little benefit. The CSV vulnerabilities we have observed in the wild can be detected with a coarser and more efficient level of granularity. Events such as receiving cross-window communication are handled by functions, thus function entry and exit points become good choke points to analyze input and return data. Consequently, for our prototype we opted to insert checkpoints at the start and end of functions.

During instrumentation, ZigZag performs a lightweight static analysis on the program's abstract syntax tree (AST) to prune the set of checkpoints that must be injected. Functions which contain `eval` sinks, XHR requests, access to the `document` object, and other potentially harmful operations are labeled as important. Only these functions are used in data collection and enforcement mode. As a consequence, large programs that only have few potentially harmful operations will have significantly less overhead as compared to instrumenting the entire program, while at the same time preserving the security of the overall approach.

Each function labeled as important during the static analysis phase is instrumented with pre- and post-function body hooks called `calltrace` and `exittrace`. The original return statement is inlined in the `exittrace` function call and returned by it. These functions access the instrumented function's parameters through the standard `arguments` variable, and either records a program state for invariant detection (learning phase) or checks for an invariant violation (enforcement phase).

ZigZag uses a number of identifiers to label program states at checkpoints. `functionid` uniquely identifies functions within a program, `codeid` labels distinct JavaScript programs, and `sessionid` labels program executions. The variables `functionid` and `codeid` are hardcoded during program instrumentation, while `sessionid` is generated for each request.

The `callcounter` variable is used instead to connect call chains. Every invocation of `calltrace` increments and returns a global `callcounter` to provide a unique identifier

```
1   function x(a, b) {
2       // function body
3       ...
4       return a+b;
5   }
```

(a) Function body before instrumentation

```
1   function x(a, b) {
2       var callcounter = __calltrace(functionid,
3                                     codeid,
4                                     sessionid);
5       // function body
6       ...
7       return __exittrace(functionid,
8                          callcounter,
9                          subexitid,
10                         codeid,
11                         sessionid,
12                         a+b);
13  }
```

(b) Function body after instrumentation

Figure 3.2: Function instrumentation example.

such that `calltrace` and `exittrace` invocations can be matched. This is necessary since JavaScript is re-entrant, and therefore multiple threads of execution can invoke a function and yield before returning, potentially resulting in out-of-order pre- and post-function hook invocations.

ZigZag can not only instrument the code initially loaded by a site, but also code dynamically downloaded during execution. JavaScript can be considered self-modifying code since a program can generate code for its own execution. We address this by wrapping `eval` invocations, script tag insertion, and writes to the DOM. Our wrapper sends the new program code to the proxy and calls the original function with the instrumented program. This technique has been shown to be effective in prior work [31].

In our prototype implementation, each of these calls incurs a roundtrip to the server, where such code is treated the same way as non-`eval` code. As a possible optimization, the instrumented version of previously observed data passed to `eval` could be inlined with the enclosing (instrumented) program, removing the need for subsequent separate roundtrips. Furthermore, we often observed `eval` to be used for JSON deserialization. If such a use case is detected, instrumentation could be bypassed entirely. However, we did not find it necessary to implement

these features in our research prototype.

The `calltrace` and `exittrace` functions reside in the same scope since they must be callable from all functions. An example of uninstrumented and instrumented code is shown in Figures 3.2a and 3.2b, respectively.

3.3 Invariant Enforcement

Given a set of invariants collected during the learning phase, ZigZag then instruments JavaScript programs to enforce these invariants. Since templated JavaScript is a prevalent technique on the modern web for lightweight parameterization of client-side code, we then present a technique for adapting invariants to handle this case. Finally, we discuss possible deployment scenarios and limitations of the system.

Daikon supports invariant output for several languages, including C++, Java, and Perl. However, it does not support JavaScript by default. Groeneveld et al. implemented extensions to Daikon to support invariant analysis using Daikon [21]. However, we found that their implementation was not capable of generating JavaScript for all of the invariants ZigZag must support, and therefore we wrote our own implementation.

In our implementation, the `calltrace` and `exittrace` functions perform a call to an enforcement function generated for each function labeled important during the static analysis step. `calltrace` examines the function input state, while `exittrace` examines the return value of the original function. These functions are generated automatically by ZigZag for each important function. Based on the invoking program point, assertions corresponding to learned invariants are executed. Should an assertion be violated, a course of action is taken depending on the system configuration. Options include terminating execution by navigating away from the current site, or alternatively reporting to the user that a violation occurred and continuing execution. Figure 3.3 shows a possible instance of the `calltrace` function, abbreviated for clarity.

3.4 Program Generalization

Modern web applications often make use of lightweight templates on the server, and sometimes within the browser as well. These templates usually take the form of a program snippet or function that largely retains the same structure with respect to the AST, but during instantiation placeholders in the template are substituted with concrete data – for instance, a timestamp or user identifier. This is often done for performance, or to reduce code duplication on the server. As an example, consider the templated version of the webmail example shown in Figure 3.5.

```
__calltrace = function(functionid, codeid, sessionid) {
    // Enforcement
    var v0 = arguments.callee.caller.caller.arguments[0];
    var v1 = ...

    if ( functionid === 0 ) {
        __assert(typeof(v0) === 'number' && v0 > 5);
        __assert(typeof(v1) === 'string' && v1 === "x");
        ...
    } else if ( functionid === 1 ) {
        ...
    }
    ...
    return __incCallCounter();
}
```

Figure 3.3: Example of invariant enforcement over a function's input state.

