December 2017

DETECTION, DIAGNOSIS AND MITIGATION OF MALICIOUS JAVASCRIPT WITH ENRICHED JAVASCRIPT EXECUTIONS

Xunchao Hu
Syracuse University

ABSTRACT

Malicious JavaScript has become an important attack vector for software exploitation attacks and imposes a severe threat to computer security. In particular, three major class of problems, malware detection, exploit diagnosis, and exploits mitigation, bring considerable challenges to security researchers. Although a lot of research efforts have been made to address these threats, they have fundamental limitations and thus cannot solve the problems.

Existing analysis techniques fall into two general categories: static analysis and dynamic analysis. Static analysis tends to produce inaccurate results (both false positive and false negative) and is vulnerable to a wide series of obfuscation techniques. Thus, dynamic analysis is constantly gaining popularity for exposing the typical features of malicious JavaScript. However, existing dynamic analysis techniques possess limitations such as limited code coverage and incomplete environment setup, leaving a broad attack surface for evading the detection.

Once a zero-day exploit is captured, it is critical to quickly pinpoint the JavaScript statements that uniquely characterize the exploit and the payload location in the exploit. However, the current diagnosis techniques are inadequate because they approach the problem either from a JavaScript perspective and fail to account for "implicit" data flow invisible at JavaScript level, or from a binary execution perspective and fail to present the JavaScript level view of exploit.

Although software vendors have deployed techniques like ASLR, sandbox, etc. to mitigate JavaScript exploits, hacking contests (e.g.,PWN2OWN, GeekPWN) have demonstrated that the

latest software (e.g., Chrome, IE, Edge, Safari) can still be exploited. An ideal JavaScript exploit mitigation solution should be flexible and allow for deployment without requiring code changes.

To combat malicious JavaScript, this dissertation addresses these problems through enriched executions, which explore arbitrary paths for detection, preserve JS-binary semantics for diagnosis, and perturbs memory with chaff code for mitigation.

Firstly, JSForce, a forced execution engine for JavaScript, is proposed and developed to improve the detection results of current malicious JavaScript detection techniques. It drives an arbitrary JavaScript snippet to execute along different paths without any input or environment setup. While increasing code coverage, JSForce can tolerate invalid object accesses while introducing no runtime errors during execution.

Secondly, JScalpel, a system that utilizes the JavaScript context information from the JavaScript level to perform context-aware binary analysis, is presented for JavaScript exploit diagnosis. In essence, it performs JS-Binary analysis to (1) generate a minimized exploit script, which in turn helps to generate a signature for the exploit, and (2) precisely locate the payload within the exploit. It replaces the malicious payload with a friendly payload and generates a PoV for the exploit.

Thirdly, ChaffyScript, a vulnerability-agnostic mitigation system, is introduced to block JavaScript exploits via undermining the memory preparation stage. Specifically, given suspicious JavaScript, ChaffyScript rewrites the code to insert memory perturbation code, and then generates semantically-equivalent code. JavaScript exploits will fail as a result of unexpected memory states introduced by memory perturbation code, while the benign JavaScript still behaves as expected since the memory perturbation code does not change the JavaScript's original semantics.

DETECTION, DIAGNOSIS AND MITIGATION OF MALICIOUS JAVASCRIPT WITH ENRICHED JAVASCRIPT EXECUTIONS

by

Xunchao Hu

B.S. Xi'an Jiaotong University, China, 2009
M.S. Xi'an Jiaotong University, China, 2012

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical & Computer Engineering

Syracuse University
December 2017

To my family

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to all who contribute and support my doctoral dissertation and graduate student life at Syracuse University.

First, it has been my great privilege and great pleasure to work under the supervision of my advisor, Professor Heng Yin. I could not have completed my research work without his insightful advice, persistent encouragement, and constant support. Working with Heng is the most brilliant decision that I have ever made since I came to USA. I also want to express my sincere respect to his academic enthusiasm and professional dedication.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Wenliang Du, Prof. C.Y. Roger Chen, Prof. Yuzhe Tang, Prof. Yanzhi Wang and Prof. Joon S. Park for their insightful comments and hard questions.

I would like to thank my fellow labmates: Mu, Andrew, Qian, Yue, Rundong and Jinghan for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years.

Special thanks are given to my family: my parents Jiayong Hu and Fengqin Gu, for giving birth to me at the first place and supporting me spiritually throughout my life; my brother Xunxiang Hu, for the continuous encouragement and support throughout my Ph.D. study.

Also, I would like to thank my son Asher Hu. He almost kills my dissertation with endless noise and extremely bad sleeping schedule.

Last but not the least, I take this opportunity to express my deepest love and overwhelming thank to my wife, Yingying Liu. This dissertation will not be possible without your love and support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

JavaScript has been a popular programming language for decades. JavaScript engines are now embedded in many other types of host software, including client-side web browsers (e.g., Chrome, FireFox, Internet Explorer), server-side in web servers and databases (e.g. Node.js), and in non-web programs such as word processors and PDF software (e.g. Adobe Reader).

Malicious JavaScript take advantage of the interactive nature of JavaScript to exploit binary vulnerabilities (e.g., use-after-free, heap/buffer overflow) of host software that are otherwise difficult to exploit. JavaScript provides attackers with a much easier way to conduct heap spraying [1], information leakage [2], and shellcode generation. Figure 2.3 presents a typical malicious JavaScript that exploit the vulnerability of IE to gain remote code execution. Malicious JavaScript are leveraged by the exploit kits used in drive-by-download attacks to remotely exploit the client side vulnerabilities to install malware on the victim machine. This has led to the emergence of an ”Exploit-as-a-Service” paradigm within the malware ecosystem [3].

Malicious JavaScript has become an important attack vector for software exploitation attacks. Attacks in browsers, as well as PDF files containing malicious embedded JavaScript, are typical examples of how attackers launch attacks using JavaScript. According to a recent report from Symantec [4], there are millions of victims attacked by malicious JavaScript on the Internet each day. In particular, three major class of problems, *malware detection, exploit diagnosis, and exploits mitigation*, bring considerable challenges to security researchers. Although a lot of

research efforts have been made to address these threats, they have fundamental limitations and thus cannot solve the problems.

Malware Detection In recent years, a number of techniques [5–13] have been proposed to detect malicious JavaScript code. Due to the dynamic features of the JavaScript language, static analysis [11, 14–16] can be easily evaded using obfuscation techniques [17]. Consequently, researchers rely upon dynamic analysis [5–7] to expose the typical features of malicious JavaScript. More specifically, these approaches rely upon visiting websites or opening PDF files with a full-fledged or emulated browser/PDF reader and then monitoring the different features ($eval$ strings [7], heap health [10], etc.) for detection.

However, the typical JavaScript malware is designed to execute within a particular environment, since they aim to exploit specific vulnerabilities, as opposed to benign JavaScript, which will run in a more environment-independent fashion. Fingerprinting techniques [18] are widely adopted by JavaScript malware to examine the runtime environment. A dynamic analysis system may fail to observe some malicious behaviors if the runtime environment is not configured as expected. Such configuration is quite challenging because of the numerous possible runtime environment settings. Hence, existing dynamic analysis systems usually share the limitations of limited code coverage and incomplete runtime environment setup, which leave attackers with a broad attack surface to evade the analysis.

Exploit Diagnosis Once a ”zero-day” JavaScript attack is captured, it must be analyzed and its inner-workings understood quickly so that proper defenses can be deployed to protect against it or similar attacks in the future. Unfortunately, this analysis process is tedious, painstaking, and

time-consuming. From the analysis perspective, an analyst seeks to answer two key questions: (1) Which JavaScript statements uniquely characterize the exploit? And (2) Where is the payload located within the exploit? The answer to the first question results in the generation of an exploit signature, which can then be deployed via an intrusion detection system (IDS) to discover and prevent the exploit. The answer to the second question allows an analyst to replace the malicious payload with an amicable payload and use the modified exploit as a proof-of-vulnerability (PoV) to perform penetration testing.

Prior exploit analysis solutions have attempted to analyze exploits at either the JavaScript level [5, 9–11, 19, 20] or the underlying binary level [21–25]. While binary level solutions execute an exploit and analyze the underlying binary execution for anomalies, they are unaware of any JavaScript level semantics and fail to present the JavaScript level view of the exploit. JavaScript level analysis fails to account for implicit data flows between statements because any DOM/BOM APIs invoked at the binary level are invisible at the JavaScript level. Unfortunately, implicit flows are quite common in attacks and are often comprised of seemingly random and irregular operations in the JavaScript that achieve a precise precondition or a specific trigger which exploits a vulnerability in the binary. The semantic gap between JavaScript level and binary level during the analysis makes it challenging to automatically answer the 2 key questions.

Exploit Mitigation JavaScript has been used to exploit the vulnerabilities found in software. A typical JavaScript exploit can be divided into three stages. (1). Pre-Exploitation. This stage examines the versions of underline operating system and software to determine exploits configuration. (2). Exploitation. This stage prepares the memory, triggers the vulnerability, disclosures the memory information, injects the payload and achieves the code execution. (3).

Post-Exploitation. After stage (2), this stage has gained the execution of code. But it requires Return Oriented Programing (ROP) to bypass DEP, and then executes payload, keeps persistent in the infected system.

To mitigate such kind of attacks, different approaches have been proposed. At JavaScript level, approaches like BrowserShield [26], instrument the sensitive JavaScript operations to stop known attacks by matching the predefined policies. Program hardening approaches like Control Flow Integrity [27], Code Pointer Integrity [28], ROP mitigation [29, 30], etc., stop the attacks by defeating different exploitation stages.

While these exploit mitigation techniques are constantly improving, hacking contests like Pwn2Own [31], GeekPwn [32], etc., consistently demonstrate that the latest versions of Chrome, Safari, Internet Explorer, and Edge can still be exploited. There are two reasons for this: First, most of the latest proposed mitigation techniques require software or compiler tool chain changes and thus could not be deployed promptly. For instance, ASLR-guard [33] is designed to thwart information disclosure attacks, but requires compiler changes and cannot be quickly deployed by software vendors. Second, the deployed mitigation techniques may fail due to newly invented exploitation techniques (e.g., sandbox bypass technique). An ideal mitigation technique should be flexible to deploy without requiring code changes and should subvert inevitable exploitation stage(s).

1.1 Thesis Statement

My thesis work aims to combat malicious JavaScript through *enriched executions, which explore arbitrary paths for detection, preserve JS-binary semantics for diagnosis, and perturb*

memory with chaff code for mitigation. To achieve this goal, I propose the following three new techniques to address the specific security problems.

(1) **Forced Execution for Malicious JavaScript Detection.** To battle malicious JavaScript, I propose JSForce, a forced execution engine for JavaScript, which drives an arbitrary JavaScript snippet to execute along different paths without any input or environment setup. While increasing code coverage, JSForce can tolerate invalid object accesses while introducing no runtime errors during execution.

(2) **Semantics-Preserving Dissection of JavaScript Exploits.** To answer the 2 key questions in exploit diagnosis, I present JScalpel, a system that utilizes the JavaScript context information from the JavaScript level to perform the context-aware binary analysis. In essence, it performs JS-Binary analysis to (1) generate a minimized exploit script, which in turn helps to generate a signature for the exploit, and (2) precisely locate the payload within the exploit. It replaces the malicious payload with a friendly payload and generates a PoV for the exploit.

(3) **Vulnerability-Agnostic Defense of JavaScript Exploits via Memory Perturbation.** To mitigate JavaScript exploits, I propose ChaffyScript, a vulnerability-agnostic mitigation system that blocks JavaScript exploits via undermining the memory preparation stage. Specifically, given suspicious JavaScript, ChaffyScript rewrites the code to insert memory perturbation code, and then generates semantically-equivalent code. JavaScript exploits will fail as a result of unexpected memory states introduced by memory perturbation code, while the benign JavaScript still behaves as expected since the memory perturbation code does not change the JavaScript's original semantics. ChaffyScript does not require any code change of host software and is flexible to deploy.

2. BACKGROUND

The drive-by-download attacks drive the emergence of "Exploit-as-a-Service" paradigm on the malware ecosystem [34]. These attacks, mostly launched via malicious JavaScript, have attracted a lot of research efforts both from academia and industry. In this chapter, the inner-workings of JavaScript exploits are first discussed to demonstrate how JavaScript is used to launch the attack. Then a discussion on the related work of detection, diagnosis, and mitigation for malicious JavaScript is presented to illustrate an overview of current research on malicious JavaScript.

2.1 JavaScript Exploits

JavaScript has been a popular programming language for decades. JavaScript engines are now embedded in many other types of host software, including client-side web browsers (e.g., Chrome, FireFox, Internet Explorer), server-side in web servers and databases (e.g. Node.js), and in non-web programs such as word processors and PDF software (e.g. Adobe Reader).

The interactive nature of JavaScript allows malicious JavaScript to take advantage of binary vulnerabilities (e.g., use-after-free, heap/buffer overflow) of host software that are otherwise difficult to exploit. JavaScript provides attackers with a much easier way to conduct heap spraying [1], information leakage [2], and shellcode generation. The exploit kits used in drive-by-download attacks leverage JavaScript code to remotely exploit the client side vulnerabilities to install malware on the victim machine. This has led to the emergence of an

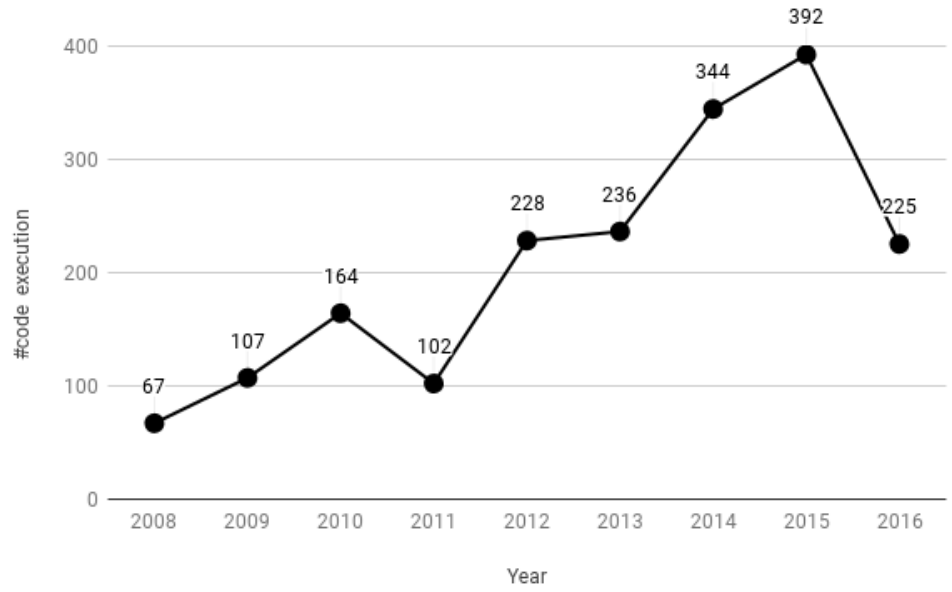

Fig. 2.1.: The number of code execution vulnerabilities discovered in popular web browsers (Chrome, Edge, IE, Safari, and FireFox) reported by CVEDetails [35]

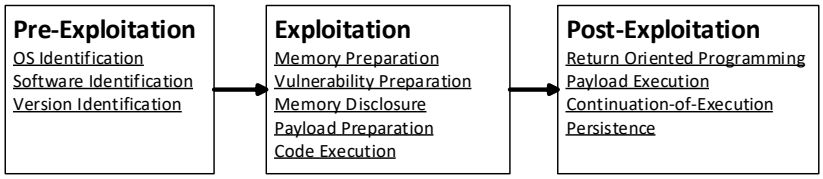

Fig. 2.2.: The overall exploitation stages

”Exploit-as-a-Service” paradigm within the malware ecosystem [3]. As illustrated in Figure 2.1, in popular web browsers, hundreds of code execution vulnerabilities are still discovered every year. This provides the attacker with a broad attack surface.

Figure 2.2 illustrates the high level stages of JavaScript exploits [36]. In the pre-exploitation stage, malware fingerprints the victim machine to determine the OS version and target software and then launches the corresponding exploit. The exploitation stage triggers the vulnerability, bypassing exploit mitigation techniques (e.g., ASLR, EMET [37], Control Flow Guard [38]) and

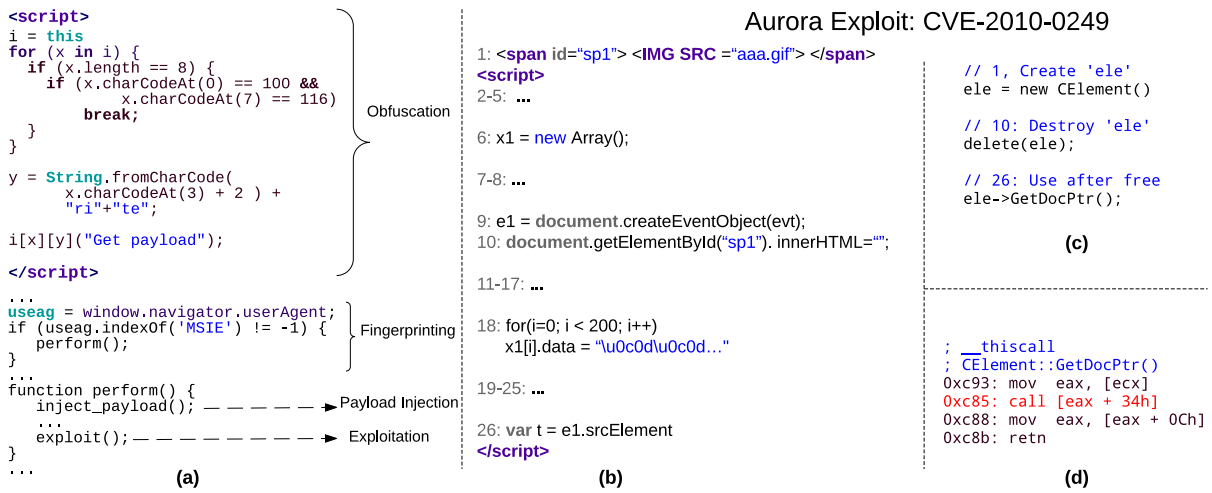

Fig. 2.3.: (a) describes the components of a modern exploit, (b) presents the relevant JavaScript code involved in Aurora Exploit and (c) presents the underlying code execution that results in use-after-free, (d) presents the assembly code for function GetDocPtr.

diverts the code execution to the injected payload. The post-exploitation stage executes a Return-Oriented-Programming (ROP) payload to bypass DEP, drops the malicious payload while attempting to evade detection from endpoint security products.

2.1.1 Anatomy of a JavaScript Attack

Modern JavaScript attacks can be compartmentalized into four general components. Figures 2.3 (a) and (b) show these four components within the Aurora exploit. Below, we briefly describe the four components.

Obfuscation: From the perspective of an attacker, the identity of the exploit server and the arsenal of exploits – especially zero-day exploits – must be obfuscated to avoid detection by anti-malware analyzers. For example, in Figure 2.3 JavaScript obfuscation is used to perform a `document.write("Get payload")` operation. Malicious scripts often use

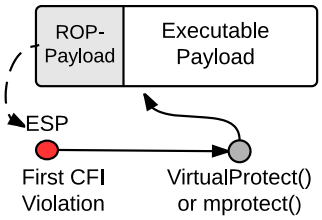

Fig. 2.4.: Non-executable (ROP) and executable payloads used in an exploit.

`document.write()` to inject code components at runtime. In Figure 2.3(a), the Aurora script iterates over `this` object and finds the `document` object at runtime. Simple static analysis-based scanners cannot identify that "i[x][y]" is actually a `document.write()` operation. Several other obfuscation techniques are employed by attackers to systematically defeat various analysis techniques [39]. On the de-obfuscation front, several solutions have been proposed to deobfuscate JavaScript (e.g., Wepawet [5], JSUnpack [7]).

Fingerprinting: An exploit uses fingerprinting to glean information about victim's environment. With such information, exploits specific to vulnerable components are launched to compromise the victim process. For example, in Figure 2.3(a), the Aurora exploit is only performed if the type of the browser is identified as being Microsoft Internet Explorer ("MSIE"). Exploit kits are known to use PluginDetect [40] to fingerprint browsers.

Payload Injection: The exploit injects a malicious payload into the victim process. Payloads can be broadly categorized as executable or non-executable payloads. Figure 4.4 presents the payloads and the flow of execution in modern exploits. The goal is to execute a malicious payload, but since the wide deployment of data execution prevention (DEP), the page containing the executable payload cannot be directly executed. First, return-oriented programming (ROP) is

used to make a page executable by invoking `VirtualProtect()` on Windows or `mprotect()` on Linux. Then, control is transferred to the malicious executable code.

In fact, though uncommon, it is possible for attacks to introduce gadgets [41]. Though our solution makes no assumptions about how the payload is stored, or what statements are used to inject the payload, the payload is typically injected using an encoded string that contains the executable. The statements that inject the payload are usually independent of the exploit code. Payload detection can be challenging because it is possible for both executable and non-executable payloads to reside within the same string.

Exploitation: In this step, using one or more carefully crafted JavaScript statements, the vulnerability in the victim process is exploited. The statements may seem random and may lack data-dependencies, but they often involve a combination of explicit and implicit data dependency. Consider the exploit statements for the Aurora exploit presented in Figure 2.3(b, c and d). (b) presents the HTML (statement 1) and JavaScript (2-26) statements that exploit a use-after-free vulnerability in `mshtml.dll` of Internet Explorer browser. Figures 2.3(c and d) present the underlying C++ and assembly code that is executed as a part of the exploit. Statement 18 corrupts the memory that was freed in statement 10. The corrupted memory is utilized in a `call` instruction arising from statement 26. All the statements in Figures 2.3(c) are pertinent to the exploit.

2.1.2 Unique Features of Malicious JavaScript

From the discussion in 2.1.1, a typical malicious JavaScript has two unique features- dynamic nature and interaction between JavaScript level and binary level.

Dynamic nature JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It supports many dynamic features [42] (e.g., call-site dynamism, `eval`, dynamic typing, etc.). While providing the flexibility of programming for developers, the dynamic features of JavaScript empower the attacker to highly obfuscate the malicious JavaScript for evasion purpose. For instance, `eval` can be used to hide malicious JavaScript in `String`. Call-site dynamism can be used to hide callee name within `String` to bypass signature-based detection system. These dynamic features are broadly employed by malicious JavaScript in the obfuscation process to evade the detection. This inspires us to propose JSForce, a forced execution engine, to improve the detection rate of JavaScript analysis system.

Interaction between JavaScript level and binary level Malicious JavaScript is used to exploit the vulnerability of host software. The complete exploitation is a close interaction process between JavaScript level and binary level. At runtime, native functions provided by host software are invoked to implement the semantics of interpreted JavaScript. The craft JavaScript triggers the vulnerability in host software, and diverts the execution to the injected payload. While the exploitation operations are specified using JavaScript, they affect the memory states and calling context states of host software at the binary level. This close interaction between JavaScript level and binary level inspires us the work JScalpel and ChaffyScript.

2.2 Detection

In the last few years, there have been a number of approaches to analyzing JavaScript code. They can be roughly divided into two categories-static approach, dynamic approach.

Static Approach. Several systems have focused on statically analyzing JavaScript code to identify malicious web pages [11, 14–16]. ZOZZLE [11], in particular, leverages features associated with AST context information (such as, the presence of a variable named shellcode in the context of a loop), for its classification. Since dynamic features of JavaScript plague the static analysis , researchers try to model those features to improve the static analysis result [43–45].

Dynamic Approach. Dynamic analysis is widely deployed to expose behaviors of obfuscated JavaScript code. Previous work [5–7] execute JavaScript using an emulated JavaScript running environment and acquire de-obfuscated JavaScript code. To de-obfuscate malicious JavaScript code, Gen et al. [6] simplify the obfuscated JavaScript code by preserving the semantics of the observational equivalence. JSGuard [8] proposed a methodology to detect JavaScript shellcode that fully uses JavaScript code execution environment information with low false negative and false positive. Liu et al. [46] propose a context-aware approach for detection and confinement of malicious JavaScript in PDF by inserting context monitoring code into a document. To analyze JavaScript code with cloaking, Kolbitsch et al. [9] uncover environment-specific malware by exploring multiple execution paths within a single execution. CODENAME can benefit the dynamic analysis in terms of improved code coverage and tolerance of invalid host environment model.

