ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities

Michael Weissbacher
Northeastern University
mw@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

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 as well. 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 paper, 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.

1 Introduction

Most of the over 2 billion Internet users [1] 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 [2, 3, 4, 5], language-based approaches to render the introduction of certain classes of vulnerabilities impossible [6, 7, 8], sandboxing of potentially vulnerable code, and signature- and anomaly-based schemes to detect attacks against legacy programs.

However, despite the large amount of research on 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.

In particular, advances in browser JavaScript engines and the adoption of HTML5 APIs has 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 or other programming mistakes.

`postMessage` is but one potential vector for 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. Also, in contrast to other popular web attack classes such as Cross-Site Scripting (XSS), CSVs represent application logic flaws that are closely tied to the intended behavior of the application and, consequently, can be difficult to identify and defend against in a generic, automated fashion.

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 automatically detect deviations from these models 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 paper 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.

The rest of the paper is organized as follows. In Section 2, we motivate the need for defending against client-side validation vulnerabilities through the introduction of a running example and define our threat model. In Section 3, we present the high-level design of ZigZag. Sections 4 and 5 describe the details of ZigZag's invariant detection and enforcement. We then evaluate a prototype implementation of ZigZag in Section 6. Finally, Sections 7 and 8 discuss related work and conclude the paper.

2 Motivation and Threat Model

To contextualize ZigZag and motivate the problem of client-slide validation 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 same origin policy 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 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 client-side validation attack [9].

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, framing the webmail service, or exploiting a logic vulnerability. 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 pa-

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

Figure 1: Insecure usage of the `postMessage` API in a hypothetical webmail client-side component.

rameters, 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. Due to the nature of CSV vulnerabilities, this attack would go unnoticed for server-side invariant detection systems such as Swaddler [10] as they focus on more traditional web attacks against server-side code. These attacks can either happen on the client alone, where such systems have no visibility, or when server interaction is triggered through exploitation of a CSV vulnerability. In addition, these requests are indistinguishable from benign user interaction. 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.

2.1 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. In particular, we presume the presence of complementary defenses against XSS-based code injection attacks such as Content Security Policy (CSP) [11] or rigorous template auto-sanitization. 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 [12].

3 System Overview

ZigZag is an in-browser anomaly detection system that defends against client-side validation (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.

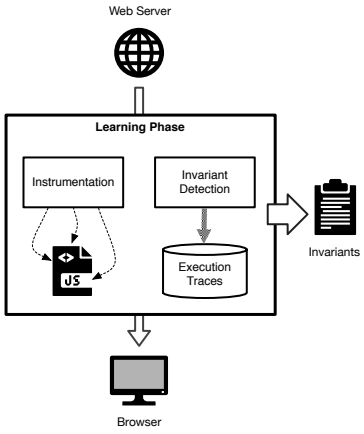
Enforcement phase. In the second phase, the invariants that were learned in the initial phase are used to harden the client-side components of the application. The hardened version of the web application preserves the semantics of the original, but also incorporates runtime checks to enforce that execution 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.

An overview of this system architecture is shown in Figure 2. 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.

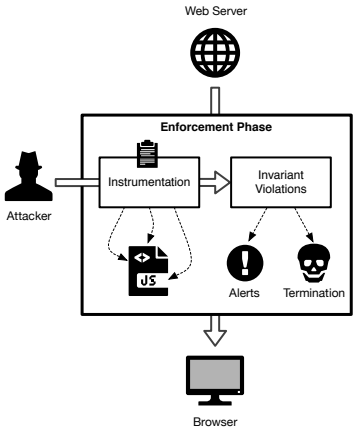
In the following sections, we describe in detail each phase of ZigZag's approach to defending against client-side validation vulnerabilities in web applications.

4 Invariant Detection

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



(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 2: 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.

Data Type	Invariants
All	Types
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 1: Invariants supported by ZigZag.