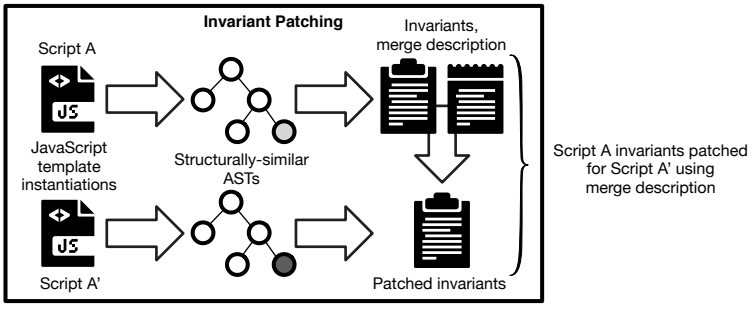


Figure 3.4: Invariant patching overview. If ZigZag detects that two JavaScript programs are structurally isomorphic aside from constant assignments, a merge description is generated that allows for efficient patching of previously-generated invariants. This scheme allows ZigZag to avoid re-instrumentation of templated JavaScript on each load.

Due to the cost of instrumentation and the prevalence of this technique, templated JavaScript poses a fundamental problem for ZigZag since a templated program causes – in the worst case – instrumentation on every resource load. Additionally, each template instantiation would represent a singleton training set, leading to artificial undertraining. Therefore, it was necessary to develop a technique for both recognizing when templated JavaScript is present and, in that case, to generalize invariants from a previously instrumented template instantiation to keep ZigZag tractable for real applications.

ZigZag handles this issue by using efficient structural comparisons to identify cases where templated code is in use, and then performing *invariant patching* to account for the differences between template instantiations in a cached instrumented version of the program.

```
1 // Server-Side JavaScript Template:
2 var state = { user: {{username}},
3               session: {{sessionid}} }
4
5 // Client-Side JavaScript code:
6 var state = { user: "UserX", session: 0 }
```

Figure 3.5: Example of a JavaScript template.

Structural Comparison

ZigZag defines two programs as structurally similar and, therefore, candidates for generalization if they differ only in values assigned to either primitive variables such as strings or integers, or as members of an array or object. Determining whether this is the case could be performed in a straightforward way by pairwise AST equality that ignores constant values in assignments. However, this straightforward approach does not scale when a large number of programs have been instrumented.

Therefore, we devised a string equality-based technique. From an AST, ZigZag extracts a string-based summary that encodes a normalized AST that ignores constant assignments. In particular, normalization strips all constant assignments of primitive data types encountered in the program. Also, assignments to object properties which have primitive data types are removed. Objects, however, cannot be removed completely as they can contain functions which are important for program structure. Removing primitive types is important as many websites generate programs that depend on the user state – e.g., setting `{logged_in: 1}` or omitting that property depending on whether a user is logged in or not. Simply removing the assignment allows ZigZag to correctly handle cases such as these.

Furthermore, normalization orders any remaining object properties such as functions or enclosed objects, in order to avoid comparison issues due to non-deterministic property orderings. Finally, the structural summary is the hash of the reduced, normalized program.

As an optimization, if the AST contains no function definitions, ZigZag skips instrumentation and serves the original program. This check is performed as part of structural summary generation, and is possible since ZigZag performs function-level instrumentation.

Code that is not enclosed by a function will not be considered. Such code cannot be addressed through event handlers and is not accessible through `postMessage`. However, calls to `eval` would invoke a wrapped function, which is instrumented and included in enforcement rules.

Fast Program Merging

The first observed program is handled as every other JavaScript program because ZigZag cannot tell from one observation whether a program represents a template instantiation. However, once ZigZag has observed two structurally similar programs, it transparently generates a *merge description* and *invariant patches* for the second and future instances.

The merge description represents an abstract version of the observed template instantiation that can be patched into a functional equivalent of new instantiations. To generate a merge description, ZigZag traverses the full AST of structurally similar programs pairwise to extract differences between the instantiations. Matching AST nodes are preserved as-is, while differences are replaced with placeholders for later substitution. Next, ZigZag compiles the merge description with the Closure compiler [4] to add instrumentation code and optimize.

The merge description is then used every time the templated resource is subsequently accessed. The ASTs of the current and original template instantiations are compared to extract the current constant assignments, and the merge description is then patched with these values for both the program body as well as any invariants to be enforced. By doing so, we bypass repeated, possibly expensive, compilations of the code. We visualized the approach in Figure 3.4.

3.5 Deployment Models

We note that several scenarios for ZigZag deployment are possible. First, application developers or providers could perform instrumentation on-site, protecting all users of the application against CSV vulnerabilities. Since no prior knowledge is necessary in order to apply ZigZag to an application, this approach is feasible even for third parties. And, in this case there is no overhead incurred due to re-instrumentation on each resource load.

On the other hand, it is also possible to deploy ZigZag as a proxy. In this scenario, network administrators could transparently protect their users by rewriting all web applications at the network gateway. Or, individual users could tunnel their web traffic through a personal proxy, while sharing generated invariants within a trusted crowd.

