JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks

Yinzhi Cao[†], Xiang Pan, Yan Chen, and Jianwei Zhuge[††]
Northwestern University [†]Columbia University [††]Tsinghua University

ABSTRACT

Drive-by download attacks, which exploit vulnerabilities of web browsers to control client computers, have become a major venue for attackers. To detect such attacks, researchers have proposed many approaches such as anomaly-based [22, 23] and vulnerability-based [44, 50] detections. However, anomaly-based approaches are vulnerable to data pollution, and existing vulnerability-based approaches cannot accurately describe the vulnerability condition of all the drive-by download attacks.

In this paper, we propose a vulnerability-based approach, namely JShield, which uses novel opcode vulnerability signature, a deterministic finite automaton (DFA) with a variable pool at opcode level, to match drive-by download vulnerabilities. We investigate all the JavaScript engine vulnerabilities of web browsers from 2009 to 2014, as well as those of portable document files (PDF) readers from 2007 to 2014. JShield is able to match all of those vulnerabilities; furthermore, the overall evaluation shows that JShield is so lightweight that it only adds 2.39 percent of overhead to original execution as the median among top 500 Alexa web sites.

1. INTRODUCTION

In the past few years, drive-by download attacks exploiting browser vulnerabilities have become a major venue for attackers to control benign computers including those of reputable companies. For example, in February 2013, both Facebook and Apple confirm being hit in "watering hole attack" [4], a variance of drive-by download attack. In such an attack, the attacker compromises a web site commonly visited by victims, injects drive-by download attacks, and then waits for victims to come, just as a predator sitting at a water hole in a desert for prey.

To defeat these severe attacks, many anomaly-based approaches [21–23, 46], which tune the detection engine based on attacker-generated exploits and benign web sites, have been proposed and significantly improved state of the art. However, although anomaly-based approaches can detect unknown attacks, they could succumb in adversarial environments. Inspired by the data pollution on polymorphic worm detection [41], we designed and implemented polluted drive-by download attacks, which could significantly reduce the detection rates of those anomaly-based detectors.

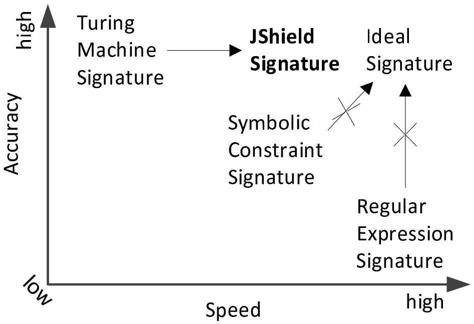

Figure 1: Signature Comparison.

To countermeasure such data pollution, we turn to vulnerability-based approaches. In this emerging direction, although the precise description of a network protocol layer vulnerability signature has already been used in the network intrusion detection systems (NIDS) [32, 54], its application on Web drive-by download attack detection is very limited. In BrowserShield [44], Reis et al. rewrite JavaScript codes and enforce stateless security policies (signatures), such as checking the parameter length of a function call. The other work by Song et al. [50] matches the inter-module communication with vulnerability signatures. However, neither of these techniques can precisely represent the exact vulnerability conditions of drive-by download attacks. More specifically, the rules in BrowserShield do not have stateful data structures to record control or data flow, and the signatures in the work by Song et al. lack sufficient information for control flow at the inter-module level.

In summary, we believe that an approach detecting drive-by download attacks should address the following **challenges:**

- **Signatures for Stateful Intra-module Vulnerabilities.** The signature for drive-by download attacks targeting stateful intra-module vulnerabilities should contain both control and data flows during matching.
- **Resilience to Polluted Data.** The system should be resilient to polluted training and testing samples provided by the attackers.
- **High Performance.** The runtime detection system should have an acceptable overhead to the pure execution of web pages.

In this paper, we propose JShield, vulnerability-based detection of drive-by download attacks. JShield has been adopted by Huawei, the world's largest telecommunication company. We position JShield opcode signatures in Figure 1 with other known vulnerability signatures. Neither symbolic constraint signature nor regular expression signature can represent a drive-by download vulnerability that takes Javascript, a Turing complete language, as input, since they do not have loops. Meanwhile, a traditional Turing machine signature is so complex that it requires a signature as large as a whole browser.

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12$15.00
http://dx.doi.org/10.1145/2664243.2664256.

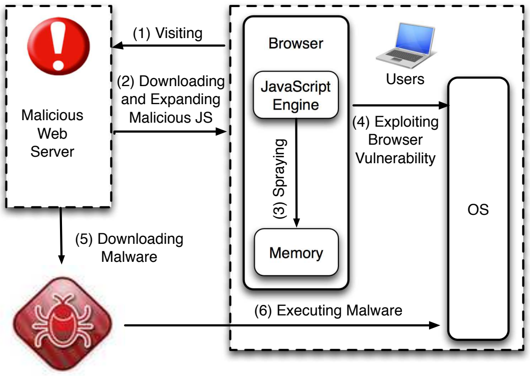

Figure 2: Six Steps of a Drive-by Download Attack.

Thus, we abstract lower level Turing machine to a higher level opcode Turing machine and design detection format of our opcode signature. To further make it scalable, we also use regular expression (filter format of opcode signature) to filter a large number of benign traffic.

The current signature generation process involves some manual effort. However, we believe that the amount of manual work is small due to the small number (approximately 10) of vulnerabilities each year. Actually, even for a large amount of signatures, most network intrusion detection/prevention system (IDS/IPS) vendors, such as Snort, Cisco and Juniper, all generate these signatures manually [5, 13].

Besides being the $first$ to design polluted drive-by download attacks and evaluate their effectiveness on the state-of-the-art anomaly-based approaches, we make the following contributions in this paper:

- **Stateful Drive-by Download Vulnerability Signatures.** Our vulnerability signature is a deterministic finite automaton (DFA) with a variable pool recording both control and data flows to detect stateful intra-module vulnerabilities of drive-by download attacks.
- **Vulnerability Signature at the Opcode Level.** A vulnerability signature at the opcode level can precisely describe a given vulnerability.
- **Fast Two-stage Matching.** We design a two-stage matching process to speed up the detection. The filtering stage adopts fast regular expression matching for a given test sample, and then if the sample is captured at the filtering stage, it is subject to a further matching with opcode signature.

To evaluate JShield, we investigate the vulnerability coverage of JShield and find that JShield is able to handle all the recent JavaScript engine vulnerabilities of web browser and portable document files (PDF) reader. The overall evaluation shows that JShield has introduced little performance overhead to pure execution. The average overhead of top 500 Alexa web sites is 9.42% and the median is 2.39%.

2. THREAT MODEL

The paper focuses on drive-by download attacks consisting of two major stages: pre-exploit stage and post-exploit stage, which can further divide into six steps as shown in Figure 2.

At the pre-exploit stage, a benign user is lured to visit a malicious web site (step one). Then, malicious contents are downloaded, and malicious JavaScript codes, possibly obfuscated by eval, setTimeout, and DOM events, get expanded by execution (step two). During execution, some malicious JavaScripts also fill the heap with shellcodes and nop sleds to overcome address space layout randomization and facilitate attacks (step three).