4.1 Invariant Detection

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 (we define checkpoints in further detail in Section 4.2). 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 that purely static systems lack.

ZigZag uses program execution traces to generate Daikon [13] 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 (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 6.1.

One pitfall of anomaly detection is undertraining. To reduce the impact on our system, we check function coverage before issuing invariants for enforcement. We only allow for enforcement of a particular function after execution traces from four or more training sessions were collected, which was sufficient for the examples we considered. This threshold, however, is configurable and can easily be increased if greater variability is observed during invariant learning.

The full set of invariants supported by ZigZag is shown in Table 1.

```
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: Function instrumentation example.

4.2 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. Since events such as receiving cross-window communication are handled by functions, function entry and exit points are natural program points to analyze input and return data. Consequently, for our prototype we opted to insert checkpoints at function prologues and epilogues.

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. Aside from increased performance, whitelisting functions that are known not to be security-relevant also leads to a reduced risk of false positives.

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 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 programs can potentially modify themselves at runtime, 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 [14].

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 enclos-

ing (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 3a and 3b, respectively.

```
1   __calltrace = function(functionid, codeid, sessionid) {
2       // Enforcement
3       var v0 = arguments.callee.caller.caller.arguments[0];
4       var v1 = ...
5
6       if ( functionid === 0 ) {
7           __assert(typeof(v0) === 'number' && v0 > 5);
8           __assert(typeof(v1) === 'string' && v1 === "x");
9           ...
10      } else if ( functionid === 1 ) {
11          ...
12      }
13      ...
14      return __incCallCounter();
15  }
```

Figure 4: Example of invariant enforcement over a function's input state.

5 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 [15]. 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 4 shows a possible instance of the `calltrace` function, abbreviated for clarity.

5.1 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

```
1   // Server-side JavaScript template
2   var state = {
3       user: {{username}},
4       session: {{sessionid}}
5   };
6
7   // Client-side JavaScript code after template instantiation
8   var state = {
9       user: "UserX",
10      session: 0
11  };
```

Figure 5: Example of a JavaScript template.

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 5.

Due to the cost of instrumentation and the prevalence of this technique, this mix of code and data 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.

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. Objects play a special role as in template instantiation properties can be omitted or ordered non-deterministically. As a result ASTs are not equal in all cases, only similar. Determining whether this

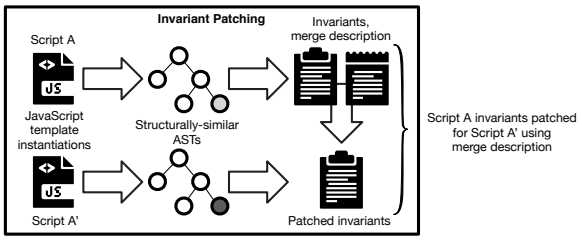


Figure 6: Invariant patching overview. If ZigZag detects that two JavaScript programs are structurally isomorphic aside from constant assignments, a merge description is generated that allows for efficient patching of previously-generated invariants. This scheme allows ZigZag to avoid re-instrumentation of templated JavaScript on each load.

is the case could be performed by pairwise AST equality that ignores constant values in assignments and normalizes objects. 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 that 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. 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 our modified version of the Closure compiler [16] 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.

5.2 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.

5.3 Limitations

ZigZag's goal is to defend against attackers that desire to achieve code execution within an origin, or act on behalf of the victim. 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.

Therefore, we presume (as discussed in Section 2.1) the presence of complementary measures to defend against XSS-based code injection. Examples of such techniques that could be applied today include Content Security Policy (CSP), or any of the number of template auto-sanitization frameworks that prevent code injection in web applications [17, 18, 6].

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. We believe that access to, or the ability to generate, benign training data is a reasonable assumption in most cases. For instance, traces could be generated from end-to-end tests used during application development, or might be collected during early beta testing using a population of well-behaving users. However, in absence of absolute ground truth, solutions to sanitize training data exist. For instance, Cretu et al. present an approach that can sanitize polluted training data sets [12].

If the training set is too small, false positives could occur. To limit the impact of undertraining, we only generate invariants for functions if we have more than four sessions, which we found to be sufficient for the test cases we evaluated. We note that the training threshold is configurable, however, and can easily be increased if greater variability is observed at invariant checkpoints. 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, or by heuristically detecting whether an invariant applies to data that originates from password fields or other sensitive input and selectively disabling the oneOf invariant. Alternatively, oneOf invariants could be hashed to avoid leaking user data in the enforcement code.

6 Evaluation

To evaluate ZigZag, we implemented a prototype of the approach using the proxy deployment scenario. We wrote Squid [19] ICAP modules to interpose on HTTP(S) traffic, and modified the Google Closure compiler [16] to instrument JavaScript code.