Researchers also try to combine static and dynamic code features to identify malicious JavaScript programs(Cujo [47]). More precisely, Cujo processes the static program and traces of its execution into q-grams that are classified using machine learning techniques. Symbolic execution [19] is also explored for malicious JavaScript analysis.

2.3 Diagnosis

Once a malicious JavaScript attack is captured, it must be analyzed and its inner-workings understood quickly so that proper defenses can be deployed to protect against it or similar attacks in the future. Unfortunately, this analysis process is tedious, painstaking, and time-consuming. Researchers have proposed different solutions for exploit diagnosis. PointerScope [21] uses type inference on binary execution to detect the pointer misuses induced by an exploit. ShellOS [23] built a hardware virtualization based platform for fast detection and forensic analysis of code injection attacks. Dynamic taint analysis [22] keeps track of the data dependency originated from untrusted user input at the instruction level, and detects an exploit on a dangerous use of a tainted input. Panorama [24] explored whole system taint tracking for malware analysis. Chen et al., [48] showed that pointer taintedness analysis could expose different classes of security vulnerabilities, such as format string, heap corruption, and buffer overflow vulnerabilities. Prospector [49] pinpoints the guilty bytes in polymorphic buffer overflows on heap or stack by tagging data from network with an age stamp. Nevertheless, those techniques focus only on binary level diagnosis and are not feasible for complex attacks launched using JavaScript code.

2.4 Mitigation

Mitigation techniques have been evolving with the advancement of exploitation techniques. To name a few:

Control Flow Integrity Beginning from [27], many CFI defenses have been proposed at the source code level[50], at the binary level [51] and at runtime[52]. Microsoft deployed control

flow guard techniques on its products [38] to mitigate the exploits. Dachshund [53] secures against blinded constants in JIT code by removing all constants from JavaScript code.

ROP mitigation Microsofts Enhanced Mitigation Experience Toolkit (EMET) [37] is a popular zero-day exploit prevention utility that provides defense against stack pivot in ROP attacks. StackArmor [54] randomizes the location of the stack, thereby making it harder for an attacker to guess the location of a ROP payload on the stack. ROPecker [55] leverages a hardware feature, the Last Branch Record, to detect the execution of ROP chains.

Randomization Randomization based mitigation is quite effective at stopping exploits, but is vulnerable to information leakage and side channel attacks [56]. To stop information leakage, ASLRGuard [33] stopped the leak of data pointers in deriving code addresses by separating code and data, providing secure storage for code pointers, and encoding the code pointers when they are treated as data. Buble [57] inserted holes in array objects by modifying the JavaScript engine to stop heap spray attacks.

The above mitigation techniques usually require changes to source code or binaries and thus cannot be deployed promptly. In addition, once deployed, they cannot be rapidly upgraded as exploitation techniques advance.

JavaScript Rewriting JavaScript rewriting has been used by researchers to meet various security requirements. BrowserShields [26] uses it to stop JavaScript exploits by matching predefined vulnerability features. But it does not work for 0-day exploits. ConScript [58] rewrites JavaScript to specify and enforce fine-grained security policies for JavaScript in the browser.

Dachshund [53] secures against blinded constants in JIT code via removing all the constants from JavaScript code. But Dachshund only works against JITSpray attacks.

3. A FORCED EXECUTION ENGINE FOR MALICIOUS JAVASCRIPT DETECTION

3.1 Introduction

Malicious JavaScript has become an important attack vector for software exploitation attacks. Attacks in browsers, as well as PDF files containing malicious embedded JavaScript, are typical examples of how attackers launch attacks using JavaScript. According to a recent report from Symantec [4], there are millions of victims attacked by malicious JavaScript on the Internet each day.

In recent years, a number of techniques [5–13] have been proposed to detect malicious JavaScript code. Due to the dynamic features of the JavaScript language, static analysis [11, 14–16] can be easily evaded using obfuscation techniques [17]. Consequently, researchers rely upon dynamic analysis [5–7] to expose the typical features of malicious JavaScript. More specifically, these approaches rely upon visiting websites or opening PDF files with a full-fledged or emulated browser/PDF reader and then monitoring the different features ($eval$ strings [7], heap health [10], etc.) for detection.

However, the typical JavaScript malware is designed to execute within a particular environment, since they aim to exploit specific vulnerabilities, as opposed to benign JavaScript, which will run in a more environment-independent fashion. Fingerprinting techniques [18] are widely adopted by JavaScript malware to examine the runtime environment. A dynamic analysis

system may fail to observe some malicious behaviors if the runtime environment is not configured as expected. Such configuration is quite challenging because of the numerous possible runtime environment settings. Hence, existing dynamic analysis systems usually share the limitations of limited code coverage and incomplete runtime environment setup, which leave attackers with a broad attack surface to evade the analysis.

To solve those limitations, Rozzle [9] explores multiple environment related paths within a single execution. But it requires a predefined environment-related profile for path exploration. Construction of a complete profile is a challenging task because of the numerous different browsers and plugins, especially for recent fingerprinting techniques [59, 60]. These fingerprinting techniques do not rely upon any specific APIs, and thus Rozzle can be evaded because the predefined profile cannot include those fingerprinting techniques. Also, Rozzle may introduce runtime errors because it executes infeasible paths which may stop the analysis before the malicious code is executed. Revolver [12] employs a machine learning-based detection algorithm to identify evasive JavaScript malware. However, it is dependent upon a known sample set and is unable to detect 0-day JavaScript malware. Although symbolic execution of JavaScript [19] can be applied to explore all of the possible execution paths, the performance overhead of a symbolic string solver [61] and the dynamic features of JavaScript make it infeasible for practical use.

In this chapter, I propose JSFORCE, a forced execution engine for JavaScript, which drives an arbitrary JavaScript snippet to execute along different paths without any input or environment setup. While increasing code coverage, JSFORCE can tolerate invalid object accesses while introducing no runtime errors during execution. This overcomes the limitations of current JavaScript dynamic analysis techniques. Note that, as an amplifier technique, JSFORCE does not

rely on any predefined profile information or full- fledged hosting programs like browsers or PDF viewers, and it can examine partial JavaScript snippets collected during an attack. As demonstrated in Section 4.5, JSFORCE can be leveraged to improve the detection rate of other dynamic analysis systems without modification of their detection policies. While the high-level concept of forced execution has been introduced in binary code analysis (X-Force [62], iRiS [63]), we face unique challenges in realizing this concept in JavaScript analysis, given that JavaScript and native code are very different languages by nature.

I implement JSFORCE on top of the V8 JavaScript engine [64] and evaluate the correctness, effectiveness, and runtime performance of JSFORCE with 220,587 HTML files and 23,509 PDF samples. Our experimental results demonstrate that adopting JSFORCE can greatly improve the JavaScript analysis results by 206.29% without any noticeable increase in false positives and with a reasonable performance overhead.

3.2 Background and Overview

To provide the reader with a better understanding of the motivation for our system and the problems that it addresses, we begin with a discussion of the malicious JavaScript code used in drive-by-download attacks.

Malicious JavaScript code Malicious JavaScript code is typically obfuscated and will attempt to fingerprint the version of the victim's software (browser, PDF reader, etc.), identify vulnerabilities within that software or the plugins that that software uses, and then launch one or more exploits. Figure 3.1 shows a listing of JavaScript code used for a drive-by-download attack against the Internet Explorer browser. Line 1 employs precise fingerprinting to deliver only

```
1  if ((navigator.appName.indexOf("Microsoft Inte" + "rnet Explorer") ==
       1) && (navigator.userAgent.indexOf("Windows N" + "T 5.1") ==  1) &&
       (navigator.userAgent.indexOf("MSI" + "E 8.0") ==  1)) {
2      att = btt + 1;
3  }
4  if (att == 0) {
5      try {
6          new ActiveXObject("UM0QS4dD");
7      } catch (e) {
8          var tlMoOul8 = '\x25' + 'u9' + '\x30' + '\x39' + YYGRl6;
9          tlMoOul8 += tlMoOul8;
10         var CBmH8 = "%u";
11         var vBYG0 = unescape;
12         var EuhV2 = "BODY";
13         ...
14     }
15 }
16 setTimeout("redir()", 3000);
```

Fig. 3.1.: The Malicious JavaScript Sample

selected exploits that are most likely to successfully attack the browser. Lines 5-7 contain evasive code to bypass emulation-based detection systems. More precisely, the code attempts to load a non-existant ActiveX control, named `UM0QS4dD` (line 6). When executed within a regular browser, this operation fails, triggering the execution of the `catch` block that contains the exploitation code (lines 7-14).

However, an emulation-based detection system must emulate the ActiveX API by simulating the loading and presence of any ActiveX control. In these systems, the loading of the ActiveX control will not raise this exception. As a result, the execution of the exploit never occurs and no malicious activity is observed. Instead, the victim is redirected to a benign page (line 16) if the fingerprinting or evasion stage fails. Attackers can also abuse the function `setTimeout` to create a time bomb [65] to evade detection. Detection systems can not afford to wait for long

periods of time during the analysis of each sample in an attempt to capture randomly triggered exploits.

Challenges and Existing Techniques Static analysis is a powerful technique that explores all paths of execution. But, one particular issue that plagues static analysis of malicious JavaScript is that not all of the code can be statically observed. For example, static analysis cannot observe malicious code hidden within `eval` strings, which are frequently exploited by attackers to obfuscate their code. Therefore, current detection approaches [5–7] rely upon dynamic analysis to expose features typically seen within malicious JavaScript. More specifically, these approaches rely upon visiting websites or opening PDF files with an instrumented browser or PDF reader, and then monitoring different features (`eval` strings [7], heap health [10], etc.) for detection.

However, dynamic analysis techniques suffer from two fundamental limitations. The first limitation is limited code coverage. This becomes a much more severe limitation within the context of analyzing malicious JavaScript. Attackers frequently employ a technique called $cloaking$ [66], which works by fingerprinting the victim's web browser and only revealing the malicious content when the victim is using a specific version of the browser with a vulnerable plugin. Cloaking makes dynamic analysis much harder because the sample must be run within every combination of web browser and plugin to ensure complete code coverage. The widely-used event callback feature of JavaScript also makes it challenging for dynamic analysis to automatically trigger code. For example, attackers can load the attack code only when a specific mouse click event is captured, and automatically determining and generating such a trigger event is difficult.

The second limitation is the complexity of the JavaScript runtime environment. JavaScript is used within many applications, and it can call the functionality of any plugin extensions supported by these applications. For dynamic analysis, any pre-defined browser setup handles a known set of browsers and plugins. Thus, there is no guarantee that this setup will detect vulnerabilities only present in less popular plugins. While it is possible to deploy a cluster of machines running many different operating systems, browser applications, and browser plugins, the exponential growth of possible combinations rapidly causes scalability issues and makes this approach infeasible.

Rozzle [9] attempts to address this code coverage problem by exploring environment-related paths within a single execution. For instance, because `att` in Figure 3.1 depends upon the environment-related API's output, Rozzle will execute lines 5-15 and reveal the malicious behaviors hidden in lines 8-14 by executing both the `try` and `catch` blocks. But, it requires a predefined environment-related profile for path exploration. Construction of a complete profile is a challenging task because of the numerous different browsers and plugins, especially for newer proposed fingerprinting techniques [18, 59, 60]. These new techniques do not rely upon any specific APIs. For instance, the JavaScript engine fingerprinting technique [60] relies upon JavaScript conformance tests such as the Sputnik [67] test suite to determine a specific browser and major version number. There are no specific APIs used for the fingerprinting. Thus, Rozzle cannot include it within the predefined profile and explore the environment-related paths. Rozzle also introduces runtime errors into the analysis engine, which may stop the analysis before any malicious code is executed. In contrast, JSFORCE does not rely upon predefined profile for path exploration and handles runtime errors using the forced execution model presented in Section 3.3.1. By overcoming those limitations of Rozzle, JSFORCE achieves greater code coverage.

Revolver [12] employs a machine learning-based detection algorithm to identify evasive JavaScript malware. However, it requires that the malicious sample is present within a known sample set so that its evasive version can be determined based upon the classification difference. By design, it can not be used for 0-day malware detection.

Symbolic execution has also been applied to the task of exposing malware [65]. This technique, while improving code coverage over dynamic analysis, suffers from scalability challenges and is, in many ways, unnecessarily precise [9]. Within the context of JavaScript analysis, symbolic execution becomes more challenging [19]. JavaScript applications accept many different kinds of input, and those inputs are structured as strings. For example, a typical application might take user input from form fields, messages from a server via `XMLHttpRequest`, and data from code running concurrently within other browser windows. It is extremely difficult for a symbolic string solver [61] to effectively supply values for all of these different kinds of inputs and reason about how those inputs are parsed and validated. The rapidly evolving JavaScript language and its host programs (browsers, PDF readers, etc.) make the modeling of the JavaScript API tedious work. Furthermore, the dynamic features (such as the `eval` function) of JavaScript make symbolic execution infeasible for many analysis efforts.

Overview JSFORCE, our proposed forced-execution engine for JavaScript, is an enhancement technology designed to better expose the behaviors of malicious JavaScript at runtime. Different detection policies can be applied to examine malicious JavaScript. While the forced execution concept is first introduced for binary code analysis (X-Force [62]), we face unique challenges, such as type inference and invalid object access recovery, in enabling the forced execution concept for JavaScript.

We now illustrate how the forced execution of JavaScript code works. Consider the snippet shown in Figure 3.1. JSFORCE forces the execution through the different code paths of the snippet. So, the exploitation code within the `catch` block (lines 7-14) will be executed, no matter how the ActiveX API is simulated by the emulation-based analysis system. Moreover, JSFORCE will immediately invoke the callback function passed to `setTimeout` to trigger the time bomb malware.

JSFORCE's path exploration forces line 2 to be executed, regardless of the result of the fingerprinting statement (line 1). Since `btt` is not defined within the code snippet under analysis, which is a common scenario because collected JavaScript code may be incomplete due to multi-stages of the attack, the execution of line 2 raises a `ReferenceError` exception when running within a normal JavaScript engine. When the exception is captured, JSFORCE creates a `FakedObject` named `btt`, which is fed to the JavaScript engine to recover from the invalid object access. However, the type of `btt` is unknown at the time of `FakedObject`'s creation. JSFORCE infers the type based upon how the `FakedObject` is used. For example, if this `FakedObject` is added to an integer, JSFORCE will then change its type from `FakedObject` to `Integer`. We call this *faked object retyping*.

3.3 JavaScript Forced Execution

This section explains the basics of how a single forced execution proceeds. The goal is to have a non-crashable execution. We first present the JavaScript language semantics and then focus on how to detect and recover from invalid object accesses. We then discuss how path exploration occurs during forced execution.

3.3.1 Forced Execution Semantics

⟨*EXPRESSIONS*⟩ ::= c	CONSTANT
| x	VARIABLE
| $x.f$	FIELD ACCESS
| $x.prot$	PROTO ACCESS
| e *op* e	BINARY OP
| $this$	THIS
| $\{f_1 : e_1, ..., f_n : e_n\}$	OBJECT LITERAL
| {function($p_1, ..., p_n$){S}}	FUNCTION DEF
| f($a_1, ..., a_n$)	FUNCTION CALL
| new f($a_1, ..., a_n$)	NEW
⟨*STATEMENTS*⟩ ::= skip	SKIP
| $S_1 : S_2$	SEQ
| $varx$	VAR DECL
| $x := e$	ASSIGN
| $x.f := e$	ASSIGN
| if e then S_1 else S_2	CONDITIONAL
| while e do S	WHILE
| try{S}catch{S}finally{S}	TRY CATCH
| return e	RETURN

Fig. 3.2.: Core JavaScript

The JavaScript Language JavaScript is a high-level, dynamic, untyped, and interpreted programming language. Figure 3.2 summarizes the syntax of the core JavaScript, which captures the essence of JavaScript. At runtime, the JavaScript engine dynamically interprets JavaScript code to 1) load/allocate objects, 2) determine the types of objects, and 3) execute the corresponding semantics. Given an arbitrary JavaScript snippet, execution may fail because of undefined/uninitialized objects or incorrect object types. For instance, the execution of line 2 in Figure 3.1 raises a `ReferenceError` exception because `btt` is not defined. To tolerate such invalid object accesses, forced execution must handle such failures.

$$
\begin{array}{rcl}
\text{Types:} & & \\
\tau & ::= & \sum_{i \in T, T \subseteq \{\bot,u,b,s,n,o\}} \varphi_i \\
\text{Rows:} & & \\
\varrho & ::= & \mathit{str}{:}\tau, \varrho \\
 & | & \varrho_\tau \\
\text{Type environments:} & & \\
\Gamma & ::= & \Gamma(x:\tau) \\
 & | & \varnothing \\
\text{Type summands} & & \\
\text{and indices:} & & \\
\varphi_\bot & ::= & \text{Undef} \\
\varphi_u & ::= & \text{Null} \\
\varphi_b & ::= & \text{Bool}(\xi_b) \\
\xi_b & ::= & false \mid true \mid \top \\
\varphi_s & ::= & \text{String}(\xi_s) \\
\xi_s & ::= & str \mid \top \\
\varphi_n & ::= & \text{Number}(\xi_n) \\
\xi_n & ::= & num \mid \top \\
\varphi_f & ::= & \text{Function}(this:\tau; \varrho \to \tau) \\
\varphi_o & ::= & \text{Obj}(\sum_{i \in T, T \subseteq \{b,s,n,f,\bot\}} \varphi_i)(\varrho) \\
\varphi_{fo} & ::= & \text{FObj} \\
\varphi_{ff} & ::= & \text{FFun}
\end{array}
$$

Fig. 3.3.: Syntax of JavaScript Types

The basic idea behind forced execution is that, whenever a reference error is discovered, a `FakedObject` is created and returned as the pointer of the property. During the execution of the program, the expected type of the `FakedObject` is indicated by the involved operation. For instance, adding a `number` object to a `FakedObject` indicates that the `FakedObject`'s type is `number`. When the type of a `FakedObject` can be determined, we update it to the corresponding type.

Potentially, we could assign `FakedObject` with the type `Object` and reuse the dynamic typing rules of the JavaScript engine to coerce the `FakedObject` to an expected type. Nevertheless, the dynamic typing rules of the JavaScript engine are designed to maintain the

correctness of JavaScript semantics and do not suffice to meet our analysis goal of achieving maximized execution. This can be attributed to two reasons. First, while the JavaScript engine can cast the `FakedObject:Object` to proper primitive values, it cannot cast the `FakedObject:Object` to proper object types. For instance, when a `FakedObject` with the type `Object` is used as a function object, the JavaScript engine will raise the `TypeError` exception according to ECMA specification [68]. Second, the casting of `FakedObject` to primitive values by the JavaScript engine can lead to unnecessary loss of precision. To understand why, consider the following loop:

```
1 c = a/2;
2 for(i= c;i<10000;i++)
3 {
4   memory[i] = nop + nop + shellcode;
5 }
```

Since `a` is not defined, a `FakedObject` will be created. With the built-in typing rule of the JavaScript engine, `c` will be assigned the value `NaN`. The loop condition $i < 10000$ will always evaluate to false. Thus, the loop body, which contains the heap spray code, will never be executed. Although the path exploration of JSFORCE will guarantee that the loop body will be executed once, without executing the loop 10,000 times, it will likely be missed by heap spray detection tools because of the small chunk of memory allocated on the heap.

Therefore, to overcome the above two issues, JSFORCE introduces two new types, `FObj` and `FFun`, to the JavaScript type system. The JavaScript type system defined in [69] is extended to

```
1  var a = null;
2  var b = c + 1;
3  var d = a.length;
4  var func = null;
5  a = "Hello World";
6  var e = new abc();
7  if (b < 5) {
8      func = function(x) {
9          return x
10     };
11 }
12 d = func(6);
13 var f = Math.abs(d);
14 array[5] = f;
```

Fig. 3.4.: JavaScript Sample

support these two new types. Figure 3.3 summarizes the new syntax of these JavaScript types. Type `FObj` is for `FakedObject`. At the moment `FakedObject` is created, we assign type `FObj` as the temporary type of `FakedObject`. It can be subtyped to any types within the JavaScript type system. When `FakedObject` is used as a function object, `FakedObject` is casted to `FakedFunction` with type `FFun`. The `FakedFunction` with type `FFun` can take arbitrary input and always returns `FakedObject:FObj`. Following JSFORCE's dynamic typing rules, `a` in the above `loop` sample will be typed to `Number` because it is used as a dividend. `c` is then assigned to `Number` and the loop body is executed repeatedly until the loop condition `i<10000` is evaluated to false. By introducing these two new types and their typing rules, JSFORCE solves the two issues mentioned in the above paragraph. In the following paragraphs, we detail the JavaScript forced execution model.

Reference Error Recovery To avoid raising `ReferenceError` exceptions, we introduce the `FakedObject` and recover the error by creating the `FakedObject` whenever necessary. There are two cases that lead to reference errors. The first case (ER_1) is a failed object lookup. Every field access or prototype access triggers a dynamic lookup using the field or prototype's name as the key. If no object is found, the lookup fails. Such failures happen when the running

environment is incomplete or some portion of the JavaScript code is missing. For example, a browser plugin referenced by the JavaScript is not installed, or only a portion of the JavaScript code is captured during the attack.

To handle this error, JSFORCE intercepts the lookup process and a `FakedObject` named as the lookup key is created whenever a failed lookup is captured. The corresponding parent object's property is also updated to the `FakedObject`. Line 2 in Figure 3.4 presents such an example. The JavaScript engine searches the current code scope for the definition of `c`, which is not defined. JSFORCE returns the `FakedObject` as the temporary value of `c` so that the execution can continue.

The second case (ER_2) occurs when the object is initialized to the value `null` or `undefined`, but later has its properties accessed. JSFORCE modifies the initialization process to replace the `null` to a `FakedObject` if an object is initialized as value `null` or `undefined`. For example, the variable `a` defined on line 1 in Figure 3.4 is assigned the value `FakedObject` instead of `null` under the forced execution engine. The variable `a` may later be updated to another value during execution, but this does not sabotage the execution of JavaScript code.