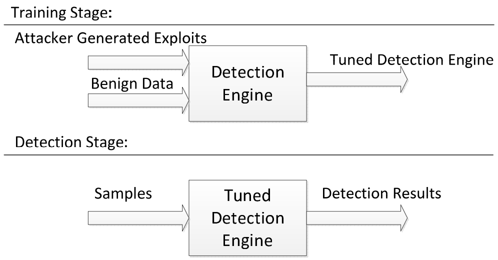

Figure 3: Anomaly-based Approaches to Detect Drive-by Download Attacks.

After all the preparation, the malicious JavaScript exploits a certain vulnerability (step four), and thus the injected shellcode takes control of the browser to download malware (step five), which is subsequently executed on the victim machine (step six).

To distinguish our threat model from others, we also mention other attacks below - these however are *out of scope* of the paper.

Attacks without any JavaScript Interaction. Similar to existing works like Zozzle [23], we only focus on the JavaScript part of a drive-by download attack. If an attack is purely related to HTML parser, CSS parser or font processing, neither Zozzle with abstract syntax tree (AST) features nor JShield with opcode vulnerability signatures can detect the attack. Instead, due to lack of full featured obfuscation techniques, this type of attacks should be prevented by traditional NIDS.

Other Web-based Attacks. Other web-based attacks, such as cross-site scripting (XSS) attacks, cross-site request forgery (CSRF), and so on are out of scope of the paper. People should rely on existing defense mechanisms [19, 52] for those attacks.

3. POLLUTED DRIVE-BY DOWNLOAD ATTACKS

Anomaly-based approaches [18, 22, 23, 46] first train a detection engine based on exploits generated by attackers as well as benign samples collected from the Internet, and then perform detection by the tuned engine. An overview architecture is shown in Figure 3. Normally, a machine learning engine is deployed in the training stage, which extracts malicious and benign features from training data and then trains the detection engine.

In this section, we discuss the efficacy of anomaly-based detection in adversarial environment. We understand that the advantage of anomaly-based detections is that they can detect unknown attacks, however in this paper we only focus on the behavior of anomaly-based detection in adversarial environment.

Generally, an anomaly-based detection could be evaded in two ways, namely, polluting attacker generated exploits in training stage and altering malicious samples in detection stage. We introduce data pollution in detection stage first because of its effectiveness. For data pollution, we use naive Bayes engine adopted by Zozzle [23] as an example to show how to evade anomaly-based approaches.

3.1 Polluting Samples at Detection Stage

An attacker can alter malicious samples by injecting benign features (BF), thus increasing the probability of classifying those samples as benign and evading naive Bayes engine. Intuitively, the benign features decrease the anomaly by reducing the significance

Table 1: Zozzle's Detection Rate.

	Original Rate	Rate after Pollution at Detection Stage
True Positive	93.1%	36.7%
False Positive	0.5%	0.5%

of the malicious features statistically. Here is the detailed reason. Assume there is a sample with malicious feature (MF). According to the definition, $P(M|MF)$, the probability of malice given the existence of one malicious feature, is larger than 0.5. Now let us assume that one benign feature (BF) is introduced to that file. Given that MF and BF are independently distributed in naive Bayes, we have Equation 1.

$$\begin{aligned} P(M|MF,BF) &= \frac{P(MF,BF|M) * P(M)}{P(MF,BF)} \\ &= \frac{P(MF|M) * P(BF|M) * P(M)}{p(MF) * P(BF)} \\ &= \frac{P(BF|M)}{P(BF)} * P(M|MF) \end{aligned} \tag{1}$$

Since fewer BFs exist in malicious files, $\frac{P(BF|M)}{P(BF)} < 1$. Thus, Equation 1 shows that the existence of one benign feature reduces the probability of the sample's malice. If enough benign features are introduced, $P(M|MF, BF, BF1, BF2...)$, the probability of the sample's malice, will be eventually less than 0.5, resulting in a mislead of the Bayes classifier.

3.2 Real-world Experiments

We use Zozzle [23], the most recent and successful machine learning detection, as a case study to show how to evade an anomaly-based approach, however, our discussion is not restricted to Zozzle. Since Zozzle is not open source, we strictly followed what has been described in the paper [23], implemented our version of Zozzle, and reproduced comparable detection rate for unpolluted samples as the one reported by Zozzle.

The testing data set is from Huawei, which contains unclassified malicious JavaScript codes collected from their customer reporting, other anti-virus software reporting, etc. After filtering all the landing pages, we collect 880 malicious samples (Zozzle adopts 919 JavaScript malware samples [23]). Meanwhile, Top 5,000 Alexa web sites [3] are used as benign samples during training. After manual and automatic selection documented in Zozzle, we collect 300 benign and malicious features.

Since an attacker cannot acquire our benign features used in the system, we collect all the common features among Top 5,000 Alexa web sites and add them to malicious samples. The size of each malicious sample increases 45% on average, *i.e.*, 396KB. As shown in Table 1, pollution at the detection stage decreases the overall accuracy to 36.7%.

In addition to Zozzle, we also evaluated five popular anti-virus programs[1] selected from an anti-virus software review web site [1], which are: 1) Avira Antivirus Premium 2013, 2) AVG Internet Security 2013, 3) Kaspersky Internet Security 2012, 4) Norton Internet Security 2013 and 5) Trend Micro Titanium Internet Security 2013. The results are shown in Table 2. Before data pollution, the detection rates of anti-virus software except AV4 are all above 85%. However, after data pollution, the detection rates of them are all below 4%. Therefore, existing anti-virus software, which are blackbox to us and some of which belong to regular expression based approach, are not robust to data pollution either.

4. OVERVIEW

Table 2: The detection rate of five anti-virus programs before and after data pollution.

Anti-virus	Original Rate	Rate After Pollution
AV1	98.00%	0.58%
AV2	89.33%	3.58%
AV3	92.41%	2.00%
AV4	20.67%	0.08%
AV5	87.58%	2.00%

As discussed in previous section, attacker generated samples adopted by anomaly-based detections could be polluted, and thus we resort to vulnerability-based detections. In this section, we first present two types of deployment for JShield. Then, we model drive-by download vulnerabilities based on their control and data flows.

4.1 Deployment

JShield is a dynamic vulnerability signature based approach to de-obfuscate and detect drive-by download attacks. There are two major types of deployment for JShield: 1) at the Web Application Firewalls (WAF) or Web IDS/IPS and 2) at Web malware scanning services. For the former, JShield is deployed as a component of anti-virus software, or as a detection engine at Internet Service Providers (ISP) gateways. For example, Huawei deploys our system in their intelligent cloud, inspecting potential malicious traffic from their switches and routers. On the other hand, JShield can also be deployed on the sever side as a Web malware scanning service, by search engines such as Google and Bing, or by a security company for online web malware scanning.