CHAPTER 4

Evaluation

To evaluate ZigZag, we implemented a prototype of the approach using the proxy deployment scenario. We used a Squid [2] ICAP module to interpose on HTTP(S) traffic, and a modified version of the Google Closure compiler [4] to instrument JavaScript code.

Our evaluation first investigates the security benefits that ZigZag can be expected to provide to potentially vulnerable JavaScript-based web applications. Second, we evaluate ZigZag's suitability for real-world deployment by measuring its performance overhead over microbenchmarks and real applications.

Synthetic Application

We evaluated ZigZag on the hypothetical webmail system first introduced in Section 1.1. This application is composed of three components, each isolated in iframes with different origins that contain multiple vulnerabilities. These iframes communicate with each other using `postMessage` on `window.top.frames`.

We simulate a situation in which an attacker is able to control one of the iframes, and wants to inject malicious code into the other origins or steal personal information. The source code snippets are described in Figures 4.1 and 4.2.

From the source code listings, it is evident that the webmail component is vulnerable to parameter injection through the `markemail` property. For instance, injecting the value `1&deleteuser=1` could allow an attacker to delete a victim's profile. Also, the ad network uses an `eval` construct for JSON deserialization. While highly discouraged, this technique is still commonly used in the wild and can be trivially exploited by sending code instead of a JSON object.

```
1 // Dispatches received messages to appropriate function
2 if (e.data.action == "markasread") {
3     markEmailAsRead(e.data);
4 }
5
6 // Communication with the server to mark emails as read
7 function markEmailAsRead(data) {
8     var xhr = new XMLHttpRequest();
9     xhr.open("POST", serverurl, true);
10    xhr.send("markasread=" + data.markemail);
11 }
12
13 // Communication with the ad network iframe
14 function sendAds(e) {
15    window.top.frames[i].postMessage({
16        'topic': 'ads',
17        'action': 'showads',
18        'content': '{JSON␣string}'
19    }, "*");
20 }
```

Figure 4.1: Vulnerable webmail component.

```
1 // Receive JSON object from webmail component
2 function showAds(data) {
3     var received = eval("(" + data.content + ")");
4     // Work with JSON object...
5 }
```

Figure 4.2: Vulnerable ad network component.

We first used the vulnerable application through the ZigZag proxy in a learning phase consisting of 30 sessions over the course of half an hour. From this, ZigZag extracted statistically likely invariants from the resulting execution traces. ZigZag then entered the enforcement phase. Using the site in a benign fashion, we verified that no invariants were violated in normal usage.

For the webmail component, and specifically the function handling the `XMLHttpRequest`, ZigZag generated the following invariants.

1. The function is only called by one parent function
2. `v0.topic === 'control'`
3. `v0.action === 'markasread'`
4. `typeof(v0.markemail) === 'number' && v0.markemail >= 0`
5. `typeof(v0.topic) === typeof(v0.action) &&`

```
v0.topic < v0.action
```

For the ad network, ZigZag generated the following invariants.

1. The function is only called by one parent function
2. `v0.topic === 'ads'`
3. `v0.action === 'showads'`
4. v0.content is JSON
5. v0 is printable
6. `typeof(v0.topic) === typeof(v0.action) &&`
   `v0.topic < v0.action`
7. `typeof(v0.topic) === typeof(v0.content) &&`
   `v0.topic < v0.content`
8. `typeof(v0.action) === typeof(v0.content) &&`
   `v0.action < v0.content`

Next, we attempted to exploit the webmail component by injecting malicious parameters into the `markemail` property. This attack generated an invariant violation since the injected parameter was not a number greater than or equal to zero.

Finally, we attempted to exploit the vulnerable ad network component by sending JavaScript code instead of a JSON object to the `eval` sink. However, this also generated an invariant violation, since ZigZag learned that `data.content` should always be a JSON object – i.e., it should not contain executable code.

Real-World Case Studies

In our next experiment, we tested ZigZag on four real-world applications that contained different types of vulnerabilities. These vulnerabilities are a combination of previously documented bugs as well as newly discovered vulnerabilities.[1] The aim of this experiment is to demonstrate that the invariants ZigZag generates can handle different types of exploitable programming mistakes.

For each of the following case studies, we first trained ZigZag by manually browsing the application with one user for five minutes, starting with a fresh browser state four times. Next, we switched ZigZag to the enforcement phase and attempted to exploit the applications. We consider the test successful if the attacks are detected with no false alarms. In each case, we list the relevant invariants responsible for attack prevention.

[1] For the vulnerabilities we discovered, we notified the respective website owners.

canadadry.com

This application does not check the origin of received messages. Therefore, by iframing the site, an attacker can execute arbitrary code if the message has a specific format, such as `capture:x; alert(3):`. This is due to the fact that the function that acts as a message receiver will, under certain conditions, call a handler that evaluates part of the untrusted message string as code. Both functions were identified as important by ZigZag's lightweight static analysis. We note that this vulnerability was previously reported in the literature [43], is acknowledged in comments in the program, and has not yet been fixed.

For the event handler, ZigZag generated the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'https://dpsg.janraincapture.com'`
3. `v0.data === 's1' || v0.data === 's2'`[2]
4. `v0.data` is printable

For the function that is called by the event handler, ZigZag generated the following invariants.