Faked Object Retyping When a `FakedObject` is used within an expression, it must be retyped to the expected type. Otherwise, incorrect typing raises a `TypeError` exception and stops the execution. JSFORCE infers the expected type of `FakedObject` by how the `FakedObject` is used. Figure 3.5 summarizes the dynamic typing rules introduced by JSFORCE. The rules are divided into the following five categories:

1) *R-ASSIGN*. This rule deals with assignment statements. When a `FakedObject` e_0 is assigned to a new value e_1, e_0 is updated to the new value $e1$ with the type τ. The

R-ASSIGN

$$\frac{\Gamma \vdash_{hs} e_0 : \varphi_{fo} \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash_{ref} e_0 = e_1 : \tau}$$

R-CALL1

$$\frac{\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rfloor : \tau_1, ..., \lceil n\text{–}1 \rfloor : \tau_n, \varrho \to \tau))(\varrho') \qquad \Gamma \vdash_{ref} e_0 : \varphi_{fo}/\tau}{\vdash_{upd} e_0 : \varphi_{fo}, \varrho'@\tau \leftarrowtail \varphi_{ff}, \Gamma \vdash_{ref} e_0(e_1, ..., e_n) : \varphi_{fo}/\perp}$$

R-CALL2

$$\frac{\begin{array}{c}\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rfloor : \tau_1, ..., \lceil n\text{–}1 \rfloor : \tau_n, \varrho \to \tau))(\varrho') \\ \Gamma \vdash_{ref} e_0 : \tau_0/\tau'\end{array} \qquad \begin{array}{ccccccc}\Gamma \vdash e_1 : \tau_1 & ...\Gamma \vdash e_{(i-1)} : \tau_{(i-1)} & \Gamma \vdash e_i : \varphi_{fo} & \Gamma \vdash e_{(i+1)} : \tau_{(i+1)} & ... & \Gamma \vdash e_n : \tau_n\end{array}}{\vdash_{upd} e_i : \varphi_{fo}@\tau \leftarrowtail \tau_i, \Gamma \vdash_{ref} e_0(e_1, ..., e_n) : \tau/\perp}$$

R-NEW

$$\frac{\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rfloor : \tau_1, ..., \lceil n\text{–}1 \rfloor : \tau_n, \varrho \to \tau))(\varrho') \qquad \Gamma \vdash_{ref} e_0 : \varphi_{fo}/\tau}{\vdash_{upd} e_0 : \varphi_{fo}, \varrho'@\tau \leftarrowtail \varphi_{ff}, \Gamma \vdash_{ref} new\ e_0(e_1, ..., e_n) : \varphi_{fo}/\perp}$$

R-BINOPERATOR1

$$\frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \Gamma \vdash e_2 : \tau' \qquad \neg(e_2\ is\ \varphi_{fo})}{\vdash_{upd} e_1 : \varphi_{fo}@\tau \leftarrowtail \tau', \Gamma \vdash e_1\ op\ e_2 : \tau'}$$

R-BINOPERATOR2

$$\frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \Gamma \vdash e_2 : \varphi_{fo}}{\vdash_{upd} e_1 : \varphi_{fo}@\tau \leftarrowtail \varphi_n, \vdash_{upd} e_2 : \varphi_{fo}@\tau \leftarrowtail \varphi_n, \Gamma \vdash e_1\ op\ e_2 : \tau}$$

R-INDEX1

$$\frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \tau_1 \trianglerighteq Obj(\varphi_1)(\varrho_1) \qquad \Gamma \vdash e_2 : \varphi_n}{\vdash_{upd} e_1 : \varphi_{fo}@\tau \leftarrowtail \tau_1, \Gamma \vdash_{hs} e_1[e_2] : \varphi_{fo}}$$

R_UNARYOPERATOR

$$\frac{\Gamma \vdash e_1 : \varphi_{fo}}{\vdash_{upd} e_2 : \varphi_{fo}@\tau \leftarrowtail \varphi_n, \Gamma \vdash op\ e_1 : \tau}$$

R-INDEX2

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \trianglerighteq Obj(\varphi_1)(\varrho_1) \qquad \Gamma \vdash e_2 : \varphi_{fo} \qquad \vdash_{upd} \varrho_1@\varphi_n \mapsto \tau'}{\vdash_{upd} e_2 : \varphi_{fo}@\tau \leftarrowtail \varphi_n, \Gamma \vdash_{hs} e_1[e_2] : \tau'}$$

Fig. 3.5.: Typing rules

JavaScript engine handles this naturally, so no interference is required. For example, variable `a` in Figure 3.4 is assigned `FakedObject` at line 1 by JSFORCE. At line 4, the variable `a` is retyped as a `string` object.

2) *R-CALL1* and *R-NEW*. These two rules describe the typing rule for the scenario when a `FakedObject:FObj` is used as a function call or by the `new` expression. Function calls and the `new` expression both expect their first operand to evaluate to a function. So, JSFORCE updates the `FakedObject:FObj` to `FakedFunction:FFun` for this situation. The `FakedFunction` is a special function object which is configured to accept

arbitrary parameters. The return value of the function is set to a `FakedObject:FObj` so that it can be retyped whenever necessary.

3) *R-CALL2*. This rule describes the case where the callee is a known function, but a `FakedObject:FObj` is passed as a function parameter. JSFORCE types the `FakedObject:FObj` to the required type of the callee's arguments. The JavaScript language has many standard built-in libraries such as `Math` and `Date`. When a `FakedObject:FObj` is used by the standard library function, we update the type based upon the specification of the library function [68]. Currently, JSFORCE implements retyping for several common libraries (e.g., `Math`, `Number`, `Date`).

4) *R-BINOPERATOR1/2* and *R-UNARYOPERATOR*. These three rules describe how to update the type if the `FakedObject:FObj` is involved in an expression with an operator. JSFORCE updates the `FakedObject:FObj`'s type based upon the semantics of the operator. For unary operators, it is straightforward to determine the type from the operator's semantics. For instance, the postfix operator indicates the type as `number`. For binary operators, the typing becomes more complicated. If both operands are `FakedObject:FObj` and the operator does not reveal the type of the operands, JSFORCE types them to `number`. This is because the `number` type can be converted to most types naturally by the JavaScript engine. For example, the `number` type in JavaScript can be converted to the `string` type, but it may fail to convert a `string` to a `number`. Later during execution, if the types can be determined, JSFORCE will update the type to the correct type. If only one of the two operands is `FakedObject:FObj`, JSFORCE determines the type based upon the other operand's type and the operator's semantics.

Statement	Action	Rule
1: var a = null;	$a \leftarrow FakedObject$	ER_2
2: var b = c + 1;	$c \leftarrow FakedObject$	ER_1
	$c \leftarrow RanNumber$	R_BINOPERATOR1
3: var d = a.length;	$a.length \leftarrow FakedObject$	ER_1
4: var func = null;	$func \leftarrow FakedObject$	ER_2
5: a = "Hello World";	$a \leftarrow "HelloWorld"$	R_ASSIGN
6: var e = new abc();	$abc \leftarrow FakedObject$	ER_1
	$abc \leftarrow fakedFunction$	R_NEW
7: if(b <5)	NO ACTION	NONE
12: d = func(6)	$func \leftarrow fakedFunction$	R_CALL1
	$d \leftarrow FakedObject$	R_ASSIGN
13: var f = Math.abs(d)	$d \leftarrow RanNumber$	R_CALL2
14: array[5] = f;	$array \leftarrow FakedObject$	ER_1
	$array \leftarrow arrayObject$	R_INDEX1
	$array[5] \leftarrow f$	R_ASSIGN

Table 3.1: Forced execution of sample in Figure 3.4

5) *R-INDEX1* and *R-INDEX2*. These two rules describe how to update the type when there are indexing operations. A `FakedObject:FObj` is updated to an $ArrayObject : \phi_o$ whenever a key is used as an array index to access elements of the `FakedObject`. JSFORCE creates an $ArrayObject$ and initializes the elements to `FakedObject:FObj`. The length of the $ArrayObject$ is set to 2*$CurrentIndex$. If an Out-Of-Boundary access is found, JSFORCE doubles the length of $ArrayObject$. If the array index is `FakedObject`, JSFORCE types it to `number` and initializes it as `0`, which avoids Out-Of-Boundary exceptions. If both the index object and base object are `FakedObject:FObj`, the R-INDEX2 rule is first applied to update the index object to `number`, then the R-INDEX1 rule is applied to update the base object to $ArrayObject$.

Example Table 3.1 presents a forced execution of the sample shown in Figure 3.4. In the execution, the branch in lines 8-11 is not taken. At line 1, JSFORCE assigns a `FakedObject:Fobj` to `a`, instead of `null`. This is because at line 3 the access to property `length` raises an exception if `a` is `null`. On line 2, we can see a `FakedObject:FObj` is first assigned to `c`. Once `c` is added to `1`, JSFORCE updates the value of `c` to a random number. Lines 6 and 7 show that if a `FakedObject:FObj` is used in the function call or `new` expression, JSFORCE updates it to `FakedFunction:FFun`. The return value of the faked function is still configured to `FakedObject:FObj`, so that at line 13, `d` is updated to hold a random number.

JSFORCE also automatically recovers from other exceptions by intercepting those exceptions to eliminate the exception condition. For example, JSFORCE will update a divisor to a non-zero value if a division-by-zero exception is raised.

3.3.2 Path Exploration in JSFORCE

One important functionality of JSFORCE is the capability of exploring different execution paths of a given JavaScript snippet to expose its behavior and acquire complete analysis results. In this subsection, we explain the path exploration algorithm and strategies.

In practice, attackers constantly adopt the dynamic features of JavaScript to aid in evading detection. This results in incomplete path exploration under two circumstances. The first is when strings are dynamically generated. For instance, `document.write` is often abused to inject dynamically decoded malicious JavaScript code into the page at runtime. The second is when event callbacks are used. As discussed in Section 3.2, attackers can abuse event callbacks to stop the execution of malicious code. JSFORCE solves this by employing specific path exploration

Algorithm 1 Path Exploration Algorithm

Definitions: $switches$ - the set of switched predicates in a forced execution, denoted by a sequence of predicate offsets in the source file(SrcName:offset). For example, $t.js : 15 \cdot t.js : 83 \cdot t.js : 100$ means the branch in source file $t.js$ with the offset 15, 83, 100 is switched. EX, WL - a set of forced executions, each denoted by a sequence of switched predicates. $preds : \overline{Predicate \times boolean}$ - the sequence of executed predicates.

Input: The tested JS
Output: $FULL\_EX$

```
1:  FULL_EX ← ∅
2:  SRC ← {JS}
3:  while SRC do
4:      WL ← {∅}
5:      EX ← ∅
6:      js ← SRC.pop()
7:      while WL do
8:          switches ← WL.pop()
9:          EX ← EX ∪ switches
10:         (preds, newJS) ← ExecuteCode(js, switches)
11:         SRC ← SRC ∪ newJS
12:         t ← len(switches)
13:         preds ← remove the first t elements in preds
14:         for all (p, b) ∈ preds do
15:             if !covered(p, ¬b) then
16:                 WL ← WL ∪ switches · (p, b)
17:             end if
18:         end for
19:     end while
20:     FULL_EX ← FULL_EX ∪ {EX : js}
21: end while
22: procedure ExecuteCode(JS, switches)
23:     preds ← switches
24:     CBQ ← ∅
25:     newJS ← ∅
26:     for all stmt ∈ JS do
27:         if isNoneEvalFunctionCallStmt(stmt) then
28:             if CalleeTakesStrings(stmt) then
29:                 newJS ← newJS ∪ GetJSFromString(stmt)
30:             end if
31:             if CalleeRegisterCallback(stmt) then
32:                 CBQ ← CBQ ∪ ExtractCBFunc(stmt)
33:             end if
34:         else if isBranchStmt(stmt) then
35:             if GetSwitch(stmt) ∈ switches then
36:                 Execute according to switches
37:             else
38:                 preds ← preds · GetPredicate(stmt)
39:             end if
40:         end if
41:     end for
42:     for all cb ∈ CBQ do
43:         (preds′, newJS′) ← ExecuteCode(cb, ∅)
44:         newJS ← newJS ∪ newJS′
45:         preds ← preds · preds′
46:     end for
        return (preds, newJS)
47: end procedure
```

strategies. Within the execution, if faked functions take strings as input, JSFORCE examines the strings and executes the code if they contain JavaScript. This strategy is only applied on faked functions since original functions (`eval`) can handle the strings as defined. JSFORCE also detects the callback registration function and invokes the callback function immediately after the current execution terminates.

JSFORCE treats `try-catch` statements as `if-else` statements, ie., it executes each `try` block and `catch` block separately. Ternary operators are also treated as `if-else` statements: both values are evaluated.

There are several different path exploration algorithms: linear search, quadratic search, and exponential search [62]. The goal of path exploration in JSFORCE is to maximize the code coverage to improve the detection rate of malicious payload with an acceptable performance overhead. Quadratic and exponential searches are too expensive, so JSFORCE employs the linear search only.

Algorithm 1 describes the path exploration algorithm, which generates a pool of forced executions that achieve maximized code coverage. The complexity is $O(n)$, where n is the number of JavaScript statements. n may change at runtime because JavaScript code can be dynamically generated. Initially, JSFORCE executes the program without switching any predicates since `switches` is initialized as $\varnothing$ (line 8) for the first time. JSFORCE executes the program according to the `switches` at line 10 and returns `preds` and dynamically generated code `newJS`. In lines 12-17, we determine if it would be of interest to further switch more predicate instances. Lines 11-13 compute the sequence of predicate instances eligible for switching. Note that it cannot be a predicate before the last switched predicate specified in `switches`. Switching such a predicate may change the control flow such that the specification

in `switches` becomes invalid. Specifically, line 16 switches the predicate if the other branch has not been covered. In each new forced execution, we essentially switch one more predicate.

The procedure `ExecuteCode` (lines 22-47) describes the execution process. It collects dynamically generated JavaScript code (lines 28-30) and the executed predicates (lines 34-38). The new generated JavaScript code, `newJS`, will be executed after the path exploration of the current `js` finishes. The registered callback functions (lines 31-33) are also queued and invoked after the current execution finishes (lines 42-46). As an example, recall the callback function `redir()` used in line 16 of Figure 3.1. Instead of waiting for the timeout, JSFORCE will trigger the `redir()` function immediately after the current execution finishes.

3.4 Implementation

JSFORCE is implemented by extending the V8 JavaScript engine [64] on the X86-64 platform. It is comprised of approximately 4,600 lines of C/C++ code and 1,500 lines of Python code. We address some prominent challenges of its implementation in this section.

Reference Error Recovery & Faked Object Retyping In V8, an abstract syntax tree (AST) is generated for every function, which is then compiled into native code (known as Just-In-Time code). V8 adopts an inline caching technique [70] to accelerate property accesses. If the property access fails, the execution jumps to the V8 runtime system which handles any inline cache misses. If the runtime system is unable to handle an inline cache miss, either due to reference error or type check error, it raises the corresponding exception and stops the execution.

We modify the inline cache miss handling process to enable reference error recovery and faked object retyping. For reference error recovery, JSFORCE creates and returns the `FakedObject`

for failed object lookup by changing the V8 property access failure handling functions like `Runtime_LoadIC_Miss`. For faked object retyping, JSFORCE inserts additional code into runtime methods like `Runtime_BinaryOpIC_Miss` that is executed prior to the exception being raised. This additional code follows the rules described in Section 3.3.1 to conduct the retyping process if the involved operation contains a `FakedObject`.

Predicates Flip We have two approaches available to flip the predicates. The first approach is to flip the predicates within the Just-in-Time code. The Just-in-Time code can be optimized (inline caching, etc.) by V8 in accordance with the execution profile. To enable predicates flipping, a runtime function must be inserted before every branch so that JSFORCE can manipulate the predicate value. This approach may affect the optimization process of Just-in-Time code.

JSFORCE takes the second approach: if the branch A of a predicate needs to be taken, JSFORCE replaces the other branch with this branch A. At runtime, no matter which branch is taken, the branch A is executed. For instance, we want to take the $\{A\}$ branch of the statement `if(e)`$\{A\}$`else`$\{B\}$. JSFORCE changes it to `if(e)`$\{A\}$`else`$\{A\}$, so that $\{A\}$ is executed at runtime.

Loops and Recursions Sometimes, JSFORCE may cause a loop to execute for a very long time, due to the introduction of faked objects. To solve this problem, JSFORCE inserts a time counter for every loop statement (`for...in` and `for...of` are excluded, as they will always terminate), and it will terminate the loop if the execution time exceeds a limit. Similarly, if JSFORCE forces a predicate that guards the termination of a recursive function call, a very deep recursion may result. To address deep recursion, JSFORCE monitors the stack depth. Once the

maximum call stack size (defined by V8) is reached, calls to that function are omitted by JSFORCE.

3.5 Evaluation

In this section, we present details on the evaluation of correctness, effectiveness and runtime performance of JSFORCE using a large number of real-world samples.

3.5.1 Dataset & Experiment Setup

Dataset The complete dataset used for our evaluation consists of two sample sets: a malicious sample set and a benign sample set. For the malicious set, we collected a sample set with 172,995 HTML files and 23,509 PDF files from various databases including VirusTotal [71], Contagio [72], MalTrafficAnalysis [73], and Threatglass [74]. Among those, all samples from VirusTotal were new samples evaluated within a month of being submitted, with the samples provided from other sources being relatively old. For the benign sample set, we crawled the Alexa top 100 websites [75] and collected 47,592 HTML files.

Experiment Setup For JavaScript code analysis, we leverage the jsunpack [7] tool. Jsunpack is a widely used malicious JavaScript code analysis tool that utilizes the SpiderMonkey [76] JavaScript engine for code execution. Six distinct configurations are predefined within jsunpack to maximize the exploration of JavaScript code by trying different browsers and language settings. For the sake of our evaluation, we replaced the SpiderMonkey from jsunpack with JSFORCE and relied upon the detection policies in jsunpack for malicious code detection. Most

Category	Total	Detected by JSFORCE	Percentage
True Positive	389	389	100%
False Positive	47,592	9	0.019%

Table 3.2: Correctness Results.

of our experiments are based upon the comparison between the original jsunpack and the JSFORCE-extended jsunpack. Note that the experiments performed within this section are only intended to show the *improvement of detection results* over the original ones when adopting JSFORCE. The detection policy itself is another important research topic which is orthogonal to the focus of JSFORCE. We conducted our experiments on a test machine equipped with Intel(R) Xeon(R) E5-2650 CPU (20M Cache, 2GHz) and 128GB of physical memory. The operating system was Ubuntu 12.04.3 (64bit).

3.5.2 Correctness

In this section, we evaluate the correctness of the analysis result for JSFORCE. The goals of this evaluation are two-fold. First, we wish to know the true positive rate of our analysis results, meaning that we wish to verify whether a JavaScript program is undoubtedly malicious if it is tagged as one by the analysis tools. Second, we wish to understand any false positives in the results so as to determine whether any benign JavaScript code can be mistakenly labeled as malicious.

True positive With our first goal in mind, we queried VirusTotal [71] for malicious HTML files and collected 389 samples which are precisely labeled with specific CVE (Common Vulnerabilities and Exposures) numbers that match CVEs listed in jsunpack. Furthermore, we

manually reviewed each of the samples and confirmed the existence of shellcode or malicious signatures. This step is to guarantee all the samples we tested are real malicious samples that should be detected by our tool. Then, we analyzed the samples using jsunpack with JSFORCE. The experimental result is listed in the first row of Table 3.2 as "true positive". It shows that JSFORCE could successfully detect all of the samples, resulting in a 100% true positive rate. To better understand these results, we further inspected the detailed analysis results to see why our tool tagged samples as malicious. Our inspection results revealed that all of the payload and malicious signatures extracted by the JSFORCE are indeed malicious, proving that our tool can achieve very high true positive rate with accurate analysis details.

False positive For our second goal, we analyzed our benign sample set using JSFORCE and then observed whether any of the samples could be incorrectly labeled as malicious. As shown in the second row of Table 3.2, the JSFORCE tags 9 out of 47,592 samples as malicious. We first manually confirmed that all 9 samples are clean and thereupon study why the false positives happen. It has been verified by manual inspection that all of the false positives are caused by the inaccurate detection policy, to be more specific, the over-relaxation of the shellcode string matching policy enforced by jsunpack. The reason why our tool could detect them as malicious is that it explores JavaScript code in a more complete fashion in consequence of our forced execution technique. Therefore, based upon the above experimental results, we argue that using JSFORCE will keep a very low false positive rate for JavaScript code analysis, and is able to assist in accomplishing more thorough results. Theoretically, JSFORCE can generate higher code coverage than jsunpack and lead to better analysis results. But, the question is by how much. With that, we conducted another set of experiments to show the effectiveness of JSFORCE.

Sample Set	**Total**	**without** JSFORCE	**with** JSFORCE	**Improvement**	**Detected By Both**	**Missed With JSFORCE**
Old HTML	66,325	193	357	84.9%	193	0
New HTML	106,018	2,250	20,649	817.3%	2250	0
HTML Total	**172,995**	**2,443**	**21,006**	**759.8%**	**2443**	**0**
Old PDF	22,081	6,306	6,475	2.7%	6306	0
New PDF	1,428	32	170	431.2%	32	0
PDF Total	**23,509**	**6,338**	**6,645**	**4.8%**	**6338**	**0**

Table 3.3: Effectiveness Results.

3.5.3 Effectiveness

For the evaluation of effectiveness, we would like to demonstrate that JSFORCE can indeed help the malicious JavaScript code analysis by performing efficient forced execution. In order to achieve that, we utilize our malicious HTML and PDF sample sets and run the sample sets against jsunpack both with or without JSFORCE for the evaluation. In the interest of showing how useful our faked object retyping is, we also conduct another experiment that disables the retyping and only keeps the reference error recovery component and path exploration component.

Experimental Results Table 5.2 illustrates the experimental results for effectiveness. It demonstrates that JSFORCE could greatly improve the detection rate for JavaScript analysis. We can see detection rate improvements of 759.84% and 4.84% for HTML and PDF samples, respectively, when using JSFORCE-extended jsunpack instead of the original version for analysis. And all the samples detected by original jsunpack are also flagged by JSFORCE-extended jsunpack. We further break down the numbers into old and new sample sets and perceive that the extended version could perform much better than original jsunpack in analyzing new samples. For new HTML samples, jsunpack with JSFORCE is able to detect 817.3% more samples while

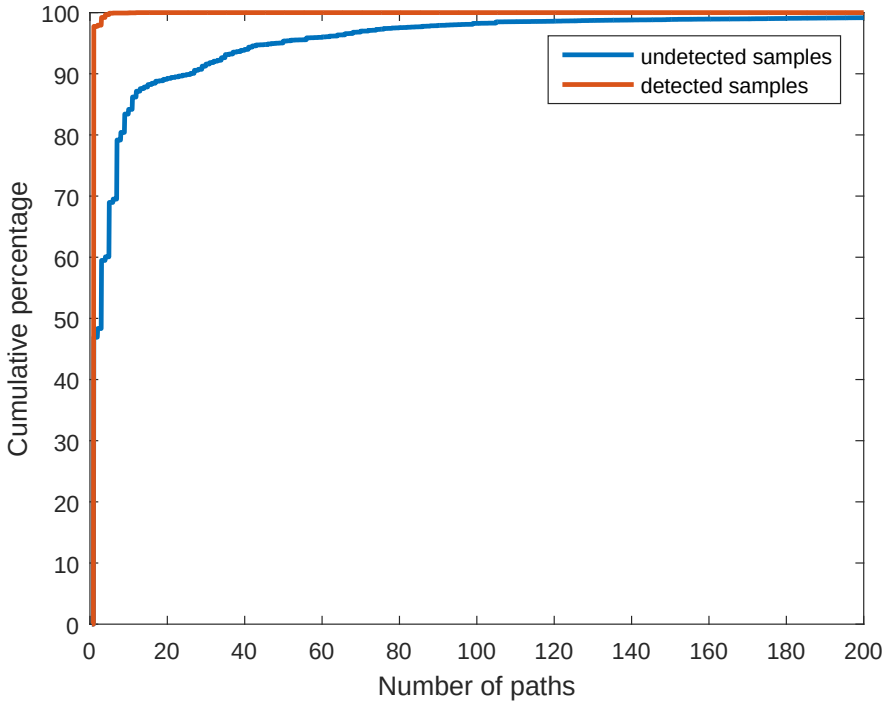

Fig. 3.6.: Num of Path Exploration during Analysis.

for old samples, the number is 84.97%. Similar results are also observed for PDF samples. After manual inspection, we confirmed that this is because many of the old samples have been analyzed for quite sometime and jsunpack already has the signatures stored in its database, leaving only a small margin for JSFORCE to improve upon. For the faked object retyping evaluation, we reran the test using 106,018 new HTML malicious samples with retyping component disabled. The result shows that only 8,677 samples can be detected by JSFORCE in contrast to 20,649 with retyping enabled. This result reveals the usefulness of our faked object retyping component during analysis. Nevertheless, through our experiments, we are able to draw the conclusion that JSFORCE is quite effective for boosting the effectiveness of JavaScript analysis.

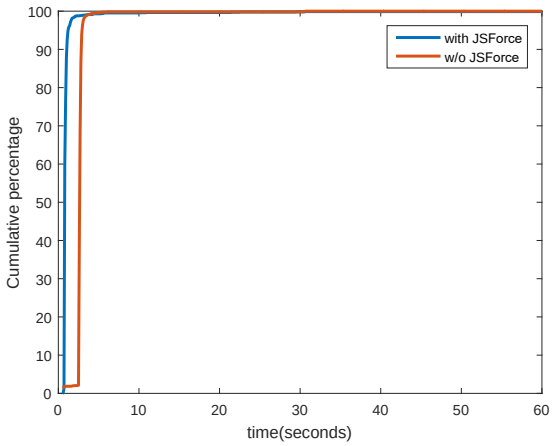

Fig. 3.7.: Runtime for Detected HTML samples.