Compared to an anomaly based approach like Zozzle [23] which needs to retrain the detection engine to accommodate new drive-by download exploits, JShield only needs to update its vulnerability signature database for new drive-by download vulnerabilities. Due to that fact that the number of vulnerabilities is always much less than the amount of exploits, the update overhead of JShield will be small.

4.2 Vulnerability Modeling

Traditionally, there are three types of signatures in literature [20]: Turing machine signature, symbolic constraint signature, and regular expression signature. We look into those signatures in the context of a drive-by download attack where the vulnerable program is a browser and the input exploit is a JavaScript program. The Turing machine signature generation process from Brumley et al. [20] would output a signature as large as a browser, hence making it unusable. On the other hand, neither symbolic constraint signature nor regular expression can represent a complex language like JavaScript which have loops. Thus, for preciseness, we need to have a new signature between a Turing machine signature generated by Brumley et al. and a symbolic constraint signature. For matching speed, we first use regular expression signature to filter a majority of benign JavaScript and then detect malicious JavaScript using the precise detection format of our opcode signature. Since regular expression signature is well known, we focus on the detection format of our opcode signature.

To model a vulnerability, we define a vulnerability condition as a function that takes a certain path of a program's control and data flow graph, and output whether the path is exploitable or not. Formally, given $c \in C$, where C is all possible paths of the program's control flow graph, and $d \in D$, where D is all possible paths of the program's data flow graph, a vulnerability condition k is a function,

$k : C \times D \rightarrow \{Safe, Exploit\}$

In order to match a certain vulnerability, its corresponding vulnerability signature need to match both the path in the control flow graph and the one in the data flow graph. In the context of a drive-

[1] This order does not correspond to the order of AV1 to AV5 in Table 2.

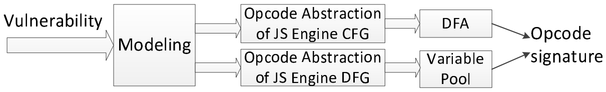
Vulnerability
Modeling
Opcode Abstraction of JS Engine CFG
Opcode Abstraction of JS Engine DFG
DFA
Variable Pool
Opcode signature

Figure 4: Vulnerability Modeling.

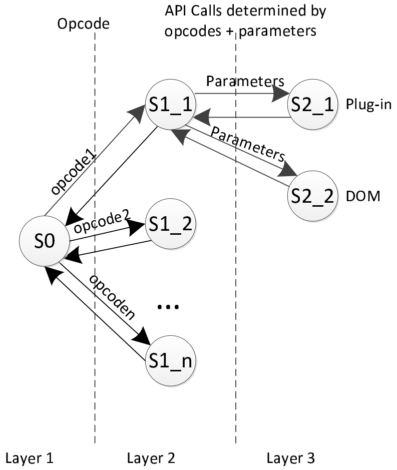

Figure 5: Simplified Opcode Level Control Flow Graph of JavaScript Interpreter.

by download attack where the vulnerability exist in a JavaScript engine or a plugin, our observation is that control and data flow graphs of the opcode input is an abstraction of the control and data flow graphs of the native JavaScipt engine. Thus, as shown in Figure 4, we can model the underline vulnerability in the JavaScript engine or plugin by abstracting the control and data flow to the opcode level.

To study the *control flow*, we investigate the source code of several JavaScript engines [14, 17, 39], and find that the main body of a JavaScript interpreter switches to different branches of codes based on the input opcode through a code pattern similar to *select input case opcode*. Then, different opcodes and parameters determine subsequent API calls to external objects, plugin or DOM. Therefore, we form the control flow graph (CFG) of JavaScript engine into an opcode driven three-layer structure as shown in Figure 5. The CFG will shift based on the input opcode sequence. In other words, opcode CFG is built upon the original application CFG.

Next, we categorize vulnerabilities into two types: JavaScript engine vulnerability (including a web browser JavaScript engine and/or a plugin JavaScript engine such as Adobe Reader JavaScript engine) and external JavaScript engine vulnerability. We will explain them respectively.

Let us assume that the control flow graph of a JavaScript engine vulnerability condition is triggered by travelling through S1_1 and S1_2 of Figure 5. Since there is only one path to travel to S1_1 and/or S1_2, which is to offer the corresponding opcode, the opcode level signature represents this vulnerability.

For an external JavaScript engine vulnerability, the API calls to those components such as plugin and DOM are determined by opcode sequences and parameters. Song et al. [50] show that the inter-module communication can represent a vulnerability and thus our opcode signature can achieve the same functionality.

Therefore, we propose an opcode level deterministic finite automaton (further explained in Section 5) to match the opcode control flow, an abstraction of the JavaScript engine control flow, in a vulnerability condition.

To study the *data flow*, JShield needs to record additional states related to the vulnerability. Therefore, we propose a variable pool (further explained in Section 5) to match the opcode data flow, an abstraction of JavaScript engine data flow, in a vulnerability condition.

```
switch (opcode) {
  case get_by_id:
    // (1) look up the prototype chain
    // (2) invoke getter method
    // (3) move results to register r0
         break;
  case put_by_id:
    // (4) move null to the prototype
    break;
}
```

Figure 6: Pseudo JavaScript Engine Codes for CVE-2009-1833.

```
var obj = new Object();
obj.__proto__.__defineGetter__("a"
                 , function () {
            this.__proto__ = null;
            gc();
            return 0;
            });
obj.a;
```

Figure 7: Example I: CVE-2009-1833: Malicious JavaScript Exploit that can Trigger a Mozilla Firefox JavaScript Engine Vulnerability.

Now, we illustrate the point by a concrete running example from CVE database.

Example I. In Figure 6, we show how to trigger CVE-2009-1833 in the pseudo code of JavaScript engine. The vulnerability is triggered by two conditions: (i) looking up through prototype chain to get a getter function, and (ii) setting the prototype itself to be null inside the getter function.

When we abstract the JavaScript engine vulnerability to the opcode level and take a look at a concrete exploit example triggering vulnerability in Figure 7 and Figure 8, we find that the opcode level CFG is an abstraction of the underline level JavaScript engine CFG. S1_1 in Figure 5 is visited by *get_by_id* and S1_2 is visited by *put_by_id*. For the data flow, to match CVE-2009-1833, JShield needs to remember the memory address of the prototype.

In sum, the JShield signature needs to match both the control flow graph and the data flow graph of opcode sequence of a JavaScript code, an abstraction for the control and data flow graph of the underline JavaScript engine, for a drive-by download vulnerability condition as shown in Figure 4.

5. OPCODE VULNERABILITY SIGNATURE

As discussed in Section 4.2, a successful opcode signature needs to match both the control flow and the data flow of a vulnerability condition. In this section, we introduce the detailed design of opcode signature matching drive-by download vulnerabilities. To speed up the matching process, two types of opcode signature, the detection format and the filter format, are described here by their definition, structure and matching process. In the end, we present the robustness of opcode vulnerability signature to polymorphic attacks.

