NANYANG TECHNOLOGICAL UNIVERSITY

Detection and Analysis of Web-based Malware and Vulnerability

Wang Junjie

School of Computer Science and Engineering

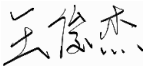
A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2018

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

24 Dec 2018



. . . . . .

Date

. . . . . .

Junjie Wang

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

24 Dec 2018

. . . . . .
Date

. . . . . .
Yang Liu

Authorship Attribution Statement

This thesis contains material from 4 paper(s) published in the following peer-reviewed journal(s) where I was the first and/or corresponding author.

Chapter 3 is published as **Junjie Wang**, Yinxing Xue, Yang Liu, and Tianhuat Tan. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification, In Proceedings of the 10thACM AsiaConference onComputer andCommunications Security (AsiaCCS 2015), 14-17 April 2015, Singapore.

The contributions of the co-authors are as follows:

- A/Prof. Yang Liu provided the initial project direction.
- Dr.Yinxing Xue edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr Tianhuat Tan.
- I co-designed the study with Dr.Yinxing Xue and performed all the laboratory work at the School of Computer Science and Engineering. I also analyzed the data.

Chapter 4 is published as Yinxing Xue, **Junjie Wang**, Yang Liu, Hao Xiao, Jun Sun and Mahinthan Chandramohan, JS*: Detection and Classification of Malicious JavaScript via Attack Behavior Modelling, In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 12-17 July 2015, Baltimore, Maryland.

The contributions of the co-authors are as follows:

- A/Prof.Yang Liu provided the initial project direction.
- Dr.Yinxing Xue edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr.Hao Xiao, Dr.Jun Sun and Dr.Mahinthan Chandramohan.
- I co-designed the study with Dr.Yinxing Xue and performed all the laboratory work at the School of Computer Science and Engineering. I also analyzed the data.

Chapter 5 is published as **Junjie Wang**, Bihuan Chen, Lei Wei, and Yang Liu, Skyfire: Data-Driven Seed Generation for Fuzzing, In Proceedings of 38th IEEE Symposium on Security and Privacy (S&P 2017), May 22-24 2017, San Jose, CA.

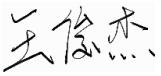
The contributions of the co-authors are as follows:

- Dr.Lei Wei provided the initial project direction.
- Dr.Bihuan Chen edited the manuscript drafts.
- The manuscript was revised by A/Prof.Yang Liu.
- I co-designed the study with Dr.Lei Wei and performed all the laboratory work at the School of Computer Science and Engineering. I also analyzed the data.

Chapter 6 is published as **Junjie Wang**, Bihuan Chen, Lei Wei, and Yang Liu, Superion: Grammar-Aware Greybox Fuzzing, In Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019), May 25-31 2019, QC Canada.

The contributions of the co-authors are as follows:

- Dr.Lei Wei provided the initial project direction.
- Dr.Bihuan Chen edited the manuscript drafts.
- The manuscript was revised by A/Prof.Yang Liu.
- I co-designed the study with Dr.Lei Wei and performed all the laboratory work at the School of Computer Science and Engineering. I also analyzed the data.

24 Dec 2018 

. . . .

Date Junjie Wang

THESIS ABSTRACT

Detection and Analysis of Web-based Malware and Vulnerability

by

Wang Junjie

Doctor of Philosophy
School of Computer Science and Engineering
Nanyang Technological University, Singapore

Since the dawn of the Internet, all of us have been swept up by the Niagara of information that fills our daily life. In this process, browsers play an extremely important role. Modern browsers have turned from a simple text displayer to a complicated software that supports rich user interfaces and a variety of file formats and protocols. This enlarges the attack surface and makes browsers one of the main targets of cyber attack.

Inside the Internet security, JavaScript malware is one of the major threats. They exploit vulnerabilities in the browsers to launch attacks remotely. To protect end-users from these threats, this thesis makes two main contributions: identifying JavaScript malware and detecting vulnerabilities in browsers, which aim at a complete solution for Internet security.