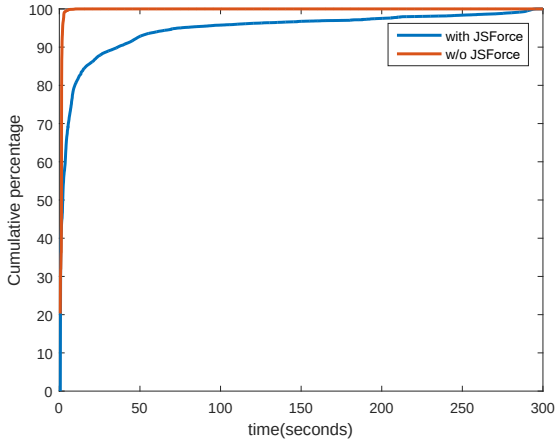

Fig. 3.8.: Runtime for Undetected HTML samples.

Number of Paths Explored Potentially, there may be a large number of paths that exist inside of a single JavaScript program. The effectiveness and efficiency of JSFORCE are closely related to the number of paths explored during analysis. Hence, we would like to show some statistics on the number of paths that JSFORCE explored during analysis.

The result depicted in Figure 3.6 shows that JSFORCE is able to detect the maliciousness of samples with a limited number of path explorations. An interesting observation is that over 96% of the samples were detected by exploring only a single path. Even though most of the analysis for detected samples can be finished by exploring just one path, the path exploration of JSFORCE is still essential. Note that 98% of the samples missed by the default jsunpack, but detected by the JSFORCE-extended version, explore at least two paths. So, the analysis could still receive an enormous benefit from JSFORCE in terms of path exploration. Please refer to the Section 6 Case Study for more details on this topic. As for any undetected samples, JSFORCE will explore the entire code space during analysis, which requires a larger amount of path exploration and longer analysis runtime.

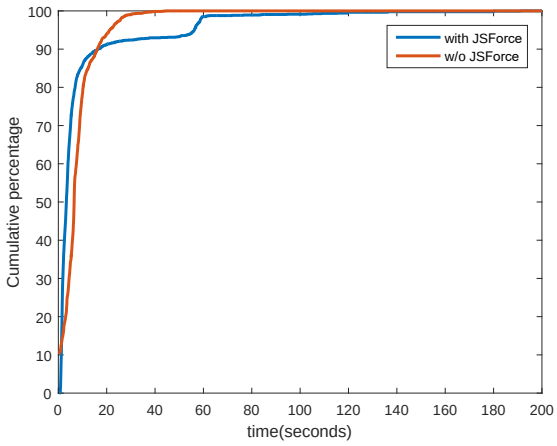

Fig. 3.9.: Runtime for Detected PDF samples.

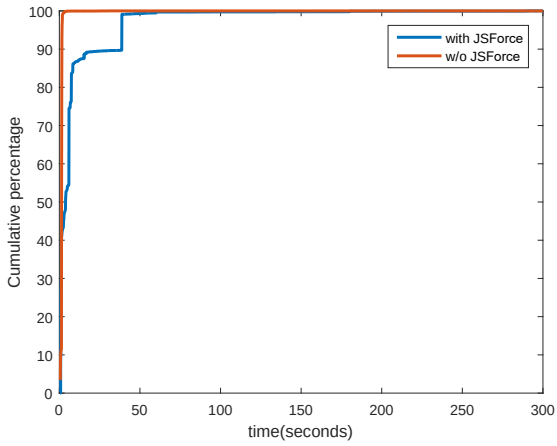

Fig. 3.10.: Runtime for Undetected PDF samples.

3.5.4 Runtime Performance

In this section, we evaluate the runtime performance of JSFORCE by using our malicious and benign datasets with a comparison between the original jsunpack and the JSFORCE-extended version.

Runtime for Detected Samples In this section, we compare the runtime performance using the HTML and PDF samples that can be detected by jsunpack both with and without JSFORCE. The reason why we chose this sample set is that we wished to observe whether the JSFORCE-extended version can achieve efficiency comparable to the original jsunpack when using a detectable malicious sample. The results are displayed in Figures 3.7 and 3.9. The results conclude that JSFORCE-extended version has better runtime performance than jsunpack for over 90.9% of HTML and 83.6% of PDF samples. This conclusion is quite surprising as the JSFORCE-extended version tends to explore multiple paths while jsunpack only probes for one.

AnalysisSystem	SampleSet	Conf. w/o Rozzle	Conf. With Rozzle	DetectedBy BothConf	MissedBy Rozzle-extendedConf.
Nozzle	Offline	1,662	11,559	1,178	484(29%)
	Online	74	224	50	24(32%)
Zozzle	Online	2,735	2,660	2,510	225(8%)

Table 3.4: Detection Results With/Without Rozzle-extended Configuration

In theory, jsunpack should have better runtime performance. However, after investigation, we found that many of the JavaScript samples require specific system configurations (such as specific browser kernel version) to run. As a result, when jsunpack performs analysis, it will run the JavaScript programs under multiple settings. This results in multiple executions, which take additional time to complete. In contrast, the JSFORCE-extended version handled this issue with forced execution, resulting in better runtime performance in practice.

Runtime for Undetected Samples Figures 3.8 and 3.10 show the runtime performance of JSFORCE for undetected samples. We empirically set the time limit to be 300 seconds in consequence of the fact that experiment shows almost all (99.6%) HTML and PDF samples can be analyzed within 300 seconds. As demonstrated in the figures, the average analysis runtime for HTML and PDF samples are 12.02 and 8.15 seconds, while the analysis for a majority (80%) of HTML samples and PDF samples are finished within 8.54 and 7.4 seconds, respectively. When compared with the original jsunpack, the JSFORCE-extended version achieves an average runtime of 16.08 seconds and 7.97 seconds for undetected HTML and PDF samples while jsunpack finishes execution in 1.13 seconds and 1.37 seconds, correspondingly. Our conclusion from these experiments are that the performance overhead of JSFORCE is quite reasonable and can certainly meet the requirements of large scale JavaScript analysis.

3.5.5 JSFORCE vs. Rozzle

Ideally, we would like to perform a head-to-head comparison between JSFORCE and Rozzle using the same dataset. Unfortunately, it is impossible given that neither the Rozzle system nor the dataset used by Rozzle is available for evaluation. It is also nontrivial to implement Rozzle by ourselves. Nevertheless, we can still highlight several advantages of JSFORCE over Rozzle, from the experimental results reported in that paper.

First, while Rozzle-extended analysis system does, JSFORCE-extended analysis system does not miss samples detected by the original analysis system. Table 3.4 summarizes the detection results presented in Rozzle paper. Using Rozzle, the experiments extend two malicious JavaScript detection systems - Nozzle [10] and Zozzle [11], and then compare the detection results with the original system using one offline sample set and one online sample set. For the offline experiment, with Rozzle, Nozzle can detect 11,559 samples and gains a significant improvement(11,559 vs. 1,662) over original Nozzle. But it misses 484 (29%) samples which can be detected by original Nozzle. For online experiments, Rozzle-extended configuration also misses 24 (32%) for Nozzle, 225 (8%) for Zozzle respectively. Rozzle paper argues this is because that the runtime errors, introduced when infeasible paths are executed, terminates the execution before the malicious behaviors are exposed. However, since JSFORCE only collects the path information and no changes are made on the path when the sample is first executed, no runtime errors are introduced by JSFORCE. Thus as demonstrated in Section 3.5.3, JSFORCE-extended analysis system can detect all the samples identified by original analysis system while providing the same magnitude improvement as Rozzle's.

Second, JSFORCE can still function even when the environment setup is incomplete, thanks to the forced execution model (Section 3.3.1), whereas Rozzle may fail due to the runtime errors. This is especially important for low-interaction honey clients like jsunpack. Those low-interaction honey clients emulate the behaviors of browsers or PDF readers, and it is quite challenging to construct a complete environment setup for the tested samples. As discussed in Section 3.6, of the malicious samples missed by jsunpack, 96.5% are because of the runtime errors caused by incomplete emulation of the running environments for JavaScript code. Since low-interaction honey clients are widely deployed in industry, we argue that JSFORCE would benefit the industry more than Rozzle.

Third, as discussed in the limitation part of Rozzle paper, Rozzle is less effective for the case that the evasive code triggers the malware execution only when a user interaction occurs, or when a timer fires. We searched the samples missed by jsunpack with the keywords like “onclick” or “settimeout”. we found that 80.6% of them deploy timers or user interaction callbacks. JSFORCE’s path exploration algorithm discovers the callback functions during the execution, and invokes them after the current run terminates. However, Rozzle may miss the malicious code hidden in callback functions.

Fourth, Rozzle cannot handle latest fingerprinting techniques discussed in Section 3.2. While we have not found samples deploying these techniques in our dataset, we believe that the attacker will deploy those new fingerprinting techniques with the advancement of anti-evasion techniques in the future. So JSFORCE is one step ahead of the attacker.

```
try{++(document["body"]);
} catch(e) {[shellcode]}
```

[a]

```
window.addEvent('load',
  function() {[shellcode]})
```

[b]

```
function frmAdd() {
  var ifrm = document.createElement('iframe');
  ifrm.style.position='absolute';
  document.body.appendChild(ifrm);};
frmAdd();
```

[c]

```
var se_show_newupdates = new Hash.Cookie('
    se_show_newupdates', {duration: 3600});
```

[d]

```
gapi.load("gapi.iframes", function() {[...]});
```

[e]

Fig. 3.11.: Case Study Samples

3.6 Case Study

To better understand the benefits of JSFORCE, we conducted a case study on 10,975 unique JavaScript code samples missed by jsunpack but detected by JSFORCE-extended version. The reasons of failed detection using jsunpack can be divided into the following two categories.

Malicious code branch is not triggered Of those 10,975 samples missed by jsunpack, we found that 10,792 (98.33%) samples are explored by at least two paths when using JSFORCE. Although jsunpack attempts to run the sample several times with different configurations to increase the chance of triggering the malicious code branch, it is usually ineffective to do so. This is because it is impossible to emulate every single combination of browser/PDFReader/plugin. In contrast, JSFORCE can explore these paths regardless of the configuration.

The sample in Figure 3.11(a) hides malicious code within a `catch` block. The attacker attempts to increase the value `document[body]` as a number, which will raise an exception when executed within a real browser. However, it does not raise an exception in jsunpack since its

SpiderMonkey engine returns `NaN` for this operation. In fact, the V8 engine used in JSFORCE also exhibits the same behavior as SpiderMonkey. But, the `catch` block is triggered by the path exploration process, so the malicious behavior is still revealed.

Other samples hide code within event callbacks. For instance, the sample in Figure 3.11(b) registers a callback function using the `window.addEvent` function. jsunpack fails to invoke the callback function due to the incorrect definition of `window.addEvent` used by jsunpack. At runtime, JSFORCE identifies `window.addEvent` as a callback registration function because an anonymous function is passed to it as the parameter. Then, this anonymous function is queued and invoked at the end of execution.

Execution fails due to runtime errors Another reason why jsunpack may fail to detect malicious JavaScript code is that the execution can fail due to runtime errors. As we conducted the evaluation, only 230 out of the 10,975 samples could be executed without any runtime errors under the six configurations. Moreover, 10,592 out of 10,975 (96.5%) failed all six configurations, rendering jsunpack completely useless when facing them. These exceptions terminate the execution before the malicious code is executed. The raised exceptions are because of the inaccurate emulation of the running environment for JavaScript code. Examining these exceptions can help security researchers improve jsunpack by supplying more precise emulation environment, which is another benefit that JSFORCE can provide.

One interesting thing about jsunpack is that it tries to fix `ReferrenceError` by providing a definition for this undefined object once `ReferrenceError` is captured. While this fix eliminates the `ReferrenceError`, it often introduces `SyntaxError` or `TypeError` at runtime. `ifrm.style` is not defined in the sample in Figure 3.11(c). So jsunpack generates code `var`

`ifrm.style = 1` for this sample. Unfortunately, it contains an unexpected token dot. This raises a `SyntaxError` exception. Another way to improve this is to assign `ifrm.style` an `Object` so that `SyntaxError` is avoided and `ifrm.style` can be typed following the typing rules of the JavaScript engine. However, as discussed in Section 3.3.1, this can still cause an exception or lead to unnecessary loss of precision. This case demonstrates the advantage of type inference model deployed by JSFORCE. Although JSFORCE cannot tolerate `SyntaxError`, the type inference model guarantees no further `TypeError` or `SyntaxError` will be introduced.

The sample in Figure 3.11(d) raises a `TypeError` exception since `Hash.Cookie` is not a constructor. Another sample in Figure 3.11(e) also raises a `TypeError` exception because `gapi.load` is not a function. JSFORCE can avoid this by applying faked object retyping technique. From another perspective, these two cases manifest the weakness of jsunpack that `Hash.Cookie` and `gapi.load` are not correctly defined. Therefore, as another application, JSFORCE can be used to evaluate the weakness of dynamic JavaScript analysis systems, so security researchers can further improve the systems respectively.

4. SEMANTICS-PRESERVING DISSECTION OF JAVASCRIPT EXPLOITS VIA DYNAMIC JS-BINARY ANALYSIS

4.1 Introduction

Previously unknown, or "zero-day", exploits are of particular interest to the security community. Once a malicious JavaScript attack is captured, it must be analyzed and its inner-workings understood quickly so that proper defenses can be deployed to protect against it or similar attacks in the future. Unfortunately, this analysis process is tedious, painstaking, and time-consuming. From the analysis perspective, an analyst seeks to answer two key questions: (1) Which JavaScript statements uniquely characterize the exploit? and (2) Where is the payload located within the exploit? The answer to the first question results in the generation of an exploit signature, which can then be deployed via an intrusion detection system (IDS) to discover and prevent the exploit. The answer to the second question allows an analyst to replace the malicious payload with an amicable payload and use the modified exploit as a proof-of-vulnerability (PoV) to perform penetration testing.

Program slicing [77] is a key technique in exploit analysis. This technique begins with a source location of interest, known as slicing source, such as a statement or instruction that causes a crash, and identifies any statements or instructions that this source location depends on. Prior exploit analysis solutions have attempted to analyze exploits at either the JavaScript level [5, 9–11, 19, 20] or the underlying binary level [21–25].

While binary level solutions execute an exploit and analyze the underlying binary execution for anomalies, they are unaware of any JavaScript level semantics and fail to present the JavaScript level view of the exploit. JavaScript level analysis fails to account for implicit data flows between statements because any DOM/BOM APIs invoked at the binary level are invisible at the JavaScript level. Unfortunately, implicit flows are quite common in attacks and are often comprised of seemingly random and irregular operations in the JavaScript that achieve a precise precondition or a specific trigger which exploits a vulnerability in the binary. The semantic gap between JavaScript level and binary level during the analysis makes it challenging to automatically answer the 2 key questions.

In this chapter, I present JSCALPEL, a system that creatively combines JavaScript and binary level analyses to analyze exploits. It stems from the observation that seemingly complex and irregular JavaScript statements in an exploit often exhibit strong data dependencies in the binary. JSCALPEL utilizes the JavaScript context information from the JavaScript level to perform *context-aware* binary analysis. Further, it leverages binary analysis to account for implicit JavaScript level dependencies arising due to side effects at the binary level. In essence, it performs JavaScript and binary, or *JS-Binary* analysis. Given a functional JavaScript exploit, JSCALPEL performs JS-Binary analysis to: (1) generate a minimized exploit script, which in turn helps to generate a signature for the exploit, and (2) precisely locate the payload within the exploit. It replaces the malicious payload with a friendly payload and generates a PoV for the exploit.

I evaluated JSCALPEL on a corpus of 15 exploits, 9 from Metasploit[1], 4 exploits from 3 different exploit kits and 2 wild exploits. On average, I was able to reduce the number of unique JavaScript statements by 49.8%, and precisely identify the payload, in a semantics-preserving

[1]Metasploit Framework – http://www.metasploit.com/, a popular penetration testing framework.

manner, meaning that the minimized exploits are still functional. In addition, we were able to replace the payload with amicable payload to perform penetration testing. Finally, I presented the wild exploit CVE-2011-1255 as a case study. I demonstrate how the exploit is minimized and payload is located.

4.2 Problem Statement and Overview

4.2.1 Problem Statement

We aim to develop JSCALPEL– a framework to combine JavaScript and binary analyses to aid in analysis of JavaScript-launched memory corruption exploits. It is motivated by two key observations.

First, analysis performed at only the JavaScript level is insufficient. In Figure 2.3(b), JavaScript level analysis of Aurora captures the explicit data dependencies between statements 9 and 26 and statements 6 and 18. However, because no explicit dependency exists between statements 18 and 26, the two groups of statements will be incorrectly deemed to be independent of each other. Second, while complete, analysis performed at only the binary level is also insufficient. In Figure 2.3(d), binary level analysis can expose the manipulation of pointers, however it can not expose exploit-related JavaScript statements in Figure 2.3 (c) due to the lack of JavaScript context. A binary-level analysis will show the memory written by the binary instructions of statement 18 is utilized through reads performed by binary instructions of statement 26, revealing a straight-forward data dependency between statements 18 and 26.

Input: JSCALPEL accepts a raw functional exploit and a vulnerable program as input. The vulnerable program can be any program like (PDF reader, web browser, etc.) as long as it can be

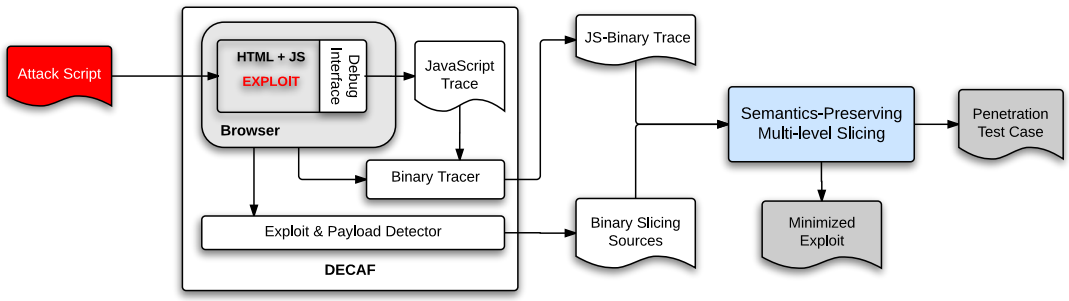

Fig. 4.1.: Architecture of JSCALPEL

exploited through JavaScript. The exploit consists of HTML and malicious JavaScript components. The exploit can be obfuscated or encrypted. JSCALPEL makes no assumptions about the nature of payloads. That is, the payload could be ROP-only, executable-only or combined.

Output: JSCALPEL performs JS-Binary tracing and slicing and generates 3 specific outputs. (1) A simplified exploit HTML that contains the key JavaScript statements that are required to accomplish the exploit, and (2) the precise JavaScript statements that inject the payload into the vulnerable process' memory along with the exact payload string – both non-executable and executable – within the JavaScript. Finally, (3) an HTML page, where the malicious payload is replaced by a benign payload is generated as a Proof-of-Vulnerability (PoV).

Delta debugging [78] is firstly proposed to generate the minimized *C* programs that crash the compiler and might be a feasible approach to minimize the exploit JavaScript to cause a crash. However, the effectiveness of this approach is unknown, because of the complex and sophisticated nature of JavaScript. Attackers can insert arbitrary junk code to make delta debugging ineffective. In contrast, JSCALPEL can precisely pinpoint the JavaScript statements that cause a crash and locate the malicious payload and our experiment has proven its effectiveness.

4.2.2 JSCALPEL– Overview

Figure 4.1 presents the architecture of JSCALPEL, which leverages Virtual Machine Monitor (VMM) based analysis. It consists of multiple components. A multi-level tracer is used to gather JavaScript and binary traces. A CFI module is used to determine the binary level "slicing sources", which are the violations that cause the exploit along with the various payload components. The multi-level slicer augments JavaScript level slicing with information from binary level slicing to obtain the relevant exploit and payload statements. Finally, JSCALPEL packages the relevant exploit statements within an HTML page to generate the minimized script. It also replaces the malicious payload with a benign payload to generate a PoV.

4.3 Multi-level Tracing and Slicing-Source Identification

We implement JSCALPEL on top of DECAF [79], a whole-system dynamic analysis framework. The tracing consists of two parts, JavaScript and binary tracers. JavaScript tracing is performed using a helper module that is injected into the browser address space. It interacts with the JavaScript debug interface within the browser to gather the JavaScript-level trace. The binary tracer and the exploit detection module are implemented as 2 plugins of DECAF. Below, we detail each of the components.

4.3.1 Context-Aware Multi-Level Tracing

JavaScript Tracer Prior approaches that gather JavaScript trace [5, 6] modify JavaScript engine or the browser to identify the precise statements being executed, however such an

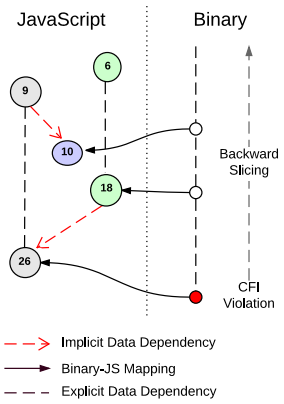

Fig. 4.2.: Multi-level analysis of Aurora Exploit.

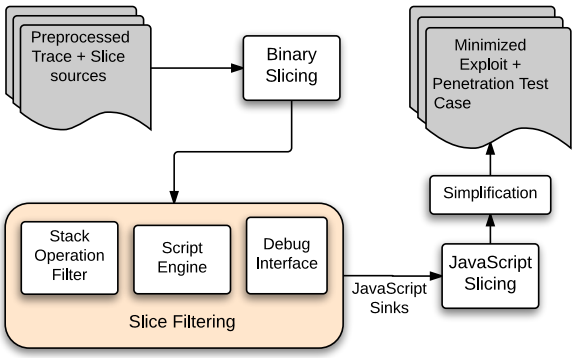

Fig. 4.3.: Semantics-Preserving Multi-level Slicing.

approach requires access to JavaScript engine (and/or browser) source code which is not available for close sourced browsers like IE.

We take a JavaScript debugger-based approach. Our approach has two key advantages. (1) Most browsers – open-sourced or otherwise – support a debugging interface to debug the JavaScript statements being executed, and (2) Because the debugger runs within the browser context, it readily provides the JavaScript-level semantics. That is, we can not only gather the exact statements being executed, but also retrieve the values of variables and arguments at various statements. From within the VMM, we hook the JavaScript debugger at specific APIs to retrieve the various JavaScript statements and the corresponding contexts. The accumulation of the JavaScript statements yields the JavaScript trace.

JavaScript tracer runs as an injected module within Internet Explorer. It implements the "active script debugger" [80] interface and performs three specific actions:

1) *Establish Context*: Through the script-debugger interface, the tracer is notified when execution reaches JavaScript code. Specifically, if a `SCRIPT` tag is encountered within an

existing script or the script generated through `eval` statement, the tracer is activated with the information regarding the statement being executed. Until the next statement executes, the tracer associates the context to the current JavaScript statement.

2) *Record Trace*: At the beginning of every JavaScript statement, the tracer records the exact statement semantics along with the variable values and arguments to APIs (if any).

3) *Drive Binary Tracer*: A stub function is defined to coordinate the JavaScript tracer and the binary tracer. Before the statement executes, the binary tracer is activated along with the context information passed as the arguments of stub function such that the binary trace is associated with the particular JavaScript statement.

Binary Tracer Binary tracer is triggered by the JavaScript tracer with the context information pertaining to a particular JavaScript statement. One way to gather a binary trace would be to monitor and capture the entire execution of the browser process at an instruction level. However, such a solution is resource intensive and inefficient. In order to be practical, our solution is selective about *what* is traced and *when* it is traced. Our goals towards an effective binary trace are to: (1) include all the relevant binary instructions that contribute to the attack, and (2) minimize the trace footprint as much as possible.