In the current version of JShield, all the opcode vulnerability signatures are generated manually for each vulnerability. However, we believe that the amount of involved manual work is small due to the small number ($<$100) of vulnerabilities each year. Actually, even for a large amount of signatures in an intrusion detection system

```
[ 199] get_by_id    r0, r1, a(@id1)
[   0] enter
[   1] convert_this r-7
[   3] mov        r0, r-7
[   6] put_by_id  r0, __proto__(@id0), Null(@k0)
[  15] ret Int32: 0(@k1)
```

Figure 8: Generated Opcode Snippet when Executing JavaScript Codes in Figure 7.

like Snort, those signatures are all generated manually [13]. Further, we also expect future improvement can automate vulnerability signature generation. As an analogy, Shield [54] proposes protocol level vulnerability signatures, and then Brumley et al. [20] propose their automatic generation.

5.1 Definition

Opcode signature is a signature to match an opcode sequence, an instruction set generated by a JavaScript interpreter for efficient execution. For example, opcodes in Figure 8 are the results of transmitting JavaScript code in Figure 7. Op code signature has two formats: a detection format used for matching and a filter format used for fast filtering.

We first formalize detection format of opcode signature as a deterministic finite automaton (DFA) plus a variable pool in Definition 1.

DEFINITION 1. *We define detection format of opcode signature as a 10-tuple (Q, Σ, P, V, g, G, f, q_0, p_0, F), where*

- *Σ is finite set of input symbols, P is finite set of variables, V are the value set of P, and Q is finite set of states.*
- *g, is a function $P \to V$.*
- *G is the set of all possible g.*
- *f is a transition function, $Q \times \Sigma \times G \to Q \times G$.*
- *$q_0 \in Q$ is a start state, $p_0 \subseteq G$ is an initial variable pool, and $F \subseteq Q$.*

For input $a_i \in \Sigma$ and a variable pool $p \subseteq G$, the next state of the automaton obeys the following conditions:

1. *$r_0 = q_0, p = p_0$.*
2. *$< r_{i+1}, p > = f(r_i, a_i, p)$, for i = 0,..., n-1.*
3. *$r_n \in F$.*

In Definition 2, we formalize the filter format[2] of the opcode vulnerability signature as a regular expression.

DEFINITION 2. *We define the filter format of opcode signature as a 5-tuple (Q, Σ, f, q_0, F), where*

- *Σ is finite set of input symbols, and Q is finite set of states.*
- *f is a transition function, $Q \times \Sigma \to Q$.*
- *$q_0 \in Q$ is a start state, and $F \subseteq Q$.*

For input $a_i \in \Sigma$, the next state of the automaton obeys the following conditions:

1. *$r_0 = q_0$.*
2. *$r_{i+1} = f(r_i, a_i)$, for i = 0,..., n-1.*
3. *$r_n \in F$.*

5.2 Structure

We introduce the structures of the detection and filter format of opcode signatures in this section.

Detection Format:

#	Method	Opcode	Condition	Action	Next
(1)	match	get_by_id	isFromProtoChain()	x=proto	(2)
	default			quit	N/A
(2)	match	enter	true	i=0	(3)
	default			quit	N/A
(3)	match	enter	true	i=i+1	(3)
	match	ret	i==0	quit	N/A
	match	ret	i>0	i=i-1	(3)
	match	put_by_id	x==dst & src==null	report	N/A
	default			jmp	(3)

Filter Format:

```
get_by_id enter .* put_by_id
```

Figure 9: Opcode Signature for CVE-2009-1833.

5.2.1 Detection Format

The detection format of an opcode signature, as shown in Figure 9, can be formalized into the following three concepts: clause, sentence, and signature.

A *clause* in an opcode signature consists of five fields: “method”, “opcode”, “condition”, “action”, and “next”. The “method” field specifies what to be taken in this clause, where two methods are currently defined: “match” and “default”. “Match” means to match the opcode, and “default” means that the default actions should be taken if no matches are found in other clauses. Then if both “opcode” field matches the input opcode and the expression in “condition” field is true, the action in the “action” field will be taken and current state will be transferred to the number in “next” field, which represents the sentence number that will be explained right after this paragraph.

Multiple clauses plus an index number together build *a sentence*, a state in automaton. The number is used to differentiate one sentence from the others. The clauses in one sentence are in sequence, which means if JShield finds the first match, the remaining ones will be skipped. If no matches are found, the action corresponded with the “default” clause will be taken.

A *signature* consists of multiple sentences. During matching, the automaton will transfer from one sentence to anther based on the matching results.

5.2.2 Filter Format

The filter format of an opcode signature, as shown in Figure 9, can be formalized as a regular expression, which takes a series of opcodes as input. For detection and filter format of opcode signature, the following statement holds: “Each detection format of an opcode signature has a corresponding filter format of that opcode signature”.

Here is the reason. Given a detection format of an opcode signature, for each sentence, we extract all *Opcode* fields and align them into a unit by bracket symbol of regular expression. Then, by following the jmp operations in *Action* field, we align the units into a regular expression. If a self loop is recognized, a symbol $*$ is introduced. The end of the regular expression is one or multiple opcodes in bracket that leads to the vulnerability.

5.2.3 Data Structure of Both Formats

The filter format is simply stored as a regular expression. To speed up matching process, we construct a reverse index for the detection format of opcode signatures by the opcode field. Suppose we have two signatures: Sig1 and Sig2. Each signature has two sentences. Each sentence has two clauses. Under each opcode, both two signatures exist. Under each signature, both two sentences exist. Under each sentence, only the clause with that opcode is

[2] Unless specified, opcode signature refers to the detection format of opcode signature. The filter format refers to the filter format of opcode signature.

Algorithm 1 Matching Detection Format

```
$State \leftarrow Starting\_State$
for Input_Opcode in All_Opcodes do
    for Signature in Pool[Input_Opcode] do
        $Sentence \Leftarrow Signature[State]$
        $Clause \Leftarrow Sentence.Clause$
        if $equal(Method, Match)$ then
            if $(IsAllTrue(Clause.Conditions)$ then
                Take Actions
                $Signature.State \Leftarrow New\_State$
                Break
            end if
        else[$equal(Method, Default)$]
            Default Actions
            $Signature.State \Leftarrow Default\_State$
            Break
        end if
    end for
end for
```

present. Other clauses of that sentence are under other opcodes.

5.3 Generating Opcode Vulnerability Signature

We generate the opcode vulnerability signatures semi-automatically with the following three steps.

1. Based on the semantics of the vulnerability (*e.g.*, from the CVE description or vulnerability patches), we locate the opcodes that are involved in the vulnerability. We create a DFA with each involved opcode being a node (state).
2. From the data flow part, we extract the critical data structure involved in the vulnerability related to each opcode operation and define a variable in the variable pool of "Action" field in opcode signature.
3. We combine the DFA and the variable pool together by introducing each variable to the "Condition" field of opcode signature based on the data flow connection between opcodes.