In identifying JavaScript malware, we first propose to classify JavaScript malware using the machine learning approach combined with dynamic confirmation. Static and dynamic approaches both have merits and drawbacks. Dynamic approaches are effective while not scalable. Static approaches are efficient but normally suffer from a high false negative ratio. To identify JavaScript malware effectively and efficiently, we propose a two-phase approach. The first phase lightweight classifies JavaScript malware from benign web pages. Then the second phase further subdivides the attack behaviors of JavaScript malware. We implement our approach as an online tool and conduct a large-scale experiment to show its effectiveness.

Towards an insightful analysis of JavaScript malware evolution trend, it is desirable to further classify them according to the exploited *attack vector* and the corresponding *attack behaviors*. Considering the emergence of numerous new JavaScript malware and their variants, such an automated classification can significantly speed up the overall response to the JavaScript malware and even shorten the time to discover the zero-day attacks. We propose to use the Deterministic Finite Automaton (DFA), to summarize patterns of malware. Our approach can automatically learn a DFA from the dynamic execution traces of JavaScript malware. The experiment results demonstrate that our approach is more scalable and effective in JavaScript malware detection and classification, compared with other commercial anti-virus tools.

Through previous two works, we realized that the root cause of the prevalence of JavaScript malware is the existence of vulnerabilities in browsers. Therefore, finding vulnerabilities in browsers and improving mitigation is of significant importance. We propose a novel data-driven seed generation approach to test the core components of browsers, especially XML engines and XSLT engines. We first learn a Probabilistic

Context-Sensitive Grammar (PCSG) from a large number of samples of one specific grammar. The feature of PCSG can help us to generate samples whose syntax and semantics are correct with high probability. The experimental results demonstrate that both the bug finding capability and code coverage of fuzzing are advanced.

We further improve coverage-based greybox fuzzing by proposing a new grammar-aware approach for programs that process structured inputs. In details, our approach requires the grammar of test inputs, which is often publicly available. Based on the grammar, we propose a grammar-aware trimming strategy to trim test inputs at the tree level. Besides, we introduce two grammar-aware mutation strategies (i.e., enhanced dictionary-based mutation and tree-based mutation). Tree-based mutation works by replacing sub-trees of the Abstract Syntax Tree (AST) of parsed test inputs. With grammar-awareness, we can effectively mutate test inputs while keeping the input structure valid, quickly carrying the fuzzing exploration into width and depth. We conduct experiments to evaluate the effectiveness of it on one XML engine, libplist and two JavaScript engines, WebKit, and Jerryscript. The results demonstrate that our approach outperforms other fuzzing tools in both code coverage and the bug-finding capability.

Contents

List of Figures

List of Tables

Acknowledgment

Firstly, I would like to give my sincere gratitude to Prof.Yang Liu, my supervisor who, with extraordinary patience and consistent encouragement, gives me great help by providing me with necessary materials, the advice of great value and inspiration of new ideas.

With a special mention to Dr.Lei Wei, Dr.Bihuan Chen, Dr.Yinxing Xue, Dr.Naipeng Dong, and Xiaoning Du. It was fantastic to have the opportunity to work the majority of my research in your facilities. I also thank everyone in the Cyber Security Lab. It was great sharing laboratory with all of you during the last four years.

I am grateful to my parent, my husband, and my brother, who have provided me with moral and emotional support in my life. I am also grateful to my other family members and friends who have supported me along the way.

A very special gratitude goes out to all down at research foundation for helping and providing the funding for the work.

1

Introduction

The Internet is increasingly changing how we conduct our daily routines. Day-to-day operations that only can be done through personal contact before, such as banking, shopping, or communication, now can be done online, a seemingly simpler and better alternative. Although more and more network traffic is flowing into mobile applications, web browsers still are the most significant interface allowing us to access the Internet. According to the Internet World Stats, there are already 3.424 billion active Internet users by July 2018, and that number continues growing.

1.1 Motivations and Goals

The prosperity of the Internet attracts plenty of unlawful hackers to seek illegal profits. Exploiting the browsers is easier to conduct than other attacks which need to launch

locally. An attacker can remotely steal a user's private information or money. Since he is hiding behind thousands of dynamic IP addresses, he can easily cover his identities and are hard to be tracked down. As a result, the number of emerging malicious websites is startlingly increasing.

On the other hand, to compete with other rivals and gain market share, browsers are heavily optimized for best performance. These features made browsers more complicated and difficult to maintain. JavaScript is a powerful client-side scripting language that provides dynamic content for websites. Many high-profile websites, such as Google