```
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     adWindow.postMessage({
16         'topic': 'ads',
17         'action': 'showads',
18         'content': '{JSON␣string}'
19     }, "*");
20 }
```

Figure 7: 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 8: Vulnerable ad network component.

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.

6.1 Synthetic Applications

Webmail service. We evaluated ZigZag on the hypothetical webmail system first introduced in Section 2. 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 7 and 8.

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.

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.content` 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.

URL fragments. Before `postMessage` became a standard for cross-origin communication in the browser, URL fragments were used as a workaround. The URL fragment portion of a URL starts after a hash sign. A distinct difference between URL fragments and the rest of the URL is that changes to the fragment will not trigger a reload of the document. Furthermore, while SOP generally denies iframes of different origin mutual access to resources, the document location can nevertheless be accessed. The combination of these two properties allows for a channel of communication between iframes of different origins.

We evaluated ZigZag on a demo program that communicates via URL fragments. The program expects as input an email address and uses it without proper sanitization in `document.write`. Another iframe could send unexpected data to be written to the DOM. The code is described in Figure 9.

```
1  function getFragment ( ) {
2      return window.location.hash.substring(1);
3  }
4
5  function fetchEmailAddress() {
6      var email = getFragment();
7      document.write("Welcome␣" + email);
8      // ...
9  }
```

Figure 9: Vulnerable fragment handling.

After the training phase, we generated the following invariants for the `getFragment` function.

1. The function is only called by one parent function
2. The return value is an email address
3. The return value is printable

6.2 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]

These applications are representative of different, previously-identified classes of CSV vulnerabilities. In particular, Son et al. [9] examined the prevalence of CSV vulnerabilities in the Alexa Top 10K websites, found 84 examples, and classified them. The aim of this experiment is to demonstrate that the invariants ZigZag generates can prevent exploitation of these known classes of vulnerabilities.

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.

Janrain. A code snippet used by `janrain.com` for user management is vulnerable to a CSV attack. The application checks the format of the string, but does not check the origin of 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

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

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 [9]. As of writing, ten out of the 13 listed sites remain vulnerable, including `wholefoodsmarket.com` and `ladygaga.com`.

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. However, this is not an isolated case. Related work [9] has shown that such a flawed origin check was used by 71 hosts in the top 10,000 websites.

ZigZag generated the following relevant invariants.

1. The function is only invoked from the global scope or as an event handler
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. 43 out of the top 10,000 websites had previously been shown to be exploitable with the same technique. 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 fixed.

Summary. These are four attacks against CSV vulnerabilities representative of the wider population. `postMessage` receivers are used on 2,245 hosts out of the top 10,000 websites. Such code is often included through third-party libraries that can be changed without the knowledge of website owners.

6.3 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.

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.

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

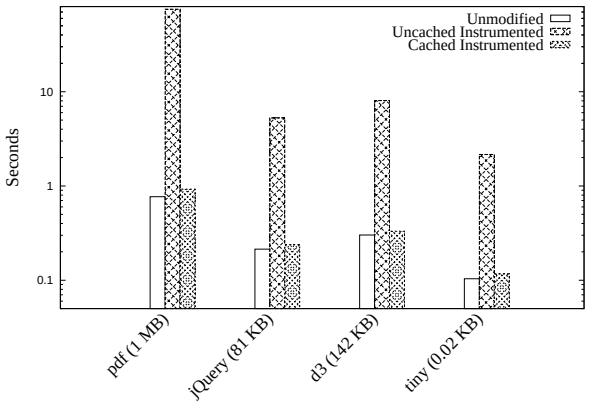


Figure 10: Instrumentation overhead for individual files. While the initial instrumentation can take a significant amount of time for large files, subsequent instrumentations have close to no overhead.