1. The function is only called by the receiver function
2. `v0 === 's1' || v0 === 's2'`[3]

The attack is thwarted by restricting the receiver origin, only allowing two types of messages to be received, and furthermore restricting control flow to the dangerous sink.

playforex.ru

This application contains an incorrect origin check that only tests whether the message origin *contains* the expected origin (using `indexOf`), not whether the origin equals or is a subdomain of the allowed origin. Therefore, any origin containing the string "playforex.ru" such as "playforex.ru.attacker.com" would be able to iframe the site and evaluate arbitrary code in that context. We reported the bug and it was promptly fixed.

ZigZag generated the following relevant invariants.

1. The function is only invoked from the global scope or as an event handler

[2] s1 and s2 were long strings, which we omitted for brevity.
[3] s1 and s2 were long strings, which we omitted for brevity.

2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://playforex.ru'`
3. `v0.data === '$('#right_buttons').hide();||`
   `v0.data === 'calculator()'`

ZigZag detected that the `onMessage` event handler only receives two types of messages, which manipulate the UI to hide buttons or show a calculator. By only accepting these two types of messages, arbitrary execution can be prevented.

Yves Rocher

This application does not perform an origin check on received messages, and all received code is executed in an `eval` sink. The bug has been reported to the website owners. ZigZag generated the following relevant invariant.

1. `v0.origin === 'http://static.ak.facebook.com' || v0.origin === 'https://s-static.ak.facebook.com'`

From our manual analysis, this program snippet is only intended to communicate with Facebook, and therefore the learned invariant above is correct in the sense that it prevents exploitation while preserving intended functionality.

adition.com

This application is part of a European ad network. It used a `new Function` statement to parse untrusted JSON data, which is highly discouraged as it is equivalent to an `eval`. In addition, no origin check is performed. This vulnerability allows attackers that are able to send messages in the context of the site to replace ads without having full JavaScript execution.

ZigZag learned that only valid JSON data is received by the function, which would prevent the attack based on the content of received messages. This is different than the Yves Rocher example, as data could be transferred from different origins while still securing the site. The bug was reported and promptly fixed by safely parsing JSON data.

Performance Overhead

Instrumentation via a proxy incurs performance overhead in terms of latency in displaying the website in the browser. We quantify this overhead in a series of experiments to evaluate the time required for instrumentation, the worst-case runtime overhead due to instrumentation, and the increase in page load latency for real web applications incurred by the entire system.

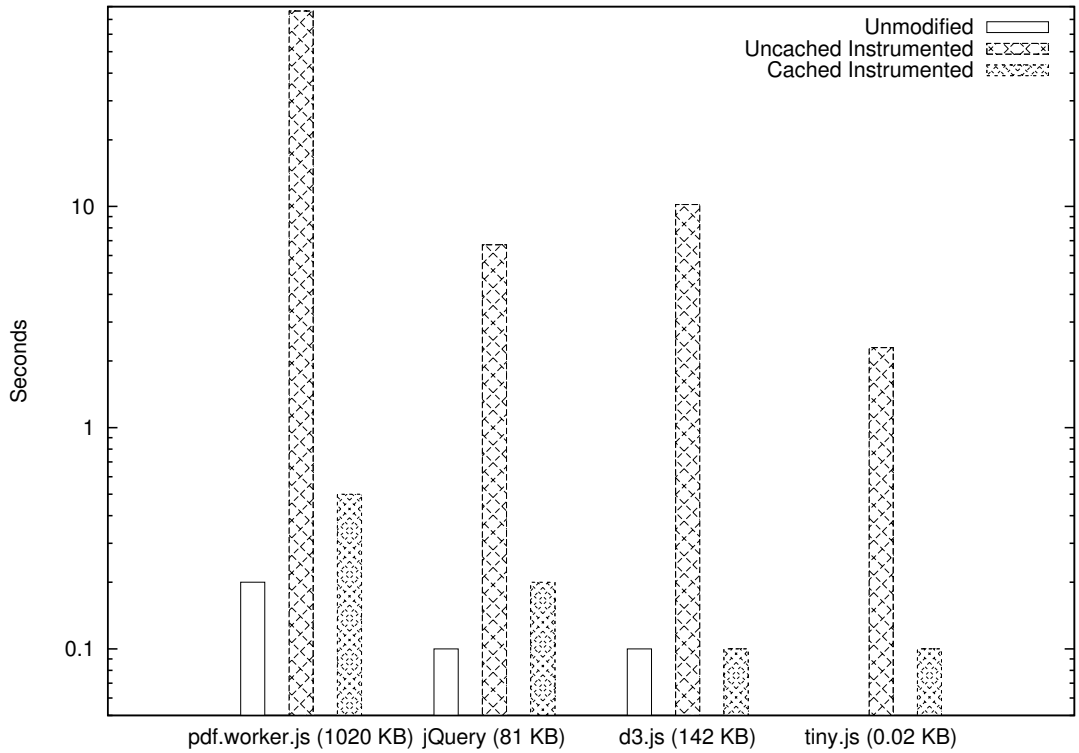


Figure 4.3: Instrumentation overhead for individual files. While the initial instrumentation can take a significant amount of time for large files, subsequent instrumentations have close to no overhead.