Again, we use CVE-2009-1833 in Figure 7 as an example. We first automatically generate control and data flow [28]. Then, manual work is involved to find out the semantics of the vulnerability, *e.g.*, for CVE-2009-1833, line 3 and line 7 together cause the vulnerability. From the control flow graph part, the sequence of three opcodes (get_by_id, enter, and put_by_id) will lead to the vulnerability condition. On contrary, another sequence of three opcodes (get_by_id, enter, and ret) will lead to a safe state. Therefore, we create a detection format of our opcode vulnerability signature with three states in DFA as shown in Figure 9, and meanwhile, we use a counter i to record the number of opcode enter and ret. Next, from data flow part, we find that line 3 and line 7 are connected by the memory address of the prototype, and therefore, we use x in variable pool to record that data. In the end, we combine the DFA and the variable pool to the detection format of our opcode vulnerability signature, and follow steps discussed in Section 5.2.2 to generate the filter format of opcode vulnerability signature.

5.4 Matching Opcode Vulnerability Signature

The matching process of opcode signatures can be divided into two parts: pre-matching by the filter format of opcode and matching by the detection format of opcode signature.

At the filtering stage, we match the opcode sequence outputted from de-obfuscation engine with the filter format of opcode signature. If a sequence of opcodes does not match with the filter format of opcode signature, we drop it off because it will not match with the detection format of that opcode signature either. By filtering a large amount of unmatched samples using fast regular expression operation, we can accelerate the total matching process.

Sample One:

```
this.__defineGetter__("x", function (x){
            'foo'.replace(/0/g,[1].push)
          });
for (let y in [,,,])
   for (let y in [,,,])
       x = x;
```

Sample Two:

```
while (true) {
  Array.prototype.push(0);
}
```

Figure 10: Example II: Different Samples that Trigger CVE-2009-0353 (Extracted and Simplified from CVE Library).

After the filtering stage, we match the opcode sequence with detection format. The pseudo-code of the matching algorithm is shown in Algorithm 1. Given one opcode as an input, the matching algorithm goes over every signature with that opcode. For each signature, JShield directly fetches the corresponding clause that belongs to the sentence of the current state because JShield has already indexed all the signatures by opcodes. Then, JShield checks whether the conditions are satisfied, and accordingly takes actions. The complexity for this process is O(Maximum number of Signatures per Opcode × Number of Opcodes).

5.5 Robustness to Polymorphic Attacks

A CVE-2009-0353 example in Figure 10, which is triggered when repeating push operation of an array exceeding the memory limit, shows that JShield reduces polymorphic attacks. Instead of reporting an out-of-memory error, illegal memory address will be overwritten.

In Figure 10, we show two different snippets of JavaScript triggering CVE-2009-0353, which fire push operation in two different ways at the JavaScript level. However, at the opcode level, both need to call opcode *get_by_id* first to get *push* method, and then repeat using opcode *call* (only one *call* is shown in the figure). After generating the call graph for two JavaScript engines [14, 16] by doxygen [7] and thus examining functions that calls *push* method, we find that the only way of calling *push* method is through the opcode *call*. In other words, the opcode signature maps to the vulnerability.

For polymorphic attacks, we show that the number of polymorphism reduces at the opcode level because (i) one opcode signature maps to multiple source code representations of that vulnerability; and (ii) one source code representation of a vulnerability maps to one specific opcode signature given an opcode instruction set.

The reasons are as follows. Assume a vulnerability is represented by an opcode signature. We follow the state transition and get opcode sequence as follows: op_code1, op_code2, op_code3 and so on. Each opcode can be included in multiple JavaScript statement. For example, *op_call* can be triggered by direct function call and *getter* property of an object. Similarly, *op_jmp* can be triggered by *while* loop, *for* loop, and so on. We will choose different JavaScript statement and write corresponding JavaScript source code. To the opposite, if we have a source code representation of a vulnerability and the opcode instruction set is fixed, we can feed

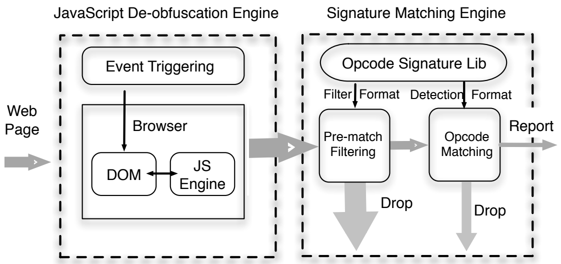

Figure 11: System Architecture.

the source code into the interpreter with the opcode instruction set. One unique opcode sequence is outputted from the interpreter.

6. SYSTEM ARCHITECTURE

Figure 11 shows the overall architecture of JShield, which consists of two main engines, a JavaScript de-obfuscation engine and a signature matching engine. The former takes a web page as input, de-obfuscates JavaScripts, and then outputs the corresponding opcode sequence; the latter takes an opcode sequence as input, matches the sequence with opcode signature, and finally gives a report about whether the incoming web page is malicious .

The detailed process is as follows. When a web page is fed to a JavaScript de-obfuscation engine of JShield, it is executed on a real browser with event triggering module, which mimics user's behaviors to trigger all the DOM events. If the web page contains PDF, JShield adopts MPScan [35] to hook an Adobe Reader and output all the opcode sequences. After de-obfuscation, the signature matching engine first filters opcode sequences outputted from JavaScript de-obfuscation engine. For the opcode sequences not filtered out, JShield further matches them with detection format of opcode vulnerability signatures.

7. IMPLEMENTATION

We use WebKit r101347 together with Qt 4.8.1 on Linux to implement JShield. Web pages are directly fed into a modified version of WebKit that is integrated with event triggering engine. External libraries and virtual functions are loaded into WebKit through JavaScript files. Then opcodes are generated and fed into the opcode matching engine.

De-obfuscation. We introduce the de-obfuscation engine of JShield, which increases the total amount of inspected source code written by attackers, from two aspects: events recording, and event triggering.

We first modify class Document to make every document maintain a queue, which records all the registered event listeners. Then we modify function addEventListener in Node.cpp file. We choose to modify this function in order to make sure that all listeners would be registered, because this function is called every time a listener is added. So when an event listener is trying to register itself on a node, the node will call its method addEventListener(), in which the node adds the listener in the queue maintained by the node's Document object. The modified addEventListener() function also determines whether to call eventTriggering() function to trigger the new registered event listener right away, based on whether the onload event listener is invoked.

Then, we add two functions: void trigerAllWindowEvents() in DOMWindow.cpp file, used to trigger all the event listeners registered on Window object, and void eventTriggering() in Document.cpp file, used to iterate and trigger all its children nodes recorded in the queue. Next, we modify the function void dispatchWindowLoadEvent() in Document.cpp to call trigerAllWindowEvents() and eventTriggering() functions and trigger all the event listeners registered so far.

Signature Matching Engine. We first extract opcodes from WebKit engine of which the interpretation mode for JavaScript is enabled. All the opcodes are interpreted by *privateExecute* function in *Interpreter.cpp* at JavaScriptCore of WebKit. We extract all the register address together with opcode names during the interpretation and feed them into the opcode matching engine.