Firstly, since binary tracer is driven by JavaScript tracer, it has the precise JavaScript context. Tracing is limited and selectively turned on only when the execution is in a JavaScript statement. It is likely that the multithreading of the browser will introduce unrelated execution trace. But it does not jeopardize the analysis since all the binary instructions that contribute to the attack are included. Secondly, the effects of statements at a JavaScript-level manifest as memory reads and

writes at a binary-level. Therefore, we implement a lightweight tracing mechanism. Instead of logging every binary instruction, we only log the memory read or write operations. We leverage memory IO specific callbacks supported by DECAF to record the values of eip, memory address, memory size, value in the memory and esp for each memory IO instruction. We also record the addresses of basic blocks that are executed and dump their raw bytes from virtual memory space of the monitored process at the end of every JavaScript statement. Furthermore, the binary tracer maintains information about active allocations made by the victim process. This information is used to identify self-modifying (or JIT) code. When such code is encountered, the code is dumped to the disk. When needed, the raw bytes are decoded to retrieve the actual instructions. The propagation of the slicing sources between registers and memory is identified by the memory IO logs and the binary instruction logic. While preserving the completed information as full instruction trace does for slicing process, this lightweight trace minimizes the trace size and also speeds up the slicing process.

Binary tracer is implemented as a plugin to DECAF. In the plugin, the stub function of JavaScript tracer is hooked to coordinate the binary tracing and JavaScript tracing. When the stub function is invoked by JavaScript tracer, the Binary tracer first reads the parameters of stub function from the stack where JavaScript Tracer passes the JavaScript statement and debugger information, then starts the logging of binary trace and generates a combined JS-Binary trace which contains the JavaScript and binary traces for each of the JavaScript statements. Meanwhile, a JS-binary map is built to keep track of corresponding JavaScript statement for every binary instruction.

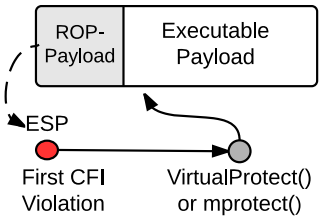

Fig. 4.4.: Non-executable (ROP) and executable payloads used in an exploit.

Obfuscation and Encryption Resistance The nature of JavaScript tracing provides inherent resistance to obfuscation and encryption because it captures each statement that is executed along with the runtime information like variable values, arguments, etc. Therefore, the intermediate statements (like the ones in Figure 2.3(a)) that are used to calculate a value are each captured with their concrete values. Similarly, encrypted statements must be decrypted before they are executed, and the decrypted statements execute. Therefore, JSCALPEL encounters and records the decrypted statements that execute.

In fact, JSCALPEL performs preliminary preprocessing by performing constant folding with the help of the script execution trace. This simple optimization will not cause over simplification and generates a functionally equivalent de-obfuscated and decrypted version of the script. Then JSCALPEL executes the de-obfuscated version to perform the analysis. This preprocessing reduces the amount of analyzed JavaScript statements.

4.3.2 Identifying Slicing Sources

JSCALPEL makes use of a CFI module to identify slicing sources. Several solutions have been proposed to implement CFI [81]. Since JSCALPEL already relies on a VMM for trace gathering, it can leverage a VMM based CFI defense. We opt the techniques presented in Total-CFI [25]

because (1) it is a recent and practical solution, (2) it has been demonstrated to work on recent real-world exploits and finally (3) it imposes low overhead. It monitors the program execution at an instruction level and each point where the CFI is violated is noted as a slicing source. Albeit the recent advancement of exploitation techniques [82] can bypass the coarse-grained CFI techniques like Total-CFI, JSCALPEL's CFI module can be enhanced to include more policies to adapt the development of exploitation techniques.

Specifically, the first violation is the slicing source for the exploit-related code, whereas the subsequent violations (if any) arise from the executable payload or ROP-payload. In Figure 4.4, the first violation is caused by the exploiting code, then the violations that occur up to the execution of executable payload serve as sources for ROP-payload. Moreover, the CFI module continues execution to check for executable payloads. If after the first violation, the execution ever reaches a region that within the list of allocated regions, the address is noted and it serves as the binary slicing source for the executable payload.

4.4 Multi-level Slicing

Multi-level slicing employed by JSCALPEL is based on the following hypothesis.

Hypothesis *Implicit data dependencies at JavaScript level often manifest as direct data dependencies at binary level.*

Memory corruption exploits typically corrupt the memory by causing precise memory writes to key locations that are read by the program and result in corruption of program counter. Chen et al., show that a common characteristic of many classes of vulnerabilities is pointer taintedness [48], where a pointer is said to be tainted if the attacker input can directly or indirectly

Algorithm 2 Binary level backward slicer

Input: binray trace B,slicing source S and JS-Binary map M and JavaScript trace list J
Output: JavaScript slice O

$S \leftarrow \{slicing\ source\ (exploit\ point\ or\ payload\ location)\}$
$O \leftarrow \varnothing$
for $i = len(J); i > 0; i++$ **do**
  $B_i \leftarrow getBinInsTraceForJS(M, J[i], B)$
  for $k = len(B_i); k > 0; k++$ **do**
    $b_{ik} \leftarrow B_i[k]$
    $L \leftarrow \varnothing$
    if $S\ is\ all\ memory\ locations$ **then**
      $M_w \leftarrow GetMemWriteRec(b_{ik})$
      if $S \cap M_w == \varnothing$ **then**
        $continue$
      end if
    end if
    if $getDestOperand(b_{ik}) \in S$ **then**
      $S \leftarrow S \cup updateSliceSource(b_{ik}, S)$
      $L \leftarrow L \cup \{b_{ik}\}$
    end if
  end for
  if $L! = \varnothing$ **then**
    $O \leftarrow O \cup \{J[i]\}$
    $L \leftarrow \varnothing$
  end if
end for

reach the program counter. In essence taint propagation reflects runtime data-flow within the program. Therefore, at a binary level, memory corruption exploits such as use-after-free, heap overflow, buffer overflow, etc. often exhibit simple data-flow, which can be captured through data-dependency analysis.

Figure 4.3 presents the overview of slicing employed by JSCALPEL. In order for the simplified exploit to be functional, it is necessary that the simplification preserves the semantics between the original and simplified scripts. Given the slicing sources and the JS-binary trace, JSCALPEL first performs a binary backward slice from the slice source provided by CFI violation and generates sources for JavaScript-level slicing. Slicing at the binary level ensures that no required statement is missed. Then, slicing is performed at a JavaScript level to include all the statements that sources are either data- or control-dependent on.

4.4.1 Binary-level Slicing

The goal of binary slicing is to identify all the JavaScript statements that are instrumental in coercing the control flow (i.e., statements that modify the program counter) or injecting the payload into memory.

Algorithm 2 describes the backward slicing method using the lightweight binary trace. For every JavaScript statement $J[i]$, the corresponding binary instruction trace B_i is extracted. A map called "JS-Binary map" M – a mapping between the JavaScript statements and the binary instructions that execute within the statement context – is used. Then for every binary instruction $b_{ik} \in B_i$, if all of the elements in the slicing source S belong to memory locations, then the slicer checks if the current binary instruction b_{ik} has memory write operations $M_w \subseteq S$ and if it is false, the slicer jumps to the next instruction $b_{i(k+1)}$. Otherwise, the slicer does as traditional slicer to disassemble the binary instruction b_{ik} and updates the slicing source S and determine if b_{ik} should be added in the binary slice L based on the propagation rules for every x86 instruction. If L is not empty when the slicing on B_i is finished, $J[i]$ is added to the JavaScript slice O as the hidden dependency slice which may be ignored by pure JavaScript-level slicing.

In theory, a binary backward slice from the slicing sources must include all the JavaScript statements that are pertinent to the attack. However, in practice we found a key problem with such an approach. It is too permissive and ends up including *all* the JavaScript statements in the script. The main reason is the binary-level amalgamation of JavaScript and browser code along with JavaScript code. In order to track the exploit-specific information-flow, the flow through pointers must be considered. However, at a binary level, due to the complex nature of a JavaScript engine, dependencies are propagated to all the statements thereby leading to dependency explosion.

We exclude data propagation arising from code corresponding to the script engine and debug interface. Particularly, we apply the following filters to minimize the dependency explosion problem.

Stack Filtering Once the dependency propagates to stack pointer `esp` or stack frame `ebp`, all data on the stack becomes dependent [83]. To avoid this, dependencies arising due to `esp` or `ebp` are removed during slicing. In certain cases, the stack data could be marked dependent, but when the callee returns, the dependency is discarded if it exists on a stack variable. So JSCALPEL records the current stack pointer for every read/write, and during backward slicing, when `call` instruction is encountered in the trace, the slicer checks the current stack pointer and clears the dependencies propagating from the callee's stack.

Module Filtering During the slicing process, the propagation to or from the JavaScript engine module or script debugger is stopped. In principle, every Javascript statement executed by the same Javascript engine instance shares the data and control dependency introduced by the Javascript engine and debugger module. This kind of dependency is *outside* of "exploit specific" dependency and should be excluded from slicing.

Other Filters Between two consecutive JavaScript statements, we found that sometimes there are data flows via CPU registers because of the deep call stack incurred by JavaScript engine and script debugger. To avoid unintended dependencies, the slicer clears the register sinks at the end of the slicing for every JavaScript statement. During our experiments (Section 4.5), we found the above filters good enough to reduce the dependency-explosion problem without missing any required statements.

4.4.2 JavaScript Slicing

The output of binary tracer provides the slicing sources for the JavaScript slicer. Suppose binary slice S contains n instructions. For each instruction S_i, let J_i be the JavaScript statement that represents the context under which S_i executes. Then, the JavaScript slicing sources are $\mathcal{O} = \bigcup_{i=0}^{n} J_i$. For every JavaScript statement in the slicing sources, we add the object used by this JavaScript statement to the slicing sources and include this JavaScript statement in the slice. Given the JavaScript trace, the slicer uses WALA's [84] slicing algorithm to include all the related JavaScript statement in the slice.

4.4.3 Minimized Exploit Script and PoV Generation

The statements are first simplified and then embedded into the exploit HTML page to obtain the minimized exploit. Also, the identified executable payload is replaced by an amicable payload to obtain a PoV in the form of a test case for the Metasploit framework.

Simplification As a final step, JSCALPEL performs constant folding and dead-code elimination at JavaScript level to simplify the slice. It is focused on strings and constants. Specifically, for each variable v, the definitions are propagated to the uses. This is repeated for all the variables in all the statements until no more propagations are possible. Finally, if a definition of a variable has no more uses, the definition is considered dead-code and is removed only if the statement is not a source for the JavaScript slicing. This distinction is important because, the need for slice sources is already established from binary slicing. The resulting processed script is used to exploit the browser and is accepted only if the exploitation succeeds. Finally, all the statements in

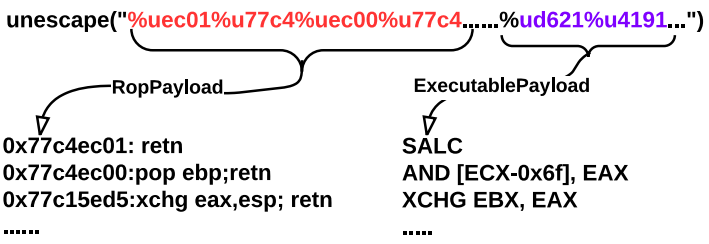

Fig. 4.5.: CVE-2012-1876: ROP- and executable-payloads within the same string.

the script that are not a part of the slice are removed. During our experiments, we found that the simplicity of simplification incorporated by JSCALPEL is sufficient to bring about significant reduction in the sizes of the scripts as highlighted in Section 4.5.

Collocated ROP and Executable Payloads In some exploits, the payload and the ROP-gadgets are contained within the same string or array. For example in Figure 4.5 the same string contains both ROP-payload and the executable shellcode. In such cases, JS-Binary analysis identifies the statement as both exploit and payload statement. This is an expected behavior. However, in order to replace the payload to generate the PoV, we must precisely identify the location of the start of the payload within the string. First, the JavaScript string that contains the payload is located in the memory. Then, from the payload-slice source we obtain the address of the entry point of the payload. Binary slicing from the payload-slice source leads us to the offset within the JavaScript string that corresponds to the payload. The substring beginning from the offset is replaced for PoV generation.

ROP-Only Payload Shacham [85] showed that a set of Turing complete gadgets can be created using program text of libc. Though we cannot find any instances of *ROP-only* payload during our experiments, it is possible to compose the entire payload using only ROP-gadgets without any

executable payload. Since JSCALPEL can locate the ROP-only payload precisely, a straightforward way is to replace malicious ROP-only payload with benign ROP-only payload.

JSCALPEL can generate dependent JavaScript statements in the script for any given binary-level source and the JS-Binary trace. Along with the exploit point and the payload entry point, CFI component of JSCALPEL captures multiple violations caused due to the ROP-gadget chain as separate binary-level slicing sources. The sources are then subject to multi-level tracing the slicing to extract the payload in JavaScript.

Disjoint Payload Detecting the entry point of executable payload is sufficient to replace the payload and generate the PoV. However, sometimes an analyst may want to locate the *entire* executable payload. This is not a problem if the payload is allocated by the same string in the JavaScript. However, it is not necessary to be so.

JSCALPEL can only detect an executable payload when it executes. Therefore, it is unaware of *all* the various fragments of payload that may be injected into the memory. As a result, JSCALPEL will only be able to detect the JavaScript statement (and all its dependencies) that injects the entry point of the payload. It may miss some JavaScript statements that inject non-entry point payload if such statements are disjoint with the JavaScript statements that inject the entry point, and the sources for those statements are missing. Note that this is not quite a limitation for JSCALPEL, because the payload entry point is sufficient to generate a PoV. One way to increase the amount of payload recovered is for the CFI module to allow the payload to execute longer and capture more binary-level sources for the payload.

4.5 Evaluation

We evaluate JSCALPEL on a corpus of 15 exploits. These samples exploit the vulnerabilities discovered from 2009 to 2013 and target at Internet Explorer 6/7/8. In contrast to the large number of browser vulnerabilities discovered every year, this sample set is relatively old and small. The reasons are twofold. First, DECAF leveraged by JSCALPEL is based on emulator QEMU and only supports 32-bit operating system. Not all of the exploits can function correctly on DECAF. Second, it is difficult to collect working exploits although many vulnerabilities are discovered every year. We went over Internet Explorer related exploits in Metasploit, and tried to set up a working environment for each of them. We were able to set up 15 exploits on the real hardware. The remaining exploits either require specific browser/plugin versions that we were unable to find, or do not use JavaScript to launch the attacks. We then tested these 15 exploits on DECAF and 9 of them worked correctly. The 6 exploits failed to work on DECAF, because they exhibited heavy heap spray behavior, which could not finish within a reasonable amount of time in DECAF. Based on a whole-system emulator QEMU, DECAF translates a virtual memory address into its corresponding physical address completely in software implementation, and thus is much slower than the MMU (Memory Management Unit) in a real CPU. In the future, we will replace DECAF with Pin to avoid this expensive memory address translation overhead. We also crawled the Virustotal with the keyword “exploit type:html”, and finally found 2 functional exploits on DECAF. In addition, from 16 exploit kits used in EkHunter[86], we managed to get 4 functional exploits from exploitkit, Siberia and Crimepack. As a result, our testset includes 9 exploits from Metasploit framework, 4 exploits from 3 different exploit kits and 2 wild exploits.

Table 4.1: Exploit Analysis Results

Source	CVE	Exploitation Component					Payload Injection	Simplified Exploit			
		I	II	III	IV	V	VI	VII	VIII	IX	X
Metasploit	2009-0075	9	6	✓	17	✓	14	30	30	0.00	
	2010-0249	3	6	✗	19	✓	10	45	22	0.51	†*
	2010-0806	2	10	✓	10	✓	14	803	13	0.98	‡* †
	2010-3962	1	1	✓	1	✓	15	105	17	0.83	‡* †
	2012-1876	32	1	✗	30	✓	14	67	47	0.30	‡* †
	2012-1889	1	2	✓	2	✓	67	77	77	0.00	
	2012-4969	16	1	✗	8	✓	53	117	70	0.40	†*
	2013-3163	9	1	✗	13	✓	32	43	42	0.02	‡†§
	2013-3897	26	1	✗	41	✓	23	187	63	0.66	§
Wild	2011-1255	40	1	✗	16	✓	26	97	44	0.55	‡†★
	2012-1889	1	2	✓	2	✓	27	53	12	0.77	‡†★
exploitkit	2010-0806	2	6	✓	6	✓	13	109	29	0.73	†*
Siberia	2010-0806	2	6	✓	6	✓	12	103	22	0.79	‡†*
Crimepack	2010-0806	2	1	✗	6	✓	11	198	30	0.85	‡†*
	2009-0075	4	6	✗	12	✓	12	36	33	0.08	‡†*

I. # of JS slicing sources. II. # of stmts from JS analysis only.
III. Can stmts from JS-only analysis cause crash? IV. # of stmts from JS-Bin analysis
V. Can stmts from JS-Bin analysis cause crash? VI. # of stmts from JS-Bin analysis
VII. # of unique JS stmts of original exploit. VIII. # of unique JS stmts of simplified exploit
IX. potency of minimization. X. Obfuscation & fingerprinting Techniques.(‡: Randomization Obf. †: Data Obf. *: Encoding Obf. §: Logic Structure Obf. ★: Fingerprinting tech)

To identify the CVE number of exploits from exploit kits and wild, we ran JSCALPEL to extract exploitation component first and then manually searched Metasploit database and National Vulnerability Database [87] for a match. While CVE-2012-1889 exploits the vulnerability in `msxml.dll`, all the remaining samples exploit `mshtml.dll`.

Though we evaluated JSCALPEL on Internet Explorer only, potentially it can work on other browsers or any other programs (e.g., Adobe Reader) that have JavaScript debug interface. The experiments were performed on a server running Ubuntu 12.04 on 32 core Intel Xeon(R) 2 GHz CPU and 128 GB memory. The code comprises of 890 lines of Python, 2300 lines of Java and 4000 lines of C++.

4.5.1 Minimizing Exploits

Table 4.1 presents the results for exploit analysis. Given one exploit, we first ran JSCALPEL to get the multi-level trace and CFI violation point. Then multi-level slicing was conducted to yield exploitation component and payload injection component. Based on this knowledge, our experiments demonstrate that for each exploit, JSCALPEL was able to generate a simplified exploit and PoV which were able to successfully exploit the vulnerability and launch the payload.

Exploitation Analysis The binary-level slicing was conducted on the multi-level trace starting from the CFI violation point. It mapped binary level slicing results to JavaScript statements with the help of JS-binary map. The number of JavaScript statements identified by binary-level analysis is listed in Column I. They were used as the slicing sources for JavaScript level slicing. This multi-level slicing extracted the exploitation related statements the number of which were listed in Column IV. Column V shows if the extracted statements can crash the browser. For the exploits with the same CVE number like CVE-2009-0075 and CVE-2010-0806, the results of Column IV can be different due to the different implementation of exploitation. But we can see that for all of the exploits, the extracted statements can crash the browser, meaning that the semantics of exploitation component are preserved.

In comparison, the JavaScript-level only analysis cannot achieve this as presented in Column II and III. Column II lists the number of JavaScript statements obtained from backward slicing only at the JavaScript level starting from the statement that causes the first CFI violation. Column III indicates if the statements extracted from JavaScript-level slicing can cause the browser to crash. We can see that for 8 out of 15 exploits, the extracted statements do not cause a crash, which means these exploits are overly simplified in these cases. For the exploits with the same

CVE number like CVE-2010-0806 and CVE-2009-0075, the JavaScript-level only analysis results were different, because the different obfuscation techniques used in these exploits introduced or eliminated unexpected dependency at JavaScript level.

Payload Injection The CFI violation information provides the exact location of the payload in memory. The multi-level slicing yields the payload injection statements of which the number is listed in Column VI of Table 4.1. Column 3 in Table 4.2 lists the payload definition statements. For each of the exploit, our JS-Binary analysis was able to precisely pinpoint the payload injection statements for PoV generation. By contrast, solutions like JSGuard [8] or NOZZLE [10] cannot do the same, because they lack the JavaScript context and can only pinpoint the payload in the memory. Solutions by scanning the exploit code directly cannot always identify the correct payload injection statements since the payload is often obfuscated.

Minimized Exploit For each of the exploits, we combined the payload injection statements (Column VI) with the exploitation component (Column IV) to generate a minimized working exploit. In the experiment, we observed that each minimized exploit was indeed functional, meaning that it can exploit the vulnerability and launch the payload successfully. The Column VII lists the number of unique JavaScript statements observed at the execution of the original exploit. Column VIII lists the number of unique JavaScript statements observed in the execution of the minimized exploit.

The minimized exploit excludes the JavaScript statements that belong to obfuscation code or fingerprinting code. We characterize those codes in Column X of Table 4.1. They cover different obfuscation or fingerprinting techniques. These techniques are designed to bypass the detection

tool and make the analysis challenging. So the minimized exploit can ease the manual analysis process by removing these JavaScript statements. To quantify the degree of code complexity reduction in these minimized exploits, we adopt a metric called "potency of minimization" from an existing work [88]. A minimization is potent if it makes the minimized program P' less obscure (or complex or unreadable) than the original program P. we choose the number of unique JavaScript statements observed in the execution as the metric because it represents the number of inspected statements by an analyst. This is formalized in the following definition:

DEFINITION (POTENCY OF MINIMIZATION) 1. Let $U(P)$ be the number of unique JavaScript statements observed at the execution of P. τ_{pot}, the minimization potency with respect to program P, is a measure of the extent to which the minimization reduces the obscurity of P. It is defined as

$$\tau_{pot} \stackrel{\text{def}}{=} 1 + \frac{U(P')}{U(P)}.$$

On average, the minimization potency was 0.498, which means we were able to eliminate 49.8% of statements in the trace, whereas the maximum is 0.98. The potency of minimization of CVE-2009-0075 and CVE-2012-1889 from Metasploit are both 0, because no obfuscation techniques are applied to them. We did observe that for the exploits from the wild and exploit kits, the average potency of minimization (0.63) was higher than that (0.41) for the exploits from Metasploit. This means that it is generally more difficult to analyze the real world exploits.

4.5.2 PoV Generation

PoV generation is an end result of payload analysis. By replacing the payload in the minimized exploit with a benign one, a PoV is generated for penetration test. Column 3 in

Table 4.2: Payload Analysis Results. All exploits provide a single JavaScript statement from the binary perspective, which is the context in which the exploiting instruction executes.

Source	CVE	Payload definition stmt	I	II
Metasploit	2009-0075	var shellcode = unescape(“%u1c35%u90a8%u3abf...”)	3024	✗
	2010-0249	var LLVc = unescape(“%u1c35%u90a8%u3abf%u..”)	3024	✗
	2010-0806	var wd$ = unescape((function(){ return “%u4772%u9314%u9815...”}))	3072	✗
	2010-3962	var shellcode = unescape(“%u0c74%ud513%uf...”)	3072	✗
	2012-1876	for (var a3d = unescape(“%uec01%u77c4%u...”),...)	3072	✓
	2012-1889	var code = unescape(“%uba92%u91b5%ub0b1...”)	3072	✗
	2012-4969	var GBvB = unescape(“%uc481%uf254%uffff...”)	618	✗
	2013-3163	p += unescape(“%ub860%u77c3%ud038...”)	36696	✓
	2013-3897	sprayHeap({shellcode:unescape(“%u868a%u77c3...”})	696	✓
wild	2011-1255	var sc = unescape(“%u9090%u9090%u9090%u1c3...”)	3024	✗
	2012-1889	var mmmbc=(“Data5756Data3352Data64c9...”)	2880	✗
exploitkit	2010-0806	var qq = unescape(”%ucddb%u74d9%uf424%u...”)	649	✗
Siberia	2010-0806	var qq = unescape(”!5350!5251!..”.replace(...”)	1750	✗
crimepack	2010-0806	var rktchpv= unescape(”%u06b8%u5c67%udae4...”)	648	✗
	2009-0075	var ysazuzbwzdqlr=unescape(”%u06b8%u5c67%u...”)	648	✗

I. Payload Length II. Collocated payload?

Table 4.2 lists the payload definition statements, where the payload content is first introduced or defined in the JavaScript code. The definition statement is usually accompanied by other statements required to inject the payload. Payload length (Column 4 in Table 4.2) is the size of the payload that was identified. In one of the samples (CVE-2013-3163), the encoder was embedded within the payload and therefore, the size of the payload was much larger than other exploits. In 3 out of 15 exploits, we found the ROP and executable payloads to be collocated within the same string. In each exploit, the payload was replaced with a benign payload and a PoV was generated.