Instrumentation overhead.

We tested the instrumentation time of standalone files to measure ZigZag's impact on load times. As samples, we selected a range of popular JavaScript programs and libraries: Mozilla pdf.js, an in-browser PDF renderer; jQuery, a popular client-side scripting library; and, d3.js, a library for data visualization. Where available, we used compressed, production versions of the libraries. As Mozilla pdf.js is not minified by default, we applied the yui compressor for simple minification before instrumenting. The worker file is at 1.5 MB uncompressed and represents an atypically large file. Additionally, we instrumented a simple function that returns the value of `document.cookie`. We performed 10 runs for cold and warm testing each. For cold runs, the database was reset after every run.

Figure 4.3 shows that while the initial instrumentation can be time-consuming for larger files, subsequent calls will incur low overhead.

Microbenchmark

To measure small-scale runtime enforcement overhead, we created a microbenchmark consisting of a repeated `postMessage` invocation where one iframe (A) sends a message to another iframe (B), and B responds to A. Specifically, A sends a message object containing a property `process` set to the constant 20. B calculates the Fibonacci number for `process`, and responds with another object that contains the result.

We trained ZigZag on this simple program and then enabled enforcement mode. We then ran the program in both uninstrumented and instrumented forms. We measured the elapsed time between sending a message from A to B and reception of the response from B to A. We used the high resolution timer API `window.performance.now` to measure the round trip time, and ran the test 100 times each. The results of this benchmark are shown in Table 4.1.

ZigZag learned and enforced the following invariants for the receiving side.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://example.com'`
3. `v0.data.process === 20`
4. `typeof(v0) === typeof(v0.data)`
5. `typeof(v0.timeStamp) === typeof(v0.data.process) &&`
   `v0.timeStamp > v0.data.process`

For the message receiver that calculates the response, ZigZag learned and enforced the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://example.com'`
3. `typeof(v0.data.process) === 'number' &&`
   `v0.data.process === 20`
4. `typeof(v0.timestamp) === typeof(v0.data.process)`

Finally, for the receiver of the response, ZigZag learned and enforced the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://example.com'`

	Uninstrumented	Instrumented
Average Runtime	3.11 ms	3.77 ms
Standard Deviation	1.80	0.54
Confidence (0.05)	0.11	0.35

Table 4.1: Microbenchmark overhead.

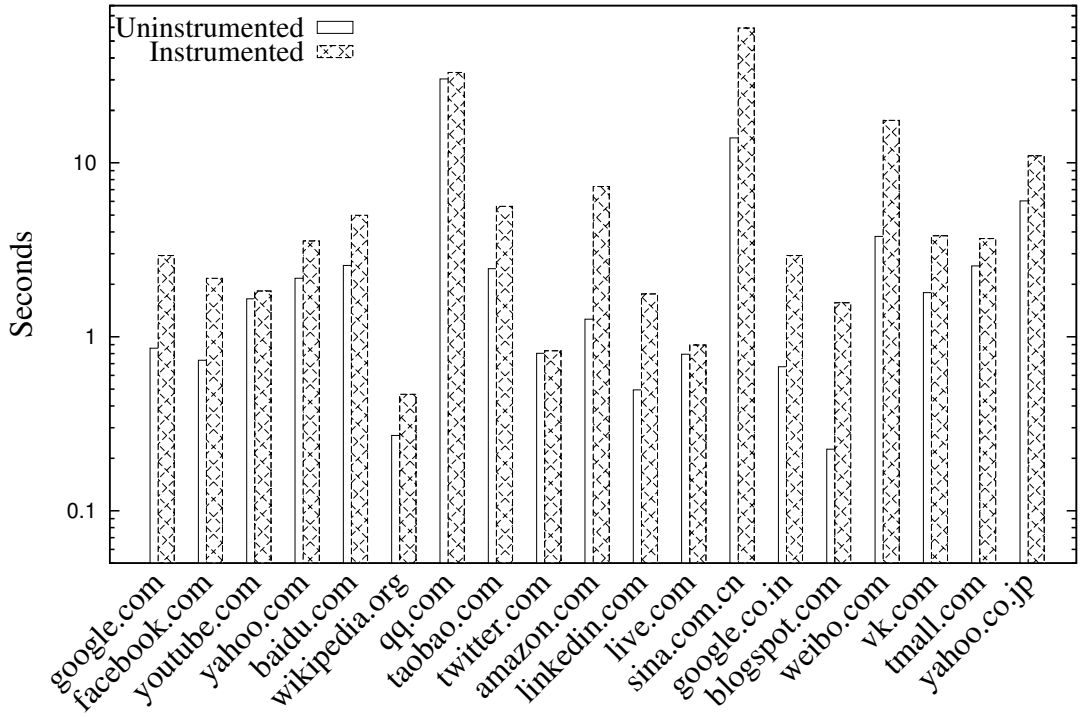


Figure 4.4: End-to-end performance benchmark on the Alexa 20 most popular websites (excluding hao123.com as it is incompatible with our prototype). A site is considered to be done loading content when the `window.load` event is fired, indicating that the entire contents of the DOM has finished loading.

3. `v0.data.response === 6765`
4. `typeof(v0) === typeof(v0.data)`
5. `typeof(v0.timeStamp) === typeof(v0.data.response) && v0.timeStamp > v0.data.response`

The above invariants represent a tight bound on the allowable data types and values sent across between each origin.