Next, we use the standard regular expression library [12] in C++ to match the filter format of opcode signature, which is stored as a string. After filtering, the detection format of opcode signatures is stored as a *map* container of STL [15], which uses a red-black tree structure. All the signatures are indexed by opcodes. The data domain of opcode signatures are implemented by a perl interpreter in C [8].

8. EVALUATION

We first introduce our methodology in Section 8.1. Next, we evaluate vulnerability coverage in Section 8.2 and robustness to data pollution in Section 8.3. In the end, We evaluate the performance of JShield including *JavaScript obfuscation engine*, *signature matching engine*, and the overall system in Section 8.4.

8.1 Methodology

To evaluate JShield, we obtain JavaScript engine vulnerabilities and plugin vulnerabilities from CVE database, the details of which can be found in Section 8.2. We adopt three metrics to evaluate signature matching:

- *Vulnerability Coverage Rate.* Vulnerability coverage rate is defined as the number of vulnerabilities covered by a certain approach divided by the number of all the vulnerabilities.
- *Robustness to Data Pollution.* We evaluate the robustness to data pollution by detecting attacks injected with benign features and attack variants using the same vulnerability.
- *Performance.* We measure the latency caused by the event triggering process, the signature matching process, and the overall JShield.

8.2 Vulnerability Coverage

Two sets of vulnerabilities are evaluated on JShield, which are JavaScript engine vulnerabilities and plug-in vulnerabilities. JShield contributes on detecting JavaScript engine vulnerabilities, but for plug-in vulnerabilities, we are on par with existing works [44, 50].

8.2.1 Data Source

We evaluate vulnerability coverage of JShield on those vulnerabilities in national vulnerability database (NVD) [11]. (i) To acquire JavaScript engine vulnerabilities of web browsers, we search the keyword "JavaScript Engine" at the search engine provide by NVD, and examine the "References to Advisories, Solutions, and Tools" part of all the results. Each "External Source" will be examined. We pay special attention to the exploiting codes posted by each external source. (ii) To acquire JavaScript engine vulnerabilities of pdf readers, we search the keyword "JavaScript reader" and "JavaScript pdf" at the search engine provide by NVD. The same procedure for a JavaScript engine vulnerability will be applied to this type of vulnerability. (iii) To acquire inter-module plugin vulnerabilities, we obtain information from Song et al [50].

During investigation, we output the opcode sequence of exploiting codes. Based on the output opcodes, we will form an opcode signature. If we can successfully construct an opcode signature, we will consider JShield can detect the vulnerability. If the signature

Table 3: Feasibility Comparison of JShield with Other Vulnerabilities-based Drive-by Download Attack Detection.

Vulnerability Position	BrowserShield [44]	Song et al. [50]	JShield
JS Engine	3/22	0/22	22/22
PDF JS Engine	4/18	0/18	18/18
Plug-in	20/21	21/21	21/21

Note: All the numbers in the table are theoretical detection ratio for each approach. BrowserShield represents those with simple policy without control and data flow matching, and Song et al. represent those monitoring only intra-module communications.

Table 4: JShield's Detection Accuracy under Data Pollution.

	Original	Pollted
TP for Web Pages	100%	100%
FP for Web Pages	0%	0%
TP for PDF	100%	100%
FP for PDF	0%	0%

Note: TP is short for true positive, and FP is short for false positive.

contains only one state, we will consider BrowserShield [44] can detect the vulnerability. If the vulnerability does not contain any multi-module communication, we will consider Song et al. [50] cannot detect the vulnerability.

8.2.2 Coverage Results

Three types of vulnerabilities are evaluated here, which are JavaScript engine vulnerabilities of web browsers, JavaScript engine vulnerabilities of PDF readers, and inter-module plug-in vulnerabilities, respectively.

- *JavaScript Engine Vulnerabilities of Web Browsers.* We evaluate all twenty four JavaScript engine vulnerabilities from 2009 to 2014. We do not find any test cases for CVE-2011-2991 and thus skip this vulnerability. CVE-2011-4682, which bypass the same origin policy, is out of scope of this paper and thus skipped. For year 2014, because both NVD and CVE prevent public view of recent vulnerabilities due to security concerns, we can only find two JavaScript engine vulnerability at the time of our investigation.

  The results show that JShield can detect all of those vulnerabilities. BrowserShield [44] can only detect 3 out of 22 JavaScript engine vulnerabilities. Song et al. [50] cannot detect any of those vulnerabilities because they are not plug-in vulnerabilities. A summary is shown in the second row of Table 3.
- *JavaScript Engine Vulnerabilities of PDF Reader.* Except for five cross-site scripting vulnerabilities, all the JavaScript engine vulnerabilities of pdf readers from 2007 to 2014. We cannot look in the references related to two Google chrome pdf vulnerability because Google chrome group restrict the permission of viewing recent vulnerabilities. The results of detecting the rest vulnerabilities are shown in the third row of Table 3.
- *Inter-module Plug-in Vulnerabilities.* We obtain all the plug-in vulnerabilities from Song et al. [50], and find that JShield can detect all the plug-in vulnerabilities as shown in the fourth row of Table 3.

8.3 Robustness to Data Pollution

The same malicious data set and methodology as Section 3.1 are used here for web pages. Benign data set for web pages are extracted from Top 500 Alexa web sites. Both malicious and benign PDFs are acquired from security company containing 214 benign and 213 malicious, and the same data pollution for web pages are performed. Since there is no training stage for JShield, the pollution of training data is not applicable for JShield. Results are shown in Table 4. Excluding those samples only contain HTML but no JavaScript, JShield can detect all those samples. Note that there is no difference between the detection rate of original data and data with injected benign features. As pointed in Section 2, those attacks are out of our threat model. We do not have any false positives because the vulnerability signatures containing control and data flow information can precisely describe the corresponding vulnerability. Interestingly, we even find that there is no web site passing the first stage, *i.e.* regular expression matching of the filter format.

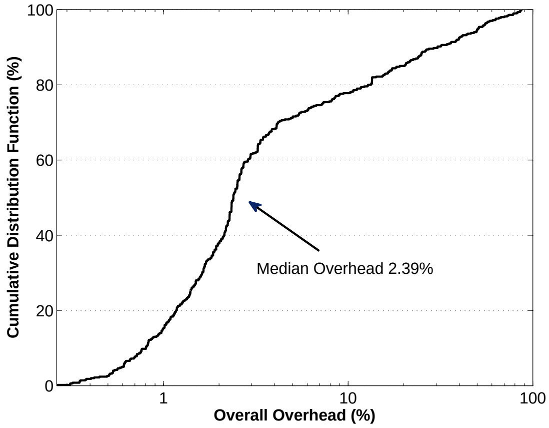

Figure 12: Cumulative Distribution Function (CDF) of Overhead Introduced by JShield on Top 500 Alexa Web Site (The average is 9.42%, and the median is 2.39%.)