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 10 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. Next, we ran the program in both uninstrumented and instrumented forms. The subject of measurement was 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 2.

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`

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

Table 2: Microbenchmark overhead.

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

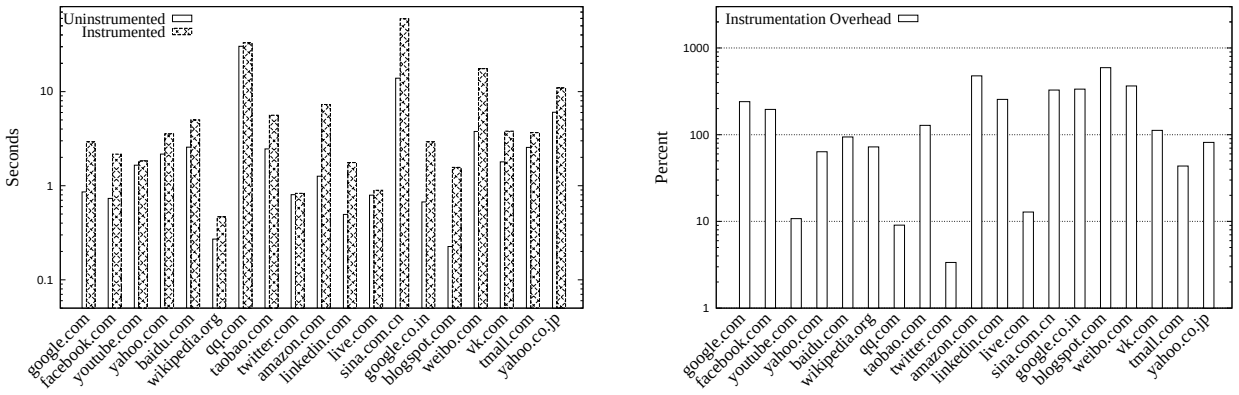
The above invariants represent a tight bound on the allowable data types and values sent across between each origin.

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 the initial request to the `window.load` event, when the DOM completed loading (including all sub-frames).[4] 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

[4]We note, however, that websites can become usable before that event fires.



(a) Absolute load times for uninstrumented and instrumented programs.

(b) Overhead due to instrumentation.

Figure 11: 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.

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 freshly loaded every time.

The performance overhead in absolute and relative terms is depicted in Figure 11. We excluded `hao123.com` from the measurement as it was incompatible with our prototype.[5] 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. That is, in relative numbers, it seems higher than the actual impact on end-users.

We note that we measure the load event, which means that all elements (including ads) have been loaded. Websites typically become usable before that event is fired. Our research prototype could be further optimized to reduce the impact of our technique for performance-critical web applications, for example by porting our ICAP Python code, including parsing libraries, to an ECAP C module. However, generally speaking we believe that trading off some performance for improved security would be acceptable for high assurance web applications and security-conscious users.

6.4 Program Generalization

As discussed in Section 3, 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.

6.5 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 in-

[5] We discovered, as others have before, that `hao123.com` does not interact well with Squid. We attempted to work around the problem by adjusting Squid's configuration as suggested by Internet forum posts, but this did not succeed. Due to time constraints, we did not expend further effort in dealing with this particular site.

teraction 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 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.

7 Related Work

In this section, we discuss ZigZag in the context of related work.

Client-side validation vulnerabilities. CSV vulnerabilities were first highlighted by Saxena et al. [3]. 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 two steps. JavaScript programs are first 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. Second, 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.

Later, the same authors improved FLAX by replacing the dynamic taint analysis component with a dynamic symbolic execution framework [4]. Again, the goal of the static analysis is to find unchecked data flows from inputs to critical sinks. This method provides no completeness and can hence miss vulnerabilities.

The main difference between ZigZag and FLAX is that FLAX focuses on detecting vulnerabilities in applications, while ZigZag is intended to defend unknown vulnerabilities against attacks.

DOM-based XSS. Cross-site scripting (XSS) is often classified as either stored, reflected, or DOM-based XSS [20]. In this last type of XSS, attacks can be performed entirely on the client-side such that no malicious data is ever sent to the server. Programs become vulnerable to such attacks through unsafe handling of DOM properties that are not controlled by the server; examples include URL fragments or the referrer.

As a defense, browser manufacturers employ client-side filtering, where the state-of-the-art is represented by the Chrome XSS Auditor. However, the auditor has shortcomings in regards to DOM-based XSS. Stock et al. [21] have demonstrated filter evasion with a 73% success rate and proposed a filter with runtime taint tracking.