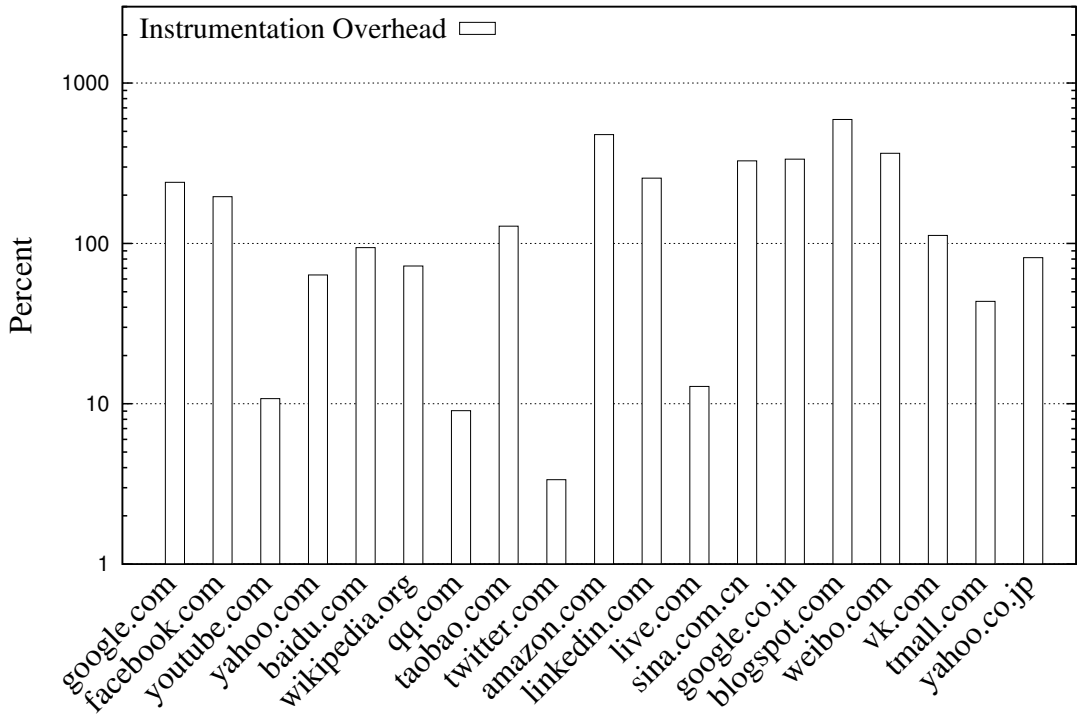


Figure 4.5: Relative load times for uninstrumented and instrumented programs.

End-to-End Benchmark

To quantify ZigZag's impact on the end-to-end user experience, we measured page load times on the Alexa Top 20. First, we manually inspected the usability of the sites and established a training set for enforcement mode. To do so, we browsed the target websites for half an hour each.

We used Chrome to load the site and measure the elapsed time from initial request to the `window.load` event, when the DOM completed loading (including all sub-frames). (We note, however, that websites can become usable before that event fires.) The browser was unmodified, with only one extension to display page load time.

Uninstrumented sites are loaded through the same HTTP(S) proxy ZigZag resides on, but the program text is not modified. Instrumented programs are loaded from a ZigZag cache that has been previously filled with instrumented code and merge descriptions. However, we do not cache original web content, which is loaded fresh every time.

The performance overhead is absolute and relative terms is depicted in Figure 4.5. We excluded `hao123.com` from the measurement as it was incompatible with our prototype. On average, load times took 4.8 seconds, representing an overhead of 180.16%, with median values of 2.01 seconds and an overhead of 112.10%. We found server-side templated JavaScript to be popular with the top-ranked websites. In particular, `amazon.com` served 15 such templates, and only 6 out of 19 serve no such templates.

`sina.com.cn` is an obvious outlier, with an absolute average overhead of 45 seconds. With 115 inlined JavaScript snippets and 112 referenced JavaScript files, this is also the strongest user of inline script. Furthermore, we noticed that the site fires the `DOMContentLoaded` event in less than 6 seconds. Hence, the website appears to become usable quickly even though not all sub-resources have finished loading.

In percentages, the highest overhead of 593.36% is introduced for `blogspot.com`, which forwards to Google. This site has the shortest uninstrumented loading time (0.226 seconds) in our data set, hence an absolute overhead will have the strongest implications on relative overhead.

Program Generalization

As discussed in Section 3.4, ZigZag supports structural similarity matching and invariant patching for templated JavaScript to avoid singleton training sets and excessive instrumentation when templated code is used. We measured the prevalence of templated JavaScript in the Alexa Top 50, and found 185 instances of such code. In addition, the median value per site was three. Without generalization and invariant patching, ZigZag would not have generated useful invariants and, furthermore, would perform significantly worse due to unnecessary re-instrumentation on template instantiations. Table 4.2 provides an overview of script types used on the Alexa Top 20.

Compatibility

To check that ZigZag is compatible with real web applications, we ran ZigZag on several complex, benign JavaScript applications. Since ZigZag relies on user interaction and the functionality of a complex web application is not easily quantifiable, we added manual quantitative testing to augment automated tests. The testers were familiar with the websites before using the instrumented version, and we performed live instrumentation using the proxy-based prototype.