8.4 Performance

We measure the performance of signature matching engines of JShield and overall performance respectively in Section 8.4.1 and 8.4.2.

8.4.1 Signature Matching Performance

We measure the JavaScript execution latency introduced by JShield against the original execution latency of top ten Alexa [3] web sites. We measure the latency by injecting codes into WebKit engine before and after JavaScript execution. Normally JavaScript is executed multiple times so we accumulated all the latency together. Both the filtering and the detection format opcode matching are measured.

Note that for top one hundred Alexa web sites, we do not find any that can bypass the pre-filtering stage against all the vulnerabilities in our library. In other words, there is no *false positive* for top one hundred Alexa web sites given our data set.

In Figure 13, we show the latency introduced by the filtering and the matching process of JShield. For matching with detection signatures, the latency is comparable to the latency (2 to 14 times slower) introduced by Nozzle [43] without object sampling. In the worst case, Twitter is about 6 times slower than normal execution. The fastest web site Wikipedia is within 2% because there is little JavaScript hosted on Wikipedia.

For filtering, since regular expression matching is very fast, it is predictable that the latency is small. As shown in the figure, filtering overhead is within 2%.

8.4.2 Overall Performance

In Figure 12, we show the overall performance overhead introduced by JShield. The experiment is performed on Top 500 Alexa [3] web site. We test each web site ten times with and without JShield support, get the average latency and then calculate the overhead. The average overhead among 500 web sites is 9.42%, the median overhead is 2.39%, and the standard deviation is 3.17%. Since the pre-filtering process - regular expression matching is very

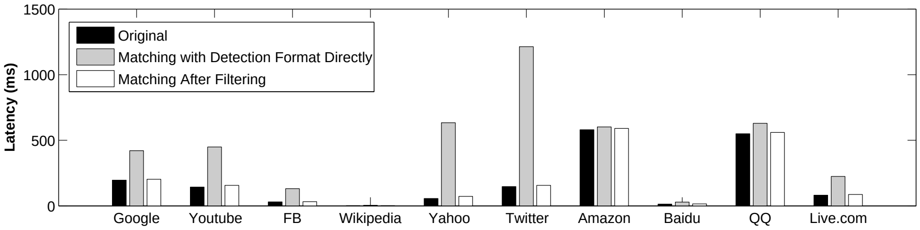

Figure 13: JavaScript Execution Latency Introduced by Matching Filtering and Detection Format of Opcode Signature in JShield (Note that none of the web sites in the graph will actually pass to stage two - detection format matching given our library. We force JShield perform detection format matching for measurement purpose only.)

fast, most of the overhead is introduced by the event triggering engine.

9. DISCUSSION

Does the opcode signature set change if we implement JShield with a different browser kernel?
Answer: Yes, one opcode signature set binds to one specific JShield implementation, which is similar to the case that one binary can only be executed upon one operating system. However, vulnerabilities in any browser can be represented in one JShield implementation since we do not require the vulnerability to exist.

Do the attackers need to know features extracted by anomaly-based detection method to pollute samples?
Answer: No, at detection stage, benign features are extracted from normal web sites instead of those used by anomaly-based detection engine (generated from normal web sites and malicious web sites).

10. RELATED WORK

In this section, we discuss the related work from three aspects: pre-exploit stage, exploit stage, and the others.

10.1 Pre-exploit Stage

Approaches Detecting Heap-spraying Attacks. Instead of detecting a drive-by vulnerability, researchers propose to detect the heap spraying stage of a drive-by download attack. For example, Nozzle [43] and Egele et al. [25] detect every object/string created by JavaScript to examine whether it is executable. DieHarder [40] provides a new memory allocator for securing the heap from execution.

Those approaches [25, 43] detecting heap spraying can be evaded by newly-emerged technology such as Heap Taichi [24], and most importantly, heap spraying is not a necessary step for a drive-by download attack. As shown in the dissected and categorized Malware samples reported by Zozzle [23], 6 out of 19 samples are not using heap spraying. There are two reasons:

- Address space layout randomization (ASLR) is not enabled by every browser. For example, Internet Explorer 7 on Windows Vista prior to SP1 does not enable ASLR by default [2].
- Several other techniques, such as JIT spraying [10] and spraying by calling other languages including VBScript and ActionScript, can substitute JavaScript heap spraying.

De-obfuscating JavaScripts. Rozzle [30] adopts symbolic execution, multi-execute JavaScript code, and partially mitigates differences between multiple browsers. Other previous approaches [18, 23, 33, 46] mostly execute JavaScript and acquire de-obfuscated JavaScript code. Revolver [29] compares the similarity between different JavaScript samples and cluster them based on AST features, however as shown in Figure 10, two samples with different ASTs can trigger the same vulnerability.

As an important step of detecting malicious JavaScript code, those de-obfuscation techniques can be deployed together with JShield. For example, Rozzle can be included in JShield as a component in the de-obfuscation engine to defeat those mechanisms that detect the monitoring environment. Since we do not have any open source Rozzle available, we leave it as our future work.

10.2 Exploit Stage

There are two ways detecting drive-by download attacks.
Vulnerability-based Approaches. BrowserShield [44], a vulnerability-based detection, checks whether a JavaScript operation violates pre-defined policy, thus leading to an attack. Similarly, Song et al. [50] proposes a vulnerability signature to detect plug-in vulnerabilities by checking inter-module communication.

Signatures used in BrowserShield [44] and Song et al. [50] cannot represent stateful intra-module vulnerability such as the one in *Example I*. Further, neither BrowserShield [44] nor Song et al. [50] have considered polymorphic attacks targeting at the same JavaScript vulnerability, like the one in *Example I* of Section 3.
Anomaly-based Approaches. Researchers propose many anomaly-based approaches, such as Zozzle [23], JSAND [22], CUJO [46], and Wepawet [18]. As illustrated in Section 3, anomaly-based approaches have several limitations in adversarial environment, *i.e.*, an attacker can utilize a polymorphic variance of existing JavaScript exploit codes or inject false malicious features to bypass anomaly-based approaches.

10.3 Others

Static Methods. There are also many static analysis of detecting malicious web page, including but not restricted to Prophiler [21], Seifert et al. [48, 49], Ma et al. [36], and so on. Obfuscation technique, such as embedding into *eval* and *document.write*, can evade those approaches.
Protection Mechanisms. There are many protection mechanisms [26, 45, 51, 53] sandboxing a browser principal, which isolate a browser principal from other parallel browser principals and the host operating system. Blade [34] detects whether an executable is downloaded through a browser GUI. If it is not from a browser GUI, the downloaded executable will be quarantined. BrowserGuard [27] adopts similar behaviour based approach to detect downloaded files. Cisco IronPort [6], SpyProxy [37] and WebShield [31] detect drive-by download at middle box and transfer safe contents back to the client. All of those are effective protection mechanisms, but it is far from deploying them upon all the client browsers and enterprise network.
Behavior Based Detection. Provos et al. [42] and Google Safe Browsing [9] use anti-virus software and execution based heuristics to detect the malicious behavior of downloaded malware. Many other approaches [38, 47, 55] use high-interaction client honeypots for detection. Their detection scope is limited because the vulnerability condition might not be triggered in their specific detection environment.