4.5.3 Effects of Filtering

The filters help to exclude the unexpected dependencies. In Table 4.3, we evaluated the effects of filtering on minimizing exploits. We found that preprocessing is effective in cases where the

Table 4.3: Effects of Filtering on Exploit Analysis.

Source	CVE	Unique # JS stmts	# JS after pre-processing	No Filter	Stack Filter	Module Filter	All Filters
Metasploit	2009-0075	30	30	30	14	28	9
	2010-0249	45	32	32	4	32	3
	2010-0806	803	27	27	13	27	2
	2010-3962	105	17	16	16	16	1
	2012-1876	67	51	50	41	50	32
	2012-1889	77	78	78	2	77	1
	2012-4969	117	77	77	16	75	16
	2013-3163	43	43	41	4	41	9
	2013-3897	187	64	64	26	64	26
wild	2011-1255	97	66	66	45	66	40
	2012-1889	53	53	51	1	1	1
exploitkit	2010-0806	109	32	31	31	31	2
Siberia	2010-0806	103	27	26	26	26	2
crimepack	2010-0806	198	195	194	22	194	2
	2009-0075	36	35	25	5	5	4

scripts are obfuscated because, during obfuscation, multiple statements are used to accomplish the tasks of a single statement like `eval`. Column 3-4 lists the number of the unique JavaScript statements in the slicing results under different filter configurations. With no filters, we did not find any significant reduction in the slicing results. This emphasizes the need for filtering. Stack Filter and Module Filter individually produced varying amount of size reduction depending on the exploit, but in general, the combination proved to be most effective. For example, for CVE-2010-3962, the combination of all the filters reduced the number of statements to a single statement, while none of the filters were individually effective.

4.5.4 Case Study – CVE-2011-1255

In order to highlight the advantages of JSCALPEL, we perform a study of the wild exploit, CVE-2011-1255 [89], which exploits a “Time Element Memory Corruption Vulnerability” of the

Timed Interactive Multimedia Extension implementation in Microsoft Internet Explorer 6 through 8. The exploit (MD5:016c280b8f1f155 80f89d058dc5102bb) targets Internet Explorer 6 on Windows XP SP3. Given the exploit sample, JSCALPEL successfully generated the minimized exploit code, payload injection code and penetration test template for Metasploit. We would like to highlight that a sample for CVE-2011-1255 was previously unavailable on Metasploit DB and JSCALPEL was able to generate one.

Simplified Exploit Statements JSCALPEL loads the simplified page and logs the JS-Binary trace until the CFI violation-point (detailed in Figure 4.6) is reached. The violation point ① represents the hijacked control flow transfer from `0x7ddd44a1` to the payload location `0x0c0c0c0c` through an indirect call instruction – `call DWORD[ecx+0X8]`. Note that the exploit does not contain any ROP-gadgets and that the entire payload is executable. From the violation, either `ecx` or `[ecx + 0x8]` may be manipulated by the attacker and therefore both will have to be considered as possible slicing sources. From the memory IO log (point ②), the location of `[ecx+0x8]` is extracted as `0x0c0c0c14`. Both `ecx` and the memory location `0x0c0c0c14` are provided as the slicing sources for the binary-level slicer to uncover the implicit data dependency pertaining to the exploit.

The binary level slicer identified 40 JavaScript level sources. JavaScript slicer included an additional 64 statements to generate the simplified exploit. Using the simplified exploit, we were able to trigger the vulnerability in IE 6.

Simplified Payload-Injection Statements and Payload Location Similar to simplifying the exploit statements, JSCALPEL uses payload location *0x0c0c0c0c* as the slicing source for

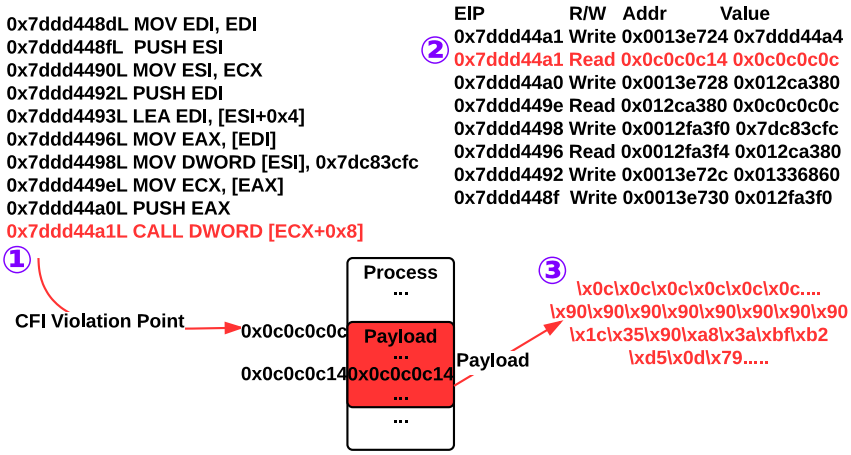

Fig. 4.6.: CFI Violation Point

identifying the payload-injection statements, and gathers the simplified statements. The binary-level slicer confirmed the statement `36: a[i] = lh.substr(0, lh.length)` as the JavaScript statement that injects payload into memory. Then, this statement was used as the slicing source for JavaScript-level slicer. Finally, JSCALPEL identified all the payload injection JavaScript statements.

The payload is located at `0x0c0c0c0c`. Therefore, JSCALPEL extracts the page at `0x0c0c0c0c` to analyze the payload. JSCALPEL first trims the padding instruction like `nop` from the payload. Next, JSCALPEL compares it with the constant strings in the payload injection JavaScript statements to identify the exact payload string. JSCALPEL identified `(var sc = unescape("%u9090%u9090%u90 90%u9090%u1c35%u90a8%u3abf%ub2d5....");)` as the JavaScript statement containing the payload. Since the entire payload is executable, JSCALPEL replaced the entire payload to generate the Metasploit test case. We generate a Ruby template script for Metasploit framework, and we were able to successfully test it on Internet Explorer 6 on Windows XP SP3.

5. VULNERABILITY-AGNOSTIC DEFENSE OF JAVASCRIPT EXPLOITS VIA MEMORY PERTURBATION

5.1 Introduction

To mitigate JavaScript exploits, software vendors have deployed many mitigation techniques like Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), control flow guard [38], sandbox [90], EMET [37], etc.. These mitigation techniques increases the bar on exploitation. Attackers have to combine complex memory preparation, memory disclosure, code reuse and other techniques to launch a successful exploit.

While these exploit mitigation techniques are constantly improving, hacking contests like Pwn2Own [31], GeekPwn [32], etc., consistently demonstrate that the latest versions of Chrome, Safari, Internet Explorer, and Edge can still be exploited. There are two reasons for this: First, most of the latest proposed mitigation techniques require software or compiler tool chain changes and thus could not be deployed promptly. For instance, ASLR-guard [33] is designed to thwart information disclosure attacks, but requires compiler changes and cannot be quickly deployed by software vendors. Second, the deployed mitigation techniques may fail due to newly invented exploitation techniques (e.g., sandbox bypass technique). We argue that *an ideal mitigation technique should be flexible to deploy without requiring code changes and should subvert inevitable exploitation stage(s).*

I observe that a typical JavaScript exploit adopts *memory preparation* to manipulate the memory states. This is a critical stage for JavaScript exploits since attackers need to put something (e.g., ROP chain, shellcode) into a known memory location prior to execution of that code. This offers the chance for defenders to stop the exploits by disturbing this *memory preparation* stage.

In this chapter, I propose CHAFFYSCRIPT, a vulnerability-agnostic mitigation system that *blocks JavaScript exploits via undermining the memory preparation stage.* Specifically, given suspicious JavaScript, CHAFFYSCRIPT rewrites the code to insert memory perturbation code, and then generates semantically-equivalent code. JavaScript exploits will fail as a result of unexpected memory states introduced by memory perturbation code, while the benign JavaScript still behaves as expected since the memory perturbation code does not change the JavaScript's original semantics.

I have implemented a prototype of CHAFFYSCRIPT, which consists of three main components: a memory allocation/free candidates discovery module to identify the potential memory preparation operations, a lightweight type inference module to prune the unnecessary memory preparation candidates, and a chaff code generation module to insert memory perturbation code alongside memory preparation operations. As a demonstration of the deployment flexibility afforded by our approach, we have integrated CHAFFYSCRIPT into a web proxy to protect users against malicious HTML files. Our evaluation results show that: 1) the probability of guessing the correct memory states after CHAFFYSCRIPT is extremely low (Section 5.6.1), 2) CHAFFYSCRIPT can thwart the latest JavaScript exploits effectively (Section 5.6.2) and 3) it incurs runtime overhead 5.88% for chrome and 12.96% for FireFox at

most, and the memory overhead is 6.1% for the minimal JS heap usage, and 8.2% for the maximal JS heap usage during runtime on Octane benchmark [91] (Section 5.6.3).

5.2 Technical background and motivation

5.2.1 Defense of JavaScript Exploits

Defense against JavaScript exploits has evolved to react to advances in exploitation techniques. Any defensive techniques that stop one of the exploitation stages (described in Figure 2.2) can prevent the exploits from infecting victim machines. Some exemplar defenses include:

1) Cloaking the OS/software version during the pre-exploitation stage to stop attackers from launching the correct exploits.

2) Tools like BrowserShield [26] instrument the execution of JavaScript to match the predefined vulnerability feature and block the execution once a match is found.

3) Randomization-based techniques like Readactor [92] try to stop the attacks by preventing memory disclosures.

4) Tools like ROPecker [55] exploits the Last Branch Record hardware feature to detect the execution of ROP chains.

5) Control Flow Integrity (CFI) [27] based techniques [38, 50] are used to prevent execution of injected payloads.

While these exploitation mitigation techniques are constantly improving, hacking contests like Pwn2Own [31], GeekPwn [32], etc., consistently demonstrate that the latest versions of Chrome, Safari, Internet Explorer, and Edge can still be exploited. There are two reasons for this: First, most of the latest proposed mitigation techniques require software or compiler tool chain changes which may cause compatibility issues and thus cannot be deployed promptly. Second, the deployed mitigation techniques may fail due to newly invented exploitation techniques. For instance, DEP mitigation can be defeated by ROP attacks. The JITSpray [93] makes the ROP defense useless since ROP is not needed anymore to bypass DEP in JITSpray based attack. To conclusion, we argue that *an ideal mitigation technique should be easy to deploy without the change of code, and undermine the inevitable exploitation stages*.

5.2.2 Memory Preparation

Based on our observations, the *memory preparation* stage is a critical stage for JavaScript exploits because attackers need to put something (e.g., ROP, shellcode) into in a known memory location prior to execution of that code. The memory preparation step can take many forms like well-known heap spraying. While most of these techniques are difficult to detect with high confidence, it often offers the first chance for defenders to detect that something malicious is happening.

Memory Management of the JavaScript Engine Before the discussion of memory preparation techniques, we first take V8 [94] as an example to present an overview of memory management within a JavaScript engine. JavaScript engines dynamically manage memory for running applications so that developers don't need to worry about memory management like

coding in C/C++. V8 divides the heap into several different spaces- *a young generation, an old generation, and a large object space.* The young generation is divided into two contiguous spaces, called semi-space. The old generation is separated into a map space and an old object space. The map space exclusively contains all map objects while the rest of old objects go into the old space.

Each space is composed of a set of pages. A page is a contiguous chunk of memory, allocated from the operating system with system call(e.g., mmap). Pages are always 1 MB in size and 1 MB aligned, except in a separate large object space. This separate space stores objects larger than *Page::kMaxHeapObjectSize*, so that theses large objects are not moved during garbage collection process.

The allocated objects will be put into different spaces based on their size, type, and age. Garbage collection process are responsible for 1) scavenging young generation space my moving live objects to the other semi-space when semi-space becomes full; 2) major collection of the whole heap to free unreferenced objects and aggressive memory compaction to clean up fragmented memory. The other JavaScript engines like SpiderMonkey [76], ChakraCore [95], JavaScriptCore [96], etc.. share the similar design of memory management. Attackers commonly abuse the memory management features to manipulate the memory states, known as memory preparation.

Memory Preparation techniques Memory allocation and free operations are used to manipulate the memory. We can categorize these techniques based upon how they change the memory layout:

1) Emit data in a target address. This is usually implemented with a heap spray technique like Heap Fengshui [97] and its successors [98]. These techniques spray crafted objects into the

heap. The size and type of the objects are carefully chosen to exploit the memory management of the JavaScript engine. ❶

2) Emit objects adjacent in memory. This is implemented by allocating two objects with the same type and size sequentially so the JavaScript engine will likely keep them adjacent in the heap; this technique is widely used by attackers. ❷

3) Create holes in memory. This is implemented by allocating adjacent objects first, then freeing one of them. A hole is created among those adjacent objects. ❸

4) De-fragment the heap. This is usually implemented via calling a garbage collection API provided by host software (e.g., *CollectGarbage()* in IE) or via a carefully crafted JavaScript snippet that forces the garbage collection process as discussed in Section 5.4.1.❹

In theory, attackers can use any JavaScript types to prepare memory, but in practice *String* and *Array* are used most frequently. This is because the implementations of these two types, especially *Typed Array*, are very close to native arrays in C/C++. Compared with the other primitive types (*Boolean, Null, Number, Symbol*) and objects (e.g., *Math*), it is much easier to control the content and layout of memory with these two types.

Realizing that memory preparation is an essential step towards successful JavaScript exploit, a natural question rises in our mind - How can we disrupt this stage without code change of the host software? The answer is: *memory perturbation*.

Table 5.1: The overview of memory perturbation techniques

Category	Approaches	Memory Change	Code Transformation Overhead
Storage	a. Split Variables	1 , 2	High
	b. Change Encoding	1	High
	c. Promote scalars to objects	1 , 2	High
	d. convert static data to procedure	1, 2	Medium
Aggregation	e. Merge scalar variables	1,2	High
	f. Split, merge, fold, flatten arrays	2	High
Ordering	g. Reorder instance variables	2	Medium
	h. Reorder arrays	2	High
Inserting	i. Insert noise data allocation/free	2	Low
	j. Insert holes into arrays	2	High

1: content change 2: layout change

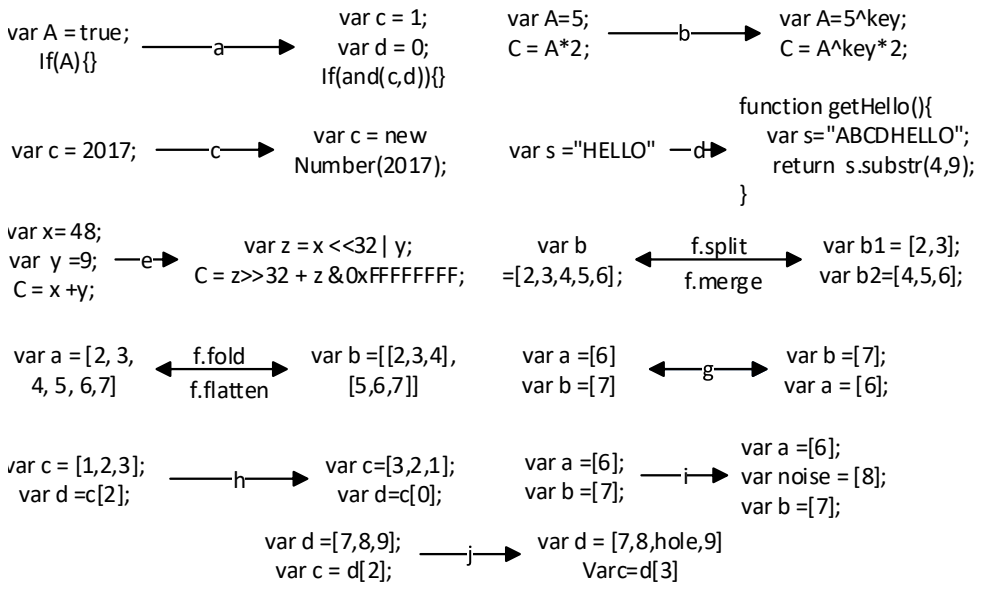

Fig. 5.1.: Samples of memory perturbation techniques summarized in Table 5.1

5.2.3 Memory Perturbation Techniques

Memory perturbation involves manipulation of the content or layout of memory without changing the semantics of the code. This technique is similar to the data obfuscation technique used in code obfuscation [88] to defeat malicious reverse engineering attacks. However, in the

context of JavaScript-based exploit defense, our goal is to subvert the memory preparation stage through memory perturbation, so that exploits are defeated at runtime due to unpredictable memory states.

Table 5.1 provides an overview of memory perturbation techniques. In general, memory perturbation techniques can be divided into 4 categories as affecting the storage, aggregation, ordering or inserting of the data in memory. Figure 5.1 presents sample code snippets for each of the approaches referenced in Table 5.1. While each of the approaches can induce similar memory changes in terms of memory layout or content, the overhead associated with each of these approaches is quite different. For instance, Approach *i* only needs one statement insertion operation for the code transformation. It does not need any further program analysis to keep the semantics intact since the inserted statement does not affect the original code's data and control flow. However, Approach *f* requires additional program analysis to keep the program semantics intact. This is because after the array is restructured, a whole program def-use analysis has to be conducted to identify all the affected code and then update the code accordingly (update array index, array name, etc.). Column D of Table 5.1 presents the code transformation overhead for each approach.

5.2.4 Our Mitigation Solution

Based upon the previous discussion, we proposed a vulnerability-agnostic defense approach for JavaScript exploits -CHAFFYSCRIPT. The basic idea of CHAFFYSCRIPT is to *sabotage the memory preparation stage via memory perturbation without changing the original JavaScript's semantics*. Specifically, given suspicious JavaScript, CHAFFYSCRIPT rewrites the code to insert

chaff code to perturb memory states, and then generates semantically-equivalent code. Since the chaff code changes the memory states at runtime, JavaScript exploits will fail as a result of unexpected memory states, while the benign JavaScript still executes as expected since the transformed code by CHAFFYSCRIPT is semantics-equivalent. Compared with current mitigation techniques, CHAFFYSCRIPT has the following advantages:

1) Vulnerability-agnostic nature. CHAFFYSCRIPT does not rely on any specific vulnerability features as BrowserShield [26] does. Thus it is vulnerability-agnostic and can be used to defend against 0-day attacks.

2) Flexible deployment. JavaScript rewriting can be implemented without the change of host software. This makes the deployment of CHAFFYSCRIPT very flexible. Users can disable or enable CHAFFYSCRIPT promptly based upon their needs.

3) Stronger protection. As evaluated in Section 5.6.1, CHAFFYSCRIPT provides much stronger protection than randomization-based approaches (e.g., ASLRGuard [33]).

In the following sections, we will elaborate details on the threat model and scope, design and implementation of CHAFFYSCRIPT.

5.3 Threat Model and Scope

We assume a commodity operating system with standard defense mechanisms, such as no-executable stack and heap, and ASLR. We assume attackers are remote, so they do not have physical access to the target system, nor do they have prior control over other programs before a successful exploit.

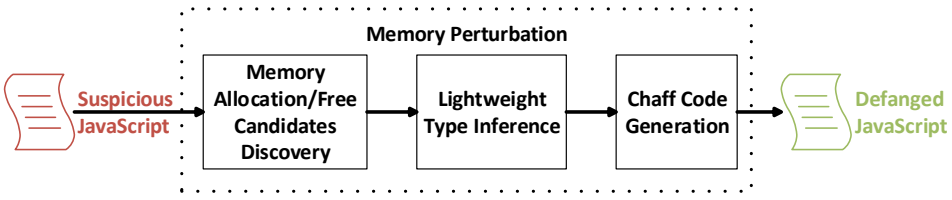

Fig. 5.2.: The overall architecture of CHAFFYSCRIPT.

We assume an adversary uses JavaScript to exploit memory corruption vulnerabilities in a program and achieve arbitrary code execution. We also assume that the adversary uses memory preparation to manipulate the memory layout. This is a fair assumption since memory preparation is a general stage used in the exploitation. We impose no restrictions on the exploitation techniques.

Random number is used by CHAFFYSCRIPT to increase the difficulty of guessing the correct memory states after memory perturbation techniques are deployed as discussed in Section 5.4.3. So one assumption of CHAFFYSCRIPT is that the attacker cannot compromise random number generator. This is a fair assumption since different secure random number generators [99] have been deployed to stop such kind of attacks.

CHAFFYSCRIPT works against JavaScript exploits that require precise memory preparation. These exploitations include but are not limited to control data attack, non-control data attack [100], and side channel attack [56].

Out-of-scope threats. Cross-site scripting (XSS) [101] and Cross-site forgery(CSRF) [102] are out of our scope.

5.4 Design

In this section, we present the design of CHAFFYSCRIPT. Based on previous discussion, we know that a successful JavaScript-based exploit requires sophisticated memory preparation. So the goal of CHAFFYSCRIPT is to: *undermine the memory preparation via memory perturbation without changing the original JavaScript's semantics*.

In theory, all of the memory perturbation techniques discussed in Section 5.2.3 could be used to undermine the memory preparation stages of JavaScript exploits. That said, to be practical an on-line defense approach should incur minimal code transformation overhead and thus cannot afford complex program analysis. With that in mind, we only apply Approach *i* in CHAFFYSCRIPT for memory perturbation, which is good enough for defeating JavaScript exploits as demonstrated in Section 5.6. Specifically, CHAFFYSCRIPT conducts JavaScript rewriting by inserting chaff code to 1) allocate random chunks of memory along with existing memory allocation operations, and 2) disable memory free operations by adding additional reference to freed objects.

Figure 5.2 demonstrates the overview of CHAFFYSCRIPT. Given a suspicious JavaScript, it first traverses the code to discover memory allocation/free candidates. Then a lightweight type inference process is conducted on these candidates to identify the interesting memory allocation/free candidates that are usually used for memory preparation by attackers; this reduces unnecessary chaff code insertions and improves runtime performance. Finally, the chaff code is generated and inserted into the original JavaScript code to get a *defanged JavaScript*. At runtime, the chaff code will allocate random memory or disable memory free operations to undermine the

memory preparation stage of JavaScript-based exploits. Benign JavaScript still executes normally since the chaff code does not change the original code's expected semantics.

Next, we will discuss the detailed design of the three CHAFFYSCRIPT modules: memory allocation/free candidate discovery, lightweight type inference, and chaff code generation.

5.4.1 Memory Allocation/De-Allocation Candidate discovery

As discussed in Section 5.2, there are two kinds of operations that affect the memory state: object allocation and de-allocation. CHAFFYSCRIPT inspects JavaScript code to identify the following memory manipulation candidates:

Memory Allocation Candidates As discussed in Section 5.2.2, *String* and *Array* are two common data types used by attackers to fill memory. CHAFFYSCRIPT traverses JavaScript code to identify potential memory allocation candidates for *String* and *Array* operations. However, precise type inference for JavaScript is quite expensive [103]. So, to be simple, the *new* expression (e.g. *var c = new Array(5)*), value initialization expression (e.g.*var c = [3,7]*), and built-in function callsite (e.g., *var c = a.substr(0,10)*) are all considered as memory allocation candidates since they can trigger memory allocation in the heap. Note that the callee name can be dynamically generated in JavaScript. Thus, we cannot statically determine if it is a targeted built-in function callsite. To be complete, the callsites with dynamically generated callees are also considered as memory allocation candidates. In JavaScript *String* objects are immutable, so operations that change *String* objects (e.g. - the *'+'* and *'+='* operators) will also cause memory allocation. CHAFFYSCRIPT also considers expressions with the *'+'* and *'+='* operators as potential memory allocation candidates.