DexterJS [22] rewrites insecure string interpolation in JavaScript programs into safe equivalents to prevent DOM-based XSS. The system executes programs with dynamic taint analysis to identify vulnerable program points and verifies them by generating exploits. DexterJS then infers benign DOM templates to create patches that can mitigate such exploits.

JavaScript code instrumentation. Proxy-based instrumentation frameworks have been proposed before [23, 14]. 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.

Anomaly detection. Anomaly detection has found wide application in security research. For instance, Daikon [13] is a system that can infer likely invariants. The system applies machine learning to make observations at runtime. Daikon supports multiple programming languages, but can also be used over arbitrary data as CSV files. In ZigZag, we extended Daikon with new invariants specific to JavaScript applications for runtime enforcement.

DIDUCE [24] is a tool that instruments Java bytecode and builds hypotheses during execution. When violations to these hypotheses occur, they can either be relaxed or raise an alert. The program can be used to help in tracking down bugs in programs semi-automatically.

ClearView [25] uses a modified version of DAIKON to create patches for high-availability binaries based on learned invariants. The focus of the system is to detect and prevent memory corruption through changing the program code at runtime. However, the embedded monitors do not extend to detecting errors in program logic.

Attacks on the workflow of PHP applications have been addressed by Swaddler [10]. Not all attacks on systems produce requests or, more generally, external be-

havior 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. To model program execution, profiles for each basic block are generated, using univariate and multivariate models. 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 be attacks. The results suggest that anomaly detection on internal application state allows a finer level of attack detection than exclusively analyzing external behavior.

While Swaddler focuses on the server component of web applications, ZigZag characterizes client-side behavior. ZigZag can protect against cross-domain attacks within browsers that Swaddler has no visibility into. Swaddler invokes detection for every basic block, while we use a dynamic level of granularity based on the types of sinks in the program, resulting in a dramatic reduction in enforcement overhead.

Client-side policy enforcement. ICESHIELD [26] is a policy enforcement tool for rules based on manual analysis. By adding JavaScript code before all other content, ICESHIELD is invoked by the browser before other code is executed. Through ECMAScript 5 features, DOM properties are frozen to maintain the integrity of the detection code. ICESHIELD protects users from drive-by downloads and exploit websites. In contrast, ZigZag performs online invariant detection and prevents previously unknown attacks.

ConScript [27] 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.

The dynamic nature of JavaScript renders a purely static approach infeasible. Chugh et al. propose a staged approach [28] where they perform an initial analysis of the program given a list of disallowed flow policies, and then add residual policy enforcement code to program points that dynamically load code. The analysis of dynamically loaded code can be performed at runtime. These policies can enforce integrity and confidentiality properties, where policies are a list of tuples of disallowed flows (from, to).

Content Security Policy (CSP) [29, 11] is a framework for restricting JavaScript execution directly in the browser. CSP can be effective at preventing significant classes of code injection in web applications if applied correctly (e.g., without the use of `unsafe-inline` and `unsafe-eval`) and if appropriate rules are enforced. However, CSP does not defend against general CSV attacks, and therefore we view it and other systems with similar goals as complementary to ZigZag. In particular, CSP could be highly useful to prevent code injection and thereby protect the integrity of ZigZag in the browser.

Web standards. Although Barth et al. [30] 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. [9]. 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.

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