11. CONCLUSION

Due to possible data pollution for anomaly-based detection of drive-by download attacks and no complete vulnerability representation for vulnerability-based detection, we propose opcode vulnerability signature, consisting of a definitive state automaton and a variable pool to represent both control and data flow of a vulnerability condition. We implement a prototype system, called JShield, which de-obfuscates JavaScript by event triggering and then performs opcode signature matching. Next, we investigate all the JavaScript engine vulnerabilities of web browsers from 2009 to 2014, and those of PDF from 2007 to 2014. We find that JShield can detect all of them. We also find that JShield can represent all the inter-module plug-in vulnerabilities obtained from Song et al. [50].

12. ACKNOWLEDGEMENT

This material is based upon work supported in part by Qatar National Research Fund under award ID SP0022512 and National Natural Science Foundation of China under grant NO. 61472209. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding foundations.

13. REFERENCES

[1] 2013 compare the best antivirus software products. http://anti-virus-software-review.toptenreviews.com/.

[2] Address space layout randomization. http://en.wikipedia.org/wiki/Address_space_layout_randomization#Microsoft_Windows.

[3] Alexa Top Websites. http://www.alexa.com/topsites.

[4] Apple hit by hackers who struck facebook. http://online.wsj.com/article/SB10001424127887324449104578314321123497696.html.

[5] Cisco ips signatures. http://tools.cisco.com/security/center/ipshome.x?i=62&shortna=CiscoIPSSignatures#CiscoIPSSignatures.

[6] Cisco ironport. http://www.cisco.com/web/ironport/index.html.

[7] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[8] Embedding perl (using perl from c). http://docstore.mik.ua/orelly/perl/prog3/ch21_04.htm.

[9] Google safe browsing. https://developers.google.com/safe-browsing/.

[10] JIT spraying. http://en.wikipedia.org/wiki/JIT_spraying.

[11] National vulnerability database. http://nvd.nist.gov/.

[12] Regular expression library in c++. http://www.cplusplus.com/reference/std/regex/.

[13] Snort rules. http://www.snort.org/snort-rules/.

[14] Spidermonkey javascript engine. https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[15] Standard template library. http://www.sgi.com/tech/stl/.

[16] V8 javascript engine. https://code.google.com/p/v8/.

[17] Webkit source codes. http://webkit.org/building/checkout.html.

[18] Wepawet. http://wepawet.iseclab.org/.

[19] Barth, A., Jackson, C., and Mitchell, J. Robust defenses for cross-site request forgery. In *CCS* (2008).

[20] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based signatures. In *SP: the 2006 IEEE Symposium on Security and Privacy* (2006).

[21] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *WWW* (2011).

[22] Cova, M., Kruegel, C., and Vigna, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW* (2010).

[23] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: fast and precise in-browser javascript malware detection. In *the 20th USENIX conference on Security* (2011).

[24] Ding, Y., Wei, T., Wang, T., Liang, Z., and Zou, W. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *ACSAC* (2010).

[25] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA* (2009).

[26] Grier, C., Tang, S., and King, S. T. Secure web browsing with the OP web browser. In *SP: IEEE Symposium on Security and Privacy* (2008).

[27] Hsu, F.-H., Tso, C.-K., Yeh, Y.-C., Wang, W.-J., and Chen, L.-H. Browserguard: A behavior-based solution to drive-by-download attacks. *Selected Areas in Communications, IEEE Journal on* (2011).

[28] Jensen, S. H., Møller, A., and Thiemann, P. Type analysis for javascript. In *SAS: the International Symposium on Static Analysis* (2009).

[29] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., and Vigna, G. Revolver: An automated approach to the detection of evasive web-based malware. In *USNIX Security Symposium* (2013).

[30] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-cloaking internet malware. In *SP: IEEE Symposium on Security and Privacy* (2012).

[31] Li, Z., Tang, Y., Cao, Y., Rastogi, V., Chen, Y., Liu, B., and Sbisa, C. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS* (2011).

[32] Li, Z., Xia, G., Gao, H., Yi, T., Chen, Y., Liu, B., Jiang, J., and Lv, Y. Netshield: Massive semantics-based vulnerability signature matching for high-speed networks. In *SIGCOMM* (2010).

[33] Lu, G., Coogan, K., and Debray, S. K. Automatic simplification of obfuscated javascript code (extended abstract). In *ICISTM: 6th International Conference of Information Systems, Technology and Management* (2012).

[34] Lu, L., Yegneswaran, V., Porras, P. A., and Lee, W. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *CCS* (2010).

[35] Lu, X., Zhuge, J., Wang, R., Cao, Y., and Chen, Y. De-obfuscation and detection of malicious pdf files with high accuracy. In *HICSS* (2013).

[36] Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *SIGKDD* (2009).

[37] Moshchuk, A., Bragin, T., Deville, D., Gribble, S., and Levy, H. Spyproxy: Execution-based detection of malicious web content. In *16th USENIX Security Symposium* (2007).

[38] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. A crawler-based study of spyware on the web. In *NDSS* (2006).

[39] Mozilla. Narcissus javascript engine. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[40] Novark, G., and Berger, E. D. DieHarder: securing the heap. In *CCS* (2010).

[41] Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M. Misleading worm signature generators using deliberate noise injection. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[42] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All your iframes point to us. In *17th USENIX Security Symposium* (2008).

[43] Ratanaworabhan, P., Livshits, B., and Zorn, B. Nozzle: A defense against heap-spraying code injection attacks. In *18th USENIX Security Symposium* (2009).

[44] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI* (2006).

[45] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *EuroSys* (2009).

[46] Rieck, K., Krueger, T., and Dewald, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC* (2010).

[47] Seifert, C., Steenson, R., Holz, T., Yuan, B., , and Davis, M. A. Know your enemy: Malicious web servers. http://www.honeynet.org/papers/mws/.

[48] Seifert, C., Welch, I., and Komisarczuk, P. Identification of malicious web pages with static heuristics. In *ATNAC: Telecommunication Networks and Applications Conference* (2008).

[49] Seifert, C., Welch, I., Komisarczuk, P., Aval, C. U., and Endicott-Popovsky, B. Identification of malicious web pages through analysis of underlying dns and web server relationships. In *LCN: Local Computer Networks* (2008).

[50] Song, C., Zhuge, J., Han, X., and Ye, Z. Preventing drive-by download via inter-module communication monitoring. In *ASIACCS* (2010).

[51] Tang, S., Mai, H., and King, S. T. Trust and protection in the illinois browser operating system. In *OSDI* (2010).

[52] Ter Louw, M., and Venkatakrishnan, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (2009).

[53] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. The multi-principal OS construction of the gazelle web browser. In *18th Usenix Security Symposium* (2009).

[54] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM* (2004).

[55] Wang, Y., Beck, D., Jiang, X., and Roussev, R. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS* (2006).