For YouTube and Vimeo, the testers browsed the sites and watched multiple videos, including pausing, resuming, and restarting at different positions. Facebook was tested by scrolling through several timelines and using the chat functionality in a group setting. The testers also

Rank	Website	Script Type		
		src	inline	template
1	google.com	0	9	3
2	facebook.com	1	11	5
3	youtube.com	3	16	2
4	yahoo.com	8	24	4
5	baidu.com	5	12	1
6	wikipedia.org	1	0	0
7	qq.com	6	23	8
8	taobao.com	12	36	0
9	twitter.com	0	4	0
10	amazon.com	29	58	15
11	linkedin.com	11	36	1
12	live.com	4	8	0
13	sina.com.cn	112	115	1
14	google.co.in	2	9	3
16	blogspot.com	0	11	1
17	weibo.com	29	23	0
18	vk.com	8	13	1
19	tmall.com	40	11	0
20	yahoo.co.jp	8	38	2

Table 4.2: Types of script usage over the Alexa Top 20 websites. Either included with the src attribute or inline, and out of those how many were detected as server-side templates. We omitted `hao123.com` as it is incompatible with ZigZag.

posted to a timeline and deleted posts. For Google Docs, the testers created and edited a document, closed it, and re-opened it. For d3.js, the testers opened several of the example visualizations and verified that they ran correctly. Finally, the testers sent and received emails with Gmail and live.com.

In all cases, no enforcement violations were detected when running the instrumented version of these web applications.

CHAPTER 5

Summary and Future Work

5.1 Comparison with Related Work

Prior work in CSV detection was focused on vulnerability discovery. Our approach is complementary as it protects programs against such attacks. We do not rely on signatures or prior knowledge of attacks, manual annotations, or policies that will restrain program execution. Our approach can detect unknown attacks by monitoring for deviations from the trained model alone. ZigZag does not rely on modifications to the browser, all monitoring and enforcement code is added to the JavaScript code by either a transparent proxy, or on-site program rewriting.

5.2 Conclusion

Most websites rely on JavaScript to improve the user experience on the web. With new HTML5 communication primitives such as `postMessage`, inter-application communication between programs of different origins in the in the browser is possible, without performing round trips to the server. However, since new APIs are not subject to the same origin policy and, through software bugs, such as broken or missing input validation, applications can be vulnerable to attacks against these client-side validation vulnerabilities (CSV). As such attacks occur on the client-side, server-side security measures are ineffective in detecting and preventing them.

In this paper, we present ZigZag, an approach to automatically defend benign-but-buggy JavaScript applications against CSV attacks. Our method leverages dynamic analysis and anomaly detection techniques to learn and enforce statistically-likely, security-relevant invariants. Based on these invariants, ZigZag generates assertions that are enforced at runtime. Based on configuration, ZigZag can either warn users of attacks, or terminate program execution before an

attack occurs. ZigZag's design inherently protects against unknown vulnerabilities as it enforces learned, benign behavior. Runtime enforcement is carried out only on the client-side code, and does not require modifications to the browser.

ZigZag can be deployed by either the website operator or a third party. Website owners can secure their JavaScript applications by replacing their programs with a version hardened by ZigZag, thereby protecting all users of the application. Third parties, on the other hand, can deploy ZigZag using a proxy that automatically hardens any website visited using it. This usage model of ZigZag protects all users of the proxy, regardless of the web application.

We evaluated ZigZag using a number of real-world web applications, including complex examples such as online word processors, video portals, and websites out of the Alexa Top 20. Our evaluation shows that ZigZag can successfully instrument complex applications and prevent attacks while not impairing the functionality of the tested web applications. Furthermore, it does not incur an unreasonable performance overhead and, thus, is suitable for real-world usage.

5.3 Future Work

ZigZag's goal is to defend against attackers that desire to achieve code execution within an origin, or act on behalf of the target. The system was not designed to be stealthy or protect its own integrity if an attacker manages to gain JavaScript code execution in the same origin. If attackers were able to perform arbitrary JavaScript commands, any kind of in-program defense would be futile without support from the browser.

Another important limitation to keep in mind is that anomaly detection relies on a benign training set of sufficient size to represent the range of runtime behaviors that could occur. If the training set contains attacks, the resulting invariants might be prone to false negatives. Also, if the training set is too small, false positives could occur. ZigZag can detect some cases of undertraining by setting a lower bound for training size per function. Undertraining, however, is not a limitation specific to ZigZag, but rather a limitation of anomaly detection in general.

With respect to templated JavaScript, while ZigZag can detect templates of previously observed programs by generalizing, entirely new program code can not be enforced without previous training.

In cases where multiple users share programs instrumented by ZigZag, users might have legitimate privacy concerns with respect to sensitive data leaking into invariants generated for enforcement. This can be addressed in large part by avoiding use of the oneOf invariant wholesale, or by detecting whether an invariant applies to data that originates from password fields or other sensitive input and selectively disabling the oneOf invariant. Such an approach would re-

quire integration of taint-tracking for user input. Alternatively, oneOf invariants could be hashed to avoid leaking user data in the enforcement code.

In our prototype we collect data only from important functions, but we sample from users equally to collect all data from those functions. In certain cases this could incur too much overhead on the client, slowing down the performance of the application. To put less burden on individual users, statistical sampling of data over a larger group could be used, similarly to Liblit et al. [32, 33].

We used browsers as platform for learning and enforcement, however, ZigZag could be ported to other environments that use JavaScript. For example, Windows Phone Apps, server-side JavaScript programs as Node.js, or browser extensions that use JavaScript, such as Chrome. For this to be effective, models of anomaly would need to be adjustment.

APPENDIX A

External Dependencies

ZigZag makes use of various external software packages and programming environments that are required to operate the system.

A.1 Daikon v4.6.4

Daikon likely program invariant generator. The program was extended with new invariants and code output. `http://groups.csail.mit.edu/pag/daikon/`.

A.2 Google Closure Tools - Compiler v1016M

Optimizing JavaScript source to source compiler with instrumentation capabilities. `https://developers.google.com/closure/compiler`

A.3 PyICAP v1

An ICAP module interfacing with Squid proxy, it is used to modify requests to collect monitoring data, and responses to add instrumentation code. The program is developed in Python which allows for flexibility in development. `https://github.com/netom/pyicap`

A.4 PyPy 2.2

An alternative implementation of the Python environment. The Just-in-Time (JIT) compiler allows for faster content adoption as compared to regular Python. PyPy is compatible with the

BeautifulSoup and lxml libraries. http://www.pypy.org/

A.5 Squid v3.3.8

Internet caching proxy. The proxy is only used to interface with an ICAP module, all caching functionality is disabled. http://www.squid-cache.org

Bibliography

[1] ADsafe . http://www.adsafe.org.

[2] Squid Internet Object Cache. http://www.squid-cache.org, 2005.

[3] XMLHttpRequest. W3C Working Draft . http://www.w3.org/TR/XMLHttpRequest/, 2012.

[4] Closure Tools - Google Developers , 2013. https://developers.google.com/closure/compiler.

[5] Content Security Policy 1.1, 2013.

[6] Acunetix. Acunetix Web Vulnerability Scanner. `http://www.acunetix.com/`, 2013.

[7] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[8] Adam Barth, Collin Jackson, and John C Mitchell. Securing Frame Communication in Browsers. *Communications of the ACM (CACM)*, 2009.

[9] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[10] Burp Spider. Web Application Security. `http://portswigger.net/burp/spider.html`, 2013.

[11] R. Chugh, J.A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM Sigplan Notices*, volume 44, pages 50–62. ACM, 2009.

[12] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[13] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the World Wide Web Conference (WWW)*, 2010.

[14] Gabriela F Cretu, Angelos Stavrou, Michael E Locasto, Salvatore J Stolfo, and Angelos D Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[15] Douglas Crockford. The application/json media type for JavaScript Object Notation (JSON). 2006. http://json.org/.

[16] Douglas Crockford. JSLint: The JavaScript Code Quality Tool, April 2011. `http://www.jslint.com/`.

[17] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2010.

[18] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[19] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy (Oakland)*, 1996.

[20] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. 1994.

[21] Frank Groeneveld, Ali Mesbah, and Arie van Deursen. Automatic Invariant Detection in Dynamic Web Applications. Technical report, Delft University of Technology, 2010.

[22] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[23] Mario Heiderich, Tilman Frosch, and Thorsten Holz. ICESHIELD: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[24] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *International Conference on World Wide Web (WWW)*, 2003.

[25] Insecure.org. NMap Network Scanner. `http://www.insecure.org/nmap/`, 2013.

[26] International Secure Systems Lab. `http://anubis.iseclab.org`, 2009.

[27] Internet World Stats. Usage and Population Statistics. `http://www.internetworldstats.com/stats.htm`, 2013.

[28] Harold S Javitz and Alfonso Valdes. The SRI IDES Statistical Anomaly Detector. In *IEEE Symposium on Security and Privacy (Oakland)*, 1991.

[29] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[30] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. 2006.

[31] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript Instrumentation in Practice. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[32] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Notices*, 2003.

[33] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable Statistical Bug Isolation. In *ACM SIGPLAN Notices*, 2005.

[34] Sergio Maffeis and Ankur Taly. Language-based Isolation of Untrusted JavaScript. In *IEEE Computer Security Foundations Symposium*, 2009.

[35] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser. In *IEEE Symposium on Security and Privacy (Oakland)*.

[36] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe Active Content in Sanitized JavaScript. Technical report, Google, Inc., 2008.

[37] Nikto. Web Server Scanner. `http://www.cirt.net/nikto2`, 2013.

[38] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 2007.

[39] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive Auto-sanitization in Web Templating Languages using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[40] 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 (Oakland)*, 2010.

[41] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2010.

[42] David Scott and Richard Sharp. Abstracting Application-level Web Security. *International Conference on World Wide Web (WWW)*, 2002.

[43] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[44] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with Content Security Policy. In *International Conference on World Wide Web (WWW)*, 2010.

[45] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.

[46] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2013.

[47] Web Application Attack and Audit Framework. `http://w3af.sourceforge.net/`.

[48] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion DetectionSystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

[49] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.

[50] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *Principles of Programming Languages (POPL)*, 2007.