8 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 in the browser is possible. However, these 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 (CSV) vulnerabilities. As these attacks occur on the client, 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. 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 hard-

ens 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 and video portals. 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.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165, the Army Research Office (ARO) under grant W911NF-09-1-0553, the Department of Homeland Security (DHS) under grant 2009-ST-061-CI0001, the National Science Foundation (NSF) under grant CNS-1408632, and SBA Research. We would like to thank the anonymous reviewers for their helpful comments. Finally, we would like to thank the Marshall Plan Foundation for partially supporting this work.

References

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

[2] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," in *IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[3] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2010.

[4] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[5] D. Crockford, "JSLint: The JavaScript Code Quality Tool," April 2011, http://www.jslint.com/.

[6] M. Samuel, P. Saxena, and D. Song, "Context-sensitive Auto-sanitization in Web Templating Languages using Type Qualifiers," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[7] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe Active Content in Sanitized JavaScript," Google, Inc., Tech. Rep., 2008.

[8] S. Maffeis and A. Taly, "Language-based Isolation of Untrusted JavaScript," in *IEEE Computer Security Foundations Symposium*, 2009.

[9] S. Son and V. Shmatikov, "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[10] M. Cova, D. Balzarotti, V. Felmetsger, and G. 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.

[11] "Content Security Policy 1.1," 2013. [Online]. Available: https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

[12] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, "Casting out Demons: Sanitizing Training Data for Anomaly Sensors," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, 2007.

[14] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "JavaScript Instrumentation in Practice," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[15] F. Groeneveld, A. Mesbah, and A. van Deursen, "Automatic Invariant Detection in Dynamic Web Applications," Delft University of Technology, Tech. Rep., 2010.

[16] "Closure Compiler," 2013. [Online]. Available: https://developers.google.com/closure/compiler

[17] "ctemplate - Powerful but simple template language for C++," 2013. [Online]. Available: https://code.google.com/p/ctemplate/

[18] "Handlebars.js: Minimal Templating on Steroids," 2007. [Online]. Available: http://handlebarsjs.com/

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

[20] A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," *Web Application Security Consortium, Articles*, 2005.

[21] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise Client-side Protection against DOM-based Cross-Site Scripting," *USENIX Security Symposium*, 2014.

[22] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-Patching DOM-based XSS At Scale," *Foundations of Software Engineering (FSE)*, 2015.

[23] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript Instrumentation for Browser Security," in *Principles of Programming Languages (POPL)*, 2007.

[24] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *International Conference on Software Engineering (ICSE)*, 2002.

[25] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically Patching Errors in Deployed Software," in *ACM Symposium on Operating Systems Principles (SIGOPS)*, 2009.

[26] M. Heiderich, T. Frosch, and T. Holz, "ICESHIELD: Detection and Mitigation of Malicious Websites with a Frozen DOM," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[27] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[28] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged Information Flow for JavaScript," in *ACM Sigplan Notices*, 2009.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *International Conference on World Wide Web (WWW)*, 2010.

[30] A. Barth, C. Jackson, and J. C. Mitchell, "Securing Frame Communication in Browsers," *Communications of the ACM*, 2009.