Memory De-allocation Candidates In JavaScript, there are three ways to explicitly free memory: assign *null* to the object, use the *delete* operator and explicitly trigger garbage collection. The first two methods remove the reference to the allocated object. For instance, *delete object.property* removes the reference to the *property* object. However, it does not directly free the *property* object in memory. When the *property* object is no longer referenced by any other objects, garbage collection process will eventually free it in memory. CHAFFYSCRIPT still considers *null* assignment and *delete* calls as memory de-allocation candidates because attackers often use them to create holes in memory, so objects allocated later can fill these holes to trigger some vulnerabilities (e.g.,Use-after-Free).

Explicit garbage collection (GC) calls are usually used by attackers to defragment the heap [1]. Some browsers like Internet Explorer and Opera provide public APIs (e.g., *CollectGarbage()* for IE) to trigger garbage process. CHAFFYSCRIPT can easily identify this kind of garbage collection process by matching the API name. However, the other browsers' garbage collection process can only be triggered when certain memory states are achieved; in these cases there are no APIs to explicitly trigger the process. In these cases, attackers usually fill objects in memory to trigger the garbage collection process. For instance, the following code can fill up the 1MB semi-space page of V8 engine and force V8 to scavenge *NewSpace*.

```
1 for(var i=0; i<((1024*1024)/0x10); i++)
2 {
3   var a = new String();
4 }
```

This kind of GC trigger is implicit and difficult to identify. Nevertheless, it includes memory allocation operations and will be considered as a *memory allocation candidate* and will still be captured by CHAFFYSCRIPT. GC events using DOM objects instead of *String* and *Array* are not captured by CHAFFYSCRIPT, but this is only one of the memory preparation techniques used by attacker. Furthermore, JavaScript exploits usually combine multiple memory preparation techniques, thus allowing CHAFFYSCRIPT to be effective even when hybrid memory preparation methods are used.

The host software also provides APIs to allocate and free objects. For instance, on browser, users can adopt the DOM API to add or remove node objects from the DOM tree. While in theory, it is possible for attackers to manipulate those APIs during memory preparation, as discussed in Section 5.2.2, it is challenging to do that since the layout and content of DOM objects are difficult to control. If new memory preparation techniques are applied by attackers, CHAFFYSCRIPT just needs to update the memory allocation/de-allocation candidate discovery stage to block those techniques.

5.4.2 Lightweight Type Inference

The collection of memory allocation candidates is a superset of the memory allocation candidates of *String* and *Array* objects. If we insert chaff code along with all the candidates, the runtime performance would be unacceptable. To improve the runtime performance of defanged JavaScript, we conduct lightweight type inference to prune the memory allocation candidates that are not related to *String* and *Array* objects. It is executed as two steps: *static type inference* and *dynamic type inference*.

Static type inference CHAFFYSCRIPT only keeps the memory allocation candidates that operate on variables typed as one of *String, Array, ArrayBuffer, Int8Array, Uint8Array, Uint8ClampedArray, Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, and Float64Array*. We can infer the types of the variables in expressions statically based upon how they are used. CHAFFYSCRIPT uses the three following type inference rules:

1) the constructor of the *new* operator indicates the type of created object. For instance, the constructor *Int16Array* in *var b = new Int16Array(256)* indicates the type of *b* is *Int16Array*.

2) the return value of a built-in function indicates the type based on its description. For instance, *var c = s.split("a")* indicates that the type of *c* is *Array*.

3) for expressions with the *'+' or '+='* operators, if the type of the operands are *String*, then the result is also a *String*.

These three simple rules do not require complex program analysis and can be used to efficiently determine the type of variables to filter out the memory allocation candidates. If static type inference cannot determine the types of all the variables used in memory allocation candidates, dynamic type inference are conducted to check the type at runtime.

Dynamic type inference Static type inference does not always work for two reasons. First, since a function call's name can be dynamically generated, we cannot determine an object's type based upon the function's return value. Second, the three typing rules are very weak and cannot determine the type of variables in some cases. For instance, the three typing rules cannot be applied on candidates var d = a + b + c. It is possible to determine the type of *a, b and c* via

backward analysis, but that is likely too expensive for our online defense system use case. Instead, we conduct *dynamic type inference* with the help of JavaScript features. In JavaScript, a variable's type can be extracted at runtime using the *instanceof* operator. With this operator, CHAFFYSCRIPT inserts the dynamic type inference after the memory allocation candidate to check if it operates on targeted types. While dynamic type inference incurs runtime performance overhead, it is less expensive than statically inferring the type of variables.

Our static type inference is very conservative; any untyped variables are enforced to conduct dynamic type inference. As a result, the dynamic type inference makes it impossible for attackers to bypass our type inference process.

5.4.3 Chaff Code Generation

The goal of inserted chaff code is to affect the memory states at runtime. It achieves this goal via the following two methods.

disable memory free operations. For memory free candidates using public APIs like *CollectGarbage()* in IE, CHAFFYSCRIPT directly removes the API call from the original code. This does not change the semantics of original JavaScript code because garbage collection APIs does not have data or control dependency on the original code.

For memory free candidates using the *delete* operator or assigning a *null* value, the above method does not work because simply removing such code will change the semantics of the original code. The attackers' goal of freeing an object is to create holes in memory, so later allocated objects can occupy the freed memory. If we keep a reference to the object before the free operation is executed, the later allocated object can not occupy the position since the allocated

memory still has a reference to it. Figure 5.7.(a,b) illustrates an example of such transformation. In code snippet *[a]*, *x* is assigned the value *null*. In the transformed code *[b]*, CHAFFYSCRIPT has added a new variable *4613335ea9901* to store a reference to *"abcdefgh"*. Although *x* is assigned to *null*, the object *"abcdefgh"* will not be scavenged since *4613335ea9901* keeps a reference to it.

```
var x = "abcdefgh";
x = null;
console.log(x);
```

[a]

```
var x = "abcdefgh";
var 4613335ea9901 = x;
x = null;
console.log(x);
```

[b]

```
var y = new ArrayBuffer(1024);
```

[c]

```
var y = new ArrayBuffer(1024);
var e512da1951cdd7 = new ArrayBuffer(y.byteLength + RANDOM);
93f208876cd2bd.push(e512da1951cdd7);
```

[d]

Fig. 5.3.: Chaff code Samples

Insert chaff code following the memory allocation candidate. As discussed in Section 5.2, if two objects of the same type are allocated sequentially and with the same size, it is likely their positions on heap are adjacent to each other. Attackers exploit this feature to manipulate the memory layout and content. CHAFFYSCRIPT is also able to exploit this detail to defeat this exploit. After every memory allocation candidate, CHAFFYSCRIPT insert code to allocate additional memory with the same type, but with variable-length padding at the end(*RANDOM*). Figure 5.7([c],[d]) presents an example of such code transformation. As you see in code snippet *[d]*, a new *ArrayBuffer* is allocated with size *(y.byteLength + RANDOM)*. With this randomness, it is almost impossible for attackers to guess the combinations of memory states as evaluated in Section 5.6.1.

To generate *RANDOM*, CHAFFYSCRIPT provides two approaches to increase an attacker's uncertainty. The first one is at runtime; every time the inserted chaff code is executed, a new random number is generated for *RANDOM*. So if a given piece of chaff code is executed 1000 times, and the range of *RANDOM* is *0-50*, it will create 50^{1000} possible memory states at runtime. The second approach happens when we are inserting the chaff code into original JavaScript code; a random number is generated for *RANDOM*. This value is used to randomly increase the size of the memory chunk allocated by this piece of chaff code. For example, if 15 places are inserted by the chaff code, and the range of *RANDOM* is *0-50*, 50^{15} possible memory states will be created at runtime. Both approaches can defeat the JavaScript exploits with different security guarantees and performance overhead as discussed in Section 5.6.

The range of *RANDOM* cannot be too big. Otherwise, the objects allocated by chaff code might be allocated in a different location. Thus that fails to break adjacent arrays (❷). In our implementation, we set it as 0-50. This range works against memory preparation techniques while providing enough randomness as demonstrated in Section 5.6.1.

Since our inserted code is independent of the original code, attackers may abuse garbage collection to scavenge the allocated chaff memory and neutralize the effects of inserted chaff code. To avoid this, CHAFFYSCRIPT keeps a reference to every allocated piece of chaff memory. This prevents scavenging of chaff memory by the garbage collection process because there is always at least one reference to the allocated chaff memory. We call the added code as *GC escaper*.

The variable names used in the chaff code are generated randomly. This prevents attackers from identifying memory allocated by the chaff code alongside their own memory preparation

code. Thus attackers cannot leverage variable naming conventions to identify memory used as part of our countermeasures, thus making bypass of these countermeasures impossible.

5.5 Implementation

We implemented CHAFFYSCRIPT using esprima [104]. It is a JavaScript parser used to generate Abstract Syntax Tree (AST) with full support for ECMAScript 6. Since it is written using JavaScript, it can be easily embedded into different documents like HTML or PDF. This makes the deployment very flexible. Estraverse [105] is used to traverse the AST, discover memory allocation/free candidates and perform lightweight type inference. The chaff code insertion is implemented via directly manipulating the original code with the offset information collected from AST generation process. We do not operate on AST directly to insert the chaff code because generation code from AST is more expensive than from manipulating original code. Section 5.6.3 evaluates the difference of rewriting performance for these two approaches.

The general workflow of CHAFFYSCRIPT can be summarized in the following steps:

1) CHAFFYSCRIPT takes JavaScript code as input and derives its AST.

2) CHAFFYSCRIPT traverses the generated AST to discover memory allocation/free candidates and conduct lightweight type inference.

3) CHAFFYSCRIPT generates chaff code snippets and inserts them into the original JavaScript code to generate the defanged JavaScript.

The deployment of CHAFFYSCRIPT is very flexible. It can be deployed as a browser extension, a web proxy, a standalone rewriting engine or one component of a JavaScript engine.

In this section, we demonstrated one deployment approach to protect users against malicious HTML files.

5.5.1 HTML Protector

Ideally, it is most user-friendly to deploy CHAFFYSCRIPT as a browser extension. Unfortunately, JavaScript rewrite, used to implement code transformation in CHAFFYSCRIPT, is not natively supported in browser extensions. Instead, we deploy CHAFFYSCRIPT as a web proxy. The downside is that a user needs to install an external program (and certificate) as opposed to only an extension. The benefit is that this proxy-based solution is browser-independent and can be easily deployed with minimal configuration.

The prototype was implemented in *Node.js* [106], using the *http-mitm-proxy* package [107]. CHAFFYSCRIPT becomes an integral part of the Web proxy. We followed Dachshund [53]'s approach to handle dynamically generated code. Specifically, the dynamic code generation functions (e.g.,*eval, SetTimeout, Function, SetInterval*) were hooked via new injected JavaScript code. To rewrite dynamically generated code, we used synchronous *XMLHttpRequest* requests from hooked JavaScript functions to the proxy. The response from the proxy contains the *defanged JavaScript* code.

5.5.2 Possible implementation of PDF protector

While we did not, it is possible to integrate CHAFFYSCRIPT as a standalone rewrite engine to protect users against malicious PDF files. We can adopt *peepdf*[108] to extract JavaScript from PDF files and write the defanged JavaScript back. For dynamically generated code, we can use

the same approach as HTML protector. For instance, *Net.HTTP* can be used to pass dynamically generated code to the rewrite engine, and retrieve the defanged code.

The attacker may embed malicious JavaScript in an unusual way as demonstrated in [109] to evade the JavaScript extraction. Thus the embedded malicious JavaScript will be ignored by *peepdf*, and *PDF protector* fails to protect users against such malicious PDF files. It is always possible for attackers to find new methods to hide the malicious JavaScript code in PDF files. However, this is not a weakness of CHAFFYSCRIPT. Extraction of JavaScript from malicious PDF files is another security problem and out of CHAFFYSCRIPT's scope. Future work could look at leveraging new JavaScript extraction techniques to improve the implementation of *PDF protector*. For instance, one solution is to leverage the state-of-art JavaScript Extractor developed in [109] to confirm that *peepdf* has extracted all of the embedded JavaScript code in a PDF file.

5.6 Evaluation

In this section, we present the evaluation of CHAFFYSCRIPT. The evaluation tries to answer the following questions:

1) How secure is CHAFFYSCRIPT's approach in theory, compared to general randomization approaches?

2) How secure is CHAFFYSCRIPT's approach against practical JavaScript-based exploits?

3) How much overhead does CHAFFYSCRIPT impose, in particular rewriting, runtime, and space?

Experimental setup. We conducted our evaluations of CHAFFYSCRIPT to check its security enhancements and potential performance impacts. The performance overhead experiment was run under Chrome 57 and Firefox 54. All the experiments are conducted on a test machine equipped with intel Core i7-4790 CPU @ 3.60GHz 8 with 16GB RAM.

5.6.1 Security Analysis

In this subsection, we present an analysis to determine the probability that an attacker could predict the memory layout after memory perturbation is introduced by CHAFFYSCRIPT. The randomness introduced by CHAFFYSCRIPT is determined by the following parameters:

1) *RANDOM* - the size variation range of created object by chaff code.

2) *M* - number of inserted chaff code.

3) *N* - executed times of chaff code at runtime.

The probability of guessing the correct memory states is defined as the follow equation.

$$probability = \begin{cases} RANDOM^{-M} & predefine\ RANDOM \\ RANDOM^{-N} & Gen\,RANDOM\,at\,runtime \end{cases}$$

If *RANDOM* is predefined when CHAFFYSCRIPT inserts the chaff code, the *probability* is $RANDOM^{-M}$. If a random number is generated for *RANDOM* every time the chaff code is executed, the *probability* is $RANDOM^{-N}$. If it is too big, the allocated objects by chaff code may not be adjacent to the objects allocated by original code. Thus it cannot break memory preparation ❷. In our implementation, we set *RANDOM* as 50 and it worked well on defeating JavaScript exploits as evaluated in Section 5.6.2.

The value of *M* and *N* is case by case. Column 4 and Column 5 in Table 5.2 records the value of *M* and *N* for 10 exploits. The average of *M* was 15, and average of *N* was 130876. The *probability* for JavaScript exploits in our implementation should be 50^{-15} and $50^{-130876}$. This provides much stronger randomness than ASLR [33] (2^{-64} at most). The highest *probability* of the 10 exploits was 50^{-10} which is still stronger than 2^{-56}. Through this analysis, we conclude that the probability for an attacker to predict the memory layout is extremely low after memory perturbation is introduced by CHAFFYSCRIPT.

Table 5.2: Experimental results of 10 latest JavaScript-based exploits using CHAFFYSCRIPT

CVE	setup	MemoryPreparation	M	N	Defeated?
CVE-2015-2419	IE11 32bit WIN7	❶ ❷ ❹	28	12594	Y
CVE-2015-1233	chrome 41.0.2272.118 WIN10 32bit	❶ ❷ ❹	9	8194	Y
CVE-2015-6086	IE11 32bit WIN7	❸ ❹	12	1280	Y
CVE-2015-6764	chrome 46.0.2490.0 WIN10 32bit	❶ ❷ ❹	14	393224	Y
CVE-2016-9079	FireFox 50.0.1 32bit Windows8.1	❶ ❷ ❹(JITSpray)	13	20564	Y
		❶ ❷ ❹	28	17408	Y
ms16-063 (cve-2016-3202)	IE11 WIN7 32bit	❶ ❷ ❸ ❹	12	110005	Y
CVE-2016-1646	chrome46.0.2490.0 WIN10 32bit	❶ ❷ ❹	18	393226	Y
chromev8 OOB write	chrome60.0.3080.5 linux14.04 64bit	❷	10	10	Y
X360_videoPlayerActiveX	VideoPlayerActiveX 2.6 IE10 WIN7 64bit	❶ ❷	9	352258	Y

M: # of inserted chaff code N: Executed times # of chaff code at runtime

5.6.2 Effectiveness

Although in theory CHAFFYSCRIPT can stop JavaScript exploits, we wanted to know how well it performed at defeating real JavaScript exploits without the knowledge of the targeted

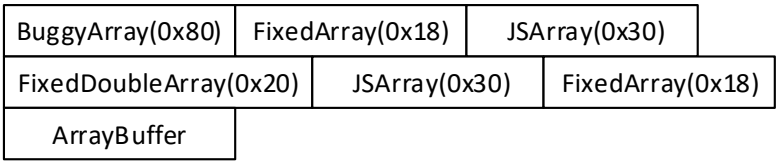

Fig. 5.4.: Expected memory layout of sample chromev8_OOB_write

vulnerabilities. To do that, we set up exploitation environments for 10 exploits and confirmed that without CHAFFYSCRIPT, all 10 of these exploits functioned correctly. These 10 exploits are representative of latest JavaScript exploits because:

1) The vulnerabilities targeted by these 10 exploits are quite new (from 2015 to 2016).

2) The target host software of these exploits covered the most popular web browsers (IE11, Chrome, and Firefox)

3) These 10 exploits used all of the memory preparation techniques presented in Column 3 in Table 5.2.

4) These 10 exploits covered the popular exploitation techniques - JITSpray and HeapSpray.

5) These 10 exploits not only targeted at vulnerabilities in host software, but also in the browser plugin (X360_videoPlayerActiveX).

For every exploit, we manually confirmed that it could be stopped using CHAFFYSCRIPT. Column 6 in Table 5.2 presents the result. As shown in the results, CHAFFYSCRIPT defeated all 10 of the exploits without requiring knowledge of vulnerabilities targeted. This experiment proves that CHAFFYSCRIPT can effectively defeat JavaScript exploits without knowledge of the targeted vulnerability.

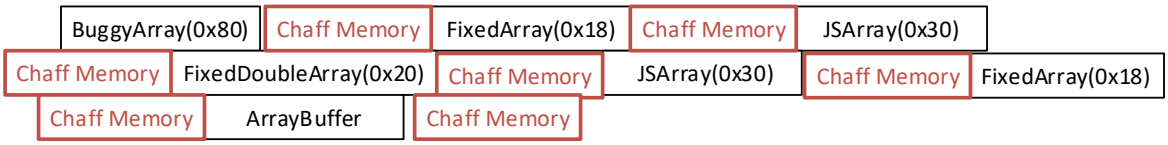

Fig. 5.5.: Memory layout of sample chromev8_OOB_write after rewritten by CHAFFYSCRIPT

Note that the chaff codes inserted into the sample *chromev8_OOB_write* are only executed 10 times, but still thwart the exploit. This is not like the other samples, in which the chaff codes are executed thousands of times and change the memory states substantially. In fact, the memory preparation of this exploit expects the memory layout as shown in Figure 5.4. After the chaff code is inserted, the actual memory layout is close to the layout as shown in Figure 5.5. The chaff memory breaks adjacent array layout(❷). Therefore, the memory preparation of this exploit fails and this exploit is thwarted eventually. This case further demonstrates the effectiveness of memory perturbation used in CHAFFYSCRIPT since it uses very few memory perturbation operations.

5.6.3 Performance

Rewriting Overhead In order to evaluate the rewriting overhead of CHAFFYSCRIPT, we chose to measure the three popular and large JavaScript libraries - JQuery (mobile-1.4.2), AngularJS (1.2.5), and React (0.13.3). These libraries are commonly embedded in web pages and relatively large compared with other JavaScript applications (JQuery has 443KB, AngularJS has 702KB, React has 587KB). For the evaluation, we rewrote these libraries using CHAFFYSCRIPT 1000

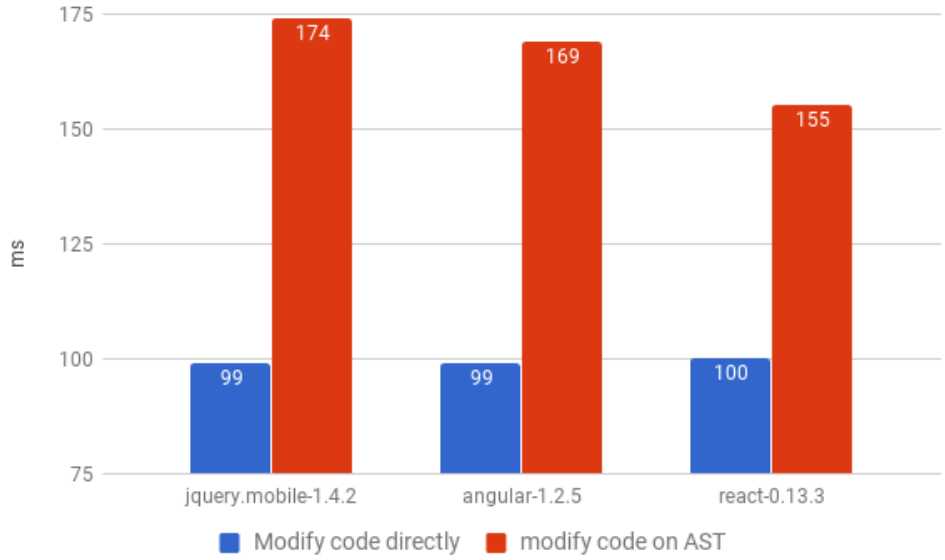

Fig. 5.6.: Rewriting Performance on well-known JavaScript libraries

times. We measured the time required to rewrite these libraries, including all the steps required to generate defanged JavaScript.

We tested two code transformation approaches. The first approach modified the code directly based upon the offset information collected by the JavaScript parser. The second approach modified code within the AST. As demonstrated in Figure 5.6. The time spent by the second approach is 1.67 times more than the first approach. CHAFFYSCRIPT chose the first approach in the implementation. On average, It took $\tilde{1}$00ms to rewrite JQuery, AngularJS, and React. Note that rewriting is a one-time effort and we can further optimize performance by rewriting multiple scripts concurrently.

Runtime Overhead Next, we evaluated the runtime performance that is incurred on the client side due to the modified JavaScript code. We leverage Octane, a commonly-used benchmark for

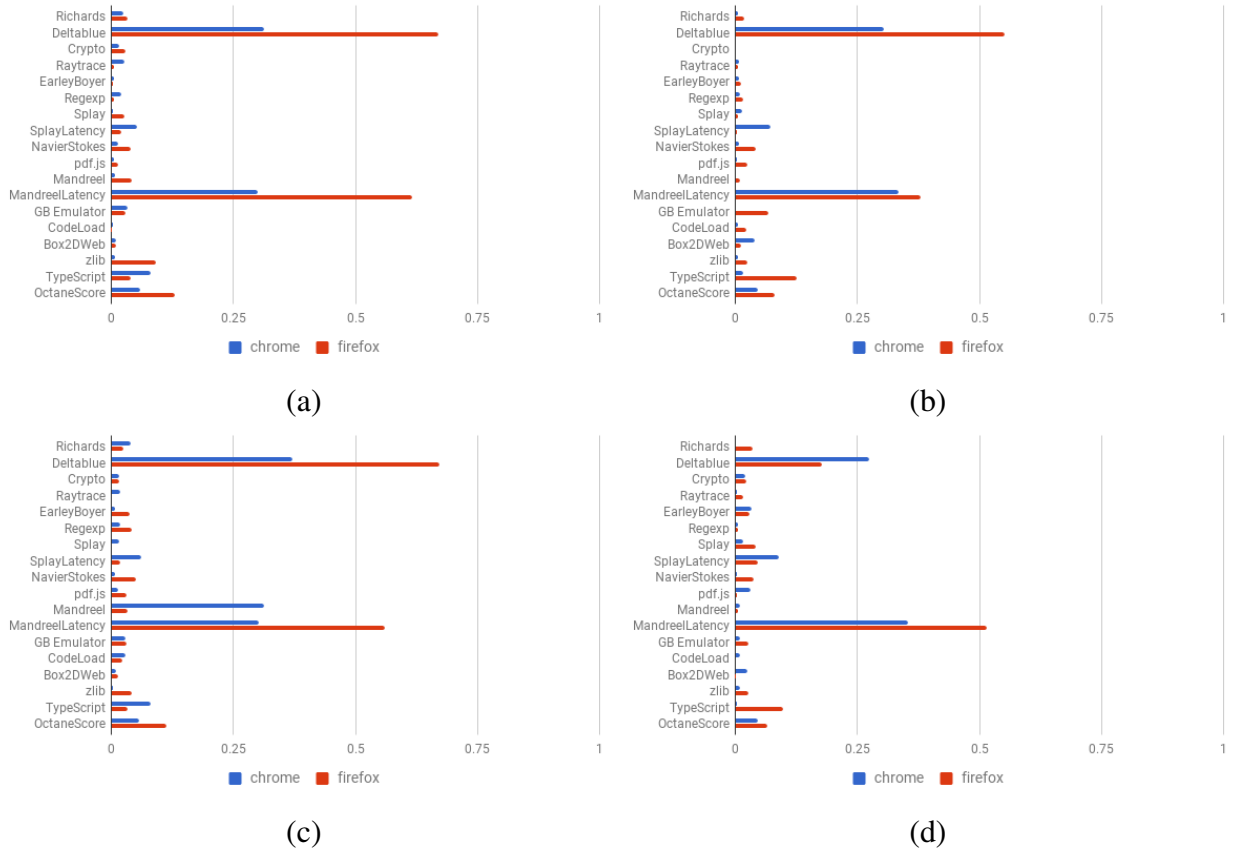

Fig. 5.7.: Runtime performance overhead under different configurations

Table 5.3: Overall Overhead of CHAFFYSCRIPT on Octane benchmark

Cofiguration	Chrome	Firefox
a. Runtime Generated *RANDOM* + GC escaper	5.88%	12.96%
b. Runtime Generated *RANDOM*	4.54%	7.98%
c. Predefined *RANDOM* + GC escaper	5.68%	11.27%
d. Predefined *RANDOM*	4.53%	6.60%

JavaScript engines [91]. For the evaluation, we ran the Octane benchmarks 5 times and used the mean scores as the final results.

Figure 5.7 illustrates the breakdown of Octane benchmark results. The performance varies quite a lot for different applications. This is because the runtime overhead is mainly caused by the inserted chaff code and the number of times that it is executed at runtime, which is very

Table 5.4: Memory Overhead of Chrome on Octane benchmark

Usage	Original	a	b	c	d
Min(MB)	34.4	36.5(6.10%)	36.5(6.10%)	36.5(6.10%)	36.4(5.81%)
Max(MB)	609	659(8.2%)	656(7.71%)	640(5.09%)	638(4.76%)

a, b, c , d refers to the four CHAFFYSCRIPT configurations described in Table 5.3

application dependent. Table 5.3 summarizes the overall overhead on the Octane benchmark under different CHAFFYSCRIPT configurations. With the strongest protection, CHAFFYSCRIPT incurs 5.88% overhead in Chrome, and 12.96% in FireFox. With the weakest protection, CHAFFYSCRIPT incurs 4.53% in Chrome and 6.60% in Firefox. As discussed in Section 5.6.1, the weakest protection can still provide an acceptable randomness strength. Note that our threat model is only relevant to non-trusted and attacker-controlled JavaScript. Thus the overhead of popular JavaScript libraries can be eliminated by whitelisting trusted scripts. This overhead after the whitelisting should allow CHAFFYSCRIPT to be deployed online to protect users against JavaScript exploits.

Memory Overhead In theory, the memory overhead should be around 2 times at most. This is because along with each object allocation, CHAFFYSCRIPT could allocate another object with a similar size to disturb memory states. If all of the inserted objects by chaff code are not freed finally by Garbage Collector, the memory usage of the defanged JavaScript should be 2 times of the original JavaScript.

To evaluate the actual memory overhead, we ran Octane on Chrome and recorded the memory usage of JavaScript heap. Table 5.4 summarizes the results. *Min* refers to the observed minimal memory usage of JavaScript heap during the running of Octane, while *Max* refers to the observed maximal memory usage. As demonstrated in the table, for all four different CHAFFYSCRIPT

configurations, the memory overhead never exceeded 8.2%. This is not a big overhead since RAM has become very cheap and current personal computers are usually equipped with at least 8GB memory. Thus, CHAFFYSCRIPT can be deployed by users without requiring upgraded hardware.

6. SUMMARY AND FUTURE WORK

6.1 Conclusion

The original thesis was that enriched executions can be leveraged to combat malicious JavaScript. More Specifically, the proposed techniques explore arbitrary paths for detection, preserve JS-binary semantics for diagnosis, and perturb memory with chaff code for mitigation. This thesis has been validated in this dissertation.

JSForce demonstrated that it could tolerate invalid object accesses while increasing code coverage and introducing no runtime errors during execution. As an amplifier technique, JSForce does not rely on any predefined profile information or full-fledged hosting programs like browsers or PDF viewers, and it can examine partial JavaScript snippets collected during an attack. JSForce can be leveraged to greatly improve the detection rate of other dynamic analysis systems without modification of their detection policies.

JScalpel showed that by bridging the semantic gap between the JavaScript level and binary level to perform dynamic JS-Binary analysis, we are able to determine the payload injection and exploitation statements of the JavaScript exploits. Thus a minimized exploit and POV can be automatically created for exploit signature generation and penetration test.

ChaffyScript illustrated that an enriched execution with memory perturbation could undermine the memory preparation stage in JavaScript exploits and thus blocks the attack. Moreover, the deployment of ChaffyScript is very flexible and requires no code change of host software.

```
var f = function() {
    if (true) {
        function g(){return 1;}
    } else {
        function g(){return 2;}
    };
    function g(){return 3;};
    return g();
    function g(){return 4;}
}
```

Fig. 6.1.: A JavaScript Sample Interpreted Differently by Different JavaScript Engines

The core of *Enriched Executions* is composed of these three techniques to combat malicious JavaScript. They can be deployed independently on different server or together on the same server. More specially, JSForce and JScalpel can be deployed on the server to provide off-line analysis to capture and diagnose malicious JavaScript. ChaffyScript can be deployed either on the end user's machine or on the web proxy server to protect against malicious JavaScript. *Enriched Executions* combines these three techniques together to effectively defeat malicious JavaScript.

6.2 Limitations and Future Work

6.2.1 Detection

If the syntax of the tested JavaScript code is not correct, JSForce drops the analysis immediately. The forced execution can introduce syntax error under some cases. For instance, the parameter of `eval` is supposed to be correct JavaScript code. When the parameter is calculated from faked strings created by JSForce, the parameter may become syntax incorrect for `eval`. In the future, we expect to develop techniques [110] to automatically fix the syntax error to enable maximized execution of the code.

While JavaScript language has the official specification from the ECMAScript community [68], the implementation of the language on different JavaScript engines differs

slightly because of the complex features and rapid evolving of JavaScript language. The attacker can exploit this weakness to create a deliberate script which exhibits differently on JSForce to evade the analysis. Maffeis et al. [111] discussed such an example presented in Figure 6.1. This code defines a function `f` whose behavior is given by one of the declarations of `g` inside the body of the anonymous function that returns `g`. However, different implementations disagree on which declaration determines the behavior of `f`. Specifically, a call to `f()` should return 4, according to ECMA specification. SpiderMonkey returns 4, while Rhino and Safari return 1, and JScript and the ECMA4 reference implementations return 2. Attackers can leverage these differences to hide the decoding key and evade analysis. To counter this, we can implement JSForce on top of different JavaScript engines, such as SpiderMonkey [76] and Chakra [95].

The current path exploration algorithm can efficiently explore most of the sample in a decent time. However, there are still some cases that take a considerable length of time to finish. To exploit this limitation, attackers may place the malicious code deep in the code logic, such that JSForce could not reach it within a predefined duration. Note that this limitation is not unique for JSForce. All the path exploration techniques share the same limitation. We leave it as future work to develop better path exploration algorithms and search heuristics.

JSForce can be evaded by techniques that do not need control-flow branches, e.g., those based on browser or JavaScript quirks. For example, the property $window.innerWidth$ is defined in FireFox and Chrome but undefined in Internet Explorer. Therefore, a malicious code that computed a decoding key from $window.innerWidth$ would obtain a different result in Firefox/Chrome and IE, and could be used to decode malicious code in specific browsers. JSForce will not trigger the malicious code path in such cases and can be evaded.

```
1  xxxxx = 'ev';
2  yyyyy = 'al';
3  zzzzz = xxxxx + yyyyy;
4  aaaaa = app;
5  try {} catch (e) {
6      zzzzz = 1;
7      aaaaa = 1;
8  }
9  try {
10     d = nothis_nothis;
11     zzzzz = 1;
12     aaaaa = 1;
13 } catch (e) {}
14 aaaaa[zzzzz]('ddddd' + 'dd=une' + 'sca' /**/ + /**/ 'pe;');
```

Fig. 6.2.: The Case of Evading JSForce

Figure 6.2 presents an example that can bypass JSForce. Using normal JavaScript engine, lines 6-7 and lines 11-12 will not be executed so that $aaaaa$ and $zzzzz$ can be correctly initialized. However, with JSForce, line 10 will not raise the exception since d will be initialized as $FakedObject$. The exponential path exploration can handle this case by exploring all the possible path combinations. But it is not feasible in practice. Our solution is that we execute the sample without forced execution when we collect the path predicates at the beginning.

6.2.2 Diagnosis

Vulnerabilities within Filtered Modules If the vulnerability exploited exists within the filtered modules, the slicer produces the incomplete slice. Current implementation of JScalpel can not detect exploits that target the filtered modules. In the future, fine-grained analysis can be applied on these modules to determine which part of the code introduces the dependency and then limit

the filter from whole module to some specific code range. This will reduce the number of vulnerabilities that JScalpel cannot handle.

Debug-Resistant JavaScript In order for JScalpel to be able to analyze a script, it is important that JScalpel executes the program and monitors from the debugger. Though we did not find any samples that can detect debuggers, it is possible that exploits could use techniques (e.g., timing-based) to determine if a debugger is running and hide the malicious behavior. Currently, JScalpel is vulnerable to such attacks. It would be an interesting future work to reconstruct JavaScript-level semantics directly from the Virtual Machine Monitor, similar to how DroidScope [112]) recovers Java/Dalvik level semantic view.

Impact of JIT-Enabled JavaScript Engine on JScalpel When JIT is enabled on JavaScript engine, the data flow within JavaScript engine becomes more complex because of the mixture of code and data. JScalpel may not work in this case. Since JScalpel is designed as an analysis tool and is not performance sensitive, the analyst can simply disable the JIT engine. However, this workaround would sacrifice the capability of analyzing attacks that perform JIT spray, as these attacks rely on the side-effects of the JIT compiler. We leave it as future work to address this issue.

6.2.3 Mitigation

In general, ChaffyScript has the following limitations:

First, attackers may find methods to bypass the JavaScript rewriting process. For instance, lexer confusing attacks [113] confuse the lexer causing executable code to be interpreted as the content of strings or comments, allowing an attacker to slip arbitrary unsafe code past a rewriter

or verifier. The rewriter of ChaffyScript is vulnerable to this attack. In the future, we would like to adopt JaTE's [114] approach by considering all formats of JavaScript comments to gain resilience to this attack. It is also possible to attack the JavaScript parser *esprima* with crafted JavaScript. As a result, the parser will fail and further code transformation cannot be conducted by ChaffyScript. Fortunately, ChaffyScript can identify such attacks because it can detect *esprima* errors. ChaffyScript can alert security researchers for further analysis once such failures are observed.

Second, the JavaScript extraction approach used in deployment may undermine ChaffyScript. Attackers may hide JavaScript in an unusual way to escape from the extraction, thus preventing ChaffyScript from rewriting those portions. For instance, attackers may abuse PDF parsers to hide malicious JavaScript code [109]. This is not a ChaffyScript issue, but rather is a JavaScript extractor issue. Deployment of a state-of-art JavaScript extractor with ChaffyScript would reduce the risk of such attacks.

Third, ChaffyScript does not work on hybrid JavaScript exploits. Basically, such kind of exploits use JavaScript to trigger the vulnerability, and use other script language (e.g., ActionScript in Flash) to prepare the memory. This is quite common in recently years since vector-related rehabilitates in Flash are quite exploit-friendly, allowing construction of arbitrary memory read/write primitives [115]. However, it is possible to deploy the techniques used in ChaffyScript on ActionScript to stop such attacks as discussed in the following subsection.

Fourth, attackers may find other objects to prepare the memory instead of *String* and *Array* operations targeted by ChaffyScript. Once these new memory preparation techniques are identified, ChaffyScript just needs an update to its memory allocation/free candidate discovery process to reflect the new memory preparation technique.

Applicability on the other script-based exploits

JavaScript is not the only script language that can be used to launch exploits; other script languages like VBScript [116] and ActionScript [117] are commonly used to launch exploits. These script-based exploits are widely used to create malicious Microsoft Documents (word, excel, powerpoint,etc.), flash files, web pages [118]. So the questions rises in our mind - Can ChaffyScript be applied to stop the other script-based exploits?

The answer is yes because of the following reasons:

1) Script-based languages share a similar memory management approach. They all use some sort of garbage collector to recycle the memory and conduct automatic garbage collection at runtime.

2) Memory preparation is a general stage in exploits. Attackers require this stage to bypass mitigation techniques like ASLR, CFG [38] with crafted memory. This stage is also used by the other script-based exploits.

3) Other script-based languages also execute interpretively and can be rewritten as JavaScript. This allows ChaffyScript to provide the protection via rewriting.

To demonstrate this, we set up the exploitation environment for CVE-2016-0189 [119]. It is a VBScript-based exploit targeting a VBScript memory corruption in IE11. We manually applied ChaffyScript's rewriting process and insert the memory perturbation code. The test shows that this exploit is successfully blocked by memory perturbation. This demonstrates that ChaffyScript can also be applied to stop the other script-based exploits.

To apply ChaffyScript on the other script language, we need the corresponding script parser, and also need to adapt the lightweight typing rules on the new script language since different languages support different typing systems. These adaptations are feasible and can be implemented with engineering efforts.

References

1. M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript.," *WOOT*, vol. 8, pp. 1–6, 2008.

2. F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

3. C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, *et al.*, "Manufacturing compromise: the emergence of exploit-as-a-service," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 821–832, ACM, 2012.

4. "2015 symantec internet security threat report." `http://www.symantec.com/security_response/publications/threatreport.jsp`.

5. M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

6. G. Lu and S. Debray, "Automatic simplification of obfuscated javascript code: A semantics-based approach," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012.

7. B. Hartstein, "Jsunpack: An automatic javascript unpacker," in *ShmooCon convention*, 2009.

8. B. Gu, W. Zhang, X. Bai, A. C. Champion, F. Qin, and D. Xuan, "Jsguard: Shellcode detection in javascript," in *Security and Privacy in Communication Networks*, 2013.

9. C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

10. P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proceedings of the Usenix Security Symposium*, 2009.

11. C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection.," in *USENIX Security Symposium*, 2011.

12. A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware.," in *USENIX Security*, pp. 637–652, Citeseer, 2013.

13. Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *the Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2014.

14. B. Feinstein, D. Peck, and I. SecureWorks, "Caffeine monkey: Automated collection, detection and analysis of malicious javascript," *Black Hat USA*, 2007.

15. P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques.," in *MALWARE*, pp. 47–54, Citeseer, 2009.

16. C. Seifert, I. Welch, and P. Komisarczuk, "Identification of malicious web pages with static heuristics," in *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian*, pp. 91–96, IEEE, 2008.

17. W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pp. 9–16, IEEE, 2012.

18. R. Upathilake, Y. Li, and A. Matrawy, "A classification of web browser fingerprinting techniques," in *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, IEEE, 2015.

19. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

20. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proc. 29th ACM Symposium on Applied Computing*, 2014.

21. M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis," in *Proceedings of 19th Annual Network & Distributed System Security Symposium*, 2012.

22. J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2005.

23. K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "Shellos: Enabling fast detection and forensic analysis of code injection attacks.," in *USENIX Security Symposium*, 2011.

24. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, (New York, NY, USA), 2007.

25. A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

26. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," *ACM Transactions on the Web (TWEB)*, 2007.

27. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, ACM, 2005.

28. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity.," in *OSDI*, vol. 14, p. 00000, 2014.

29. V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing.," in *USENIX Security*, vol. 30, p. 38, 2013.

30. V. Pappas, "kbouncer: Efficient and transparent rop mitigation," *tech. rep. Citeseer*, 2012.

31. "Pwn2own." `https://en.wikipedia.org/wiki/Pwn2Own`.

32. "Geekpwn." `http://2017.geekpwn.org/1024/en/index.html`.

33. K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 280–291, ACM, 2015.

34. C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, "Manufacturing compromise: The emergence of exploit-as-a-service," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

35. "Cve details." `http://www.cvedetails.com/`.

36. "Rop is dying and your exploit mitigations are on life support." `https://www.endgame.com/blog/technical-blog/rop-dying-and-your-exploit-mitigations-are-life-support`.

37. "The enhanced mitigation experience toolkit." `https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit`.

38. "Control flow guard." `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`.

39. S. Labs, "Malware with your mocha? obfuscation and anti-emulation tricks in malicious javascript.." http://www.sophos.com/en-us/why-sophos/our-people/technical-papers/malware-with-your-mocha.aspx.

40. "Plugindetect: Browser plugin detector.." http://www.pinlady.net/PluginDetect/.

41. K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013.

42. G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *ACM Sigplan Notices*, vol. 45, pp. 1–12, ACM, 2010.

43. J. G. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of javascript sandboxing," *arXiv preprint arXiv:1506.07813*, 2015.

44. A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical javascript apis," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 363–378, IEEE, 2011.

45. S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: Vetting browser extensions for security vulnerabilities.," in *USENIX Security Symposium*, vol. 10, pp. 339–354, 2010.

46. D. Liu, H. Wang, and A. Stavrou, "Detecting malicious javascript in pdf through document instrumentation," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 2014.

47. K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

48. S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Formal reasoning of various categories of widely exploited security vulnerabilities using pointer taintedness semantics," in *Security and Protection in Information Processing Systems*, 2004.

49. A. Slowinska and H. Bos, "The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack," in *23rd Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

50. C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm.," in *USENIX Security Symposium*, pp. 941–955, 2014.

51. C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 559–573, IEEE, 2013.

52. A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 311–322, ACM, 2013.

53. G. Maisuradze, M. Backes, and C. Rossow, "Dachshund: Digging for and securing against (non-) blinded constants in jit code," 2017.

54. X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.," in *NDSS*, 2015.

55. Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG, *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

56. B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu," *NDSS (Feb. 2017)*, 2017.

57. F. Gadaleta, Y. Younan, and W. Joosen, "Bubble: A javascript engine level countermeasure against heap-spraying attacks," in *International Symposium on Engineering Secure Software and Systems*, pp. 1–17, Springer, 2010.

58. L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 481–496, IEEE, 2010.

59. K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," *Proceedings of W2SP*, vol. 2, 2011.

60. M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, 2013.

61. M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1232–1243, ACM, 2014.

62. F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*, 2014.

63. Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 44–56, ACM, 2015.

64. "V8 javascript engine." https://code.google.com/p/v8/.

65. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, pp. 65–88, Springer, 2008.

66. D. Y. Wang, S. Savage, and G. M. Voelker, "Cloak and dagger: dynamics of web search cloaking," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 477–490, ACM, 2011.

67. "Sputnik." https://code.google.com/p/sputniktests/.

68. `http://www.ecmascript.org/`.

69. P. Thiemann, "Towards a type system for analyzing javascript programs," in *Programming Languages and Systems*, pp. 408–422, Springer, 2005.

70. U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP'91 European Conference on Object-Oriented Programming*, Springer, 1991.

71. "Virus total." `https://www.virustotal.com/`.

72. `http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html`.

73. `http://malware-traffic-analysis.net/`.

74. `http://threatglass.com/`.

75. `http://www.alexa.com/topsites`.

76. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

77. M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, IEEE Press, 1981.

78. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, 2002.

79. A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

80. "Active script debugging overview." http://msdn.microsoft.com/en-us/library/z537xb90(v=vs.94).aspx.

81. N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *arXiv preprint arXiv:1602.04056*, 2016.

82. F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, 2015.

83. A. Slowinska and H. Bos, "Pointless tainting? evaluating the practicality of pointer tainting," in *Proceedings of the 4th ACM European conference on Computer systems*, ACM, 2009.

84. "The T.J. Watson Libraries for Analysis (WALA)." http://wala.sourceforge.net/.

85. H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

86. B. Eshete, A. Alhuzhali, M. Monshizadeh, P. Porras, and V. Yegneswaran, "Ekhunter: A counter-offensive toolkit for exploit kit infiltration," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, Feb 2015.

87. "National vulnerability database." "https://nvd.nist.gov/".

88. C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

89. "Detailed analysis exp/20111255-a." http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Exp~20111255-A/detailed-analysis.aspx.

90. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 79–93, IEEE, 2009.

91. "The javascript benchmark suite for the mordern web." https://developers.google.com/octane/.

92. S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 763–780, IEEE, 2015.

93. A. Sintsov, "Writing jit-spray shellcode for fun and profit," *Writing*, 2010.

94. "Chrome v8 engine." `https://developers.google.com/v8/`.

95. `https://github.com/Microsoft/ChakraCore`.

96. "Javascriptcore." `https://trac.webkit.org/wiki/JavaScriptCore`.

97. A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

98. "The art of leaks: The return of fengshui." `https://cansecwest.com/slides/2014/The%20Art%20of%20Leaks%20-%20read%20version%20-%20Yoyo.pdf`.

99. "Random number generator attack." `https://en.wikipedia.org/wiki/Random_number_generator_attack`.

100. Y. Yu, "Write once, pwn anywhere," *BlackHat*, 2014.

101. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 330–337, ACM, 2006.

102. A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 75–88, ACM, 2008.

103. C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *European conference on Object-oriented programming*, pp. 428–452, Springer, 2005.

104. "Ecmascript parsing infrastructure for multipurpose analysis." `http://esprima.org/`.

105. "Ecmascript js ast traversal functions." `https://github.com/estools/estraverse`.

106. "Node.js." `https://nodejs.org/en/`.

107. "Http mitm proxy." `https://github.com/joeferner/node-http-mitm-proxy`.

108. "Peepdf: a python tool to explore pdf files." `https://github.com/jesparza/peepdf`.

109. C. Carmony, X. Hu, H. Yin, A. V. Bhaskar, and M. Zhang, "Extract me if you can: Abusing pdf parsers in malware detectors.," in *NDSS*, 2016.

110. D. T. Barnard and R. C. Holt, "Hierarchic syntax error repair for lr grammars," *International Journal of Computer & Information Sciences*, vol. 11, no. 4, pp. 231–258, 1982.

111. S. Maffeis, J. C. Mitchell, and A. Taly, "An operational semantics for javascript," in *Programming languages and systems*, pp. 307–325, Springer, 2008.

112. L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

113. "Lexer confusing attack." `https://github.com/google/caja/wiki/JsControlFormatChars`.

114. T. Tran, R. Pelizzi, and R. Sekar, "Jate: Transparent and efficient javascript confinement," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 151–160, ACM, 2015.

115. "Aslr bypass apocalypse in recent zero-day exploits." `https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`.

116. "Vbscript." `https://en.wikipedia.org/wiki/VBScript`.

117. "Actionscript technology center." `http://www.adobe.com/devnet/actionscript.html`.

118. "Akbuilder is the latest exploit kit to target word documents, spread malware." `https://nakedsecurity.sophos.com/2017/02/07/akbuilder-is-the-latest-exploit-kit-to-target-word-documents-spread-malware/`.

119. "Proof-of-concept exploit for cve-2016-0189 (vbscript memory corruption in ie11)." `https://github.com/theori-io/cve-2016-0189`.

VITA

Xunchao Hu was born in Feicheng, Shandong Province, China. He received his Bachelor of Science degree in Software Engineering and Master of Science degree in Control Science and Engineering from Xi'an Jiaotong University, China. He received his PhD in Electrical and Computer Engineering from Syracuse University in December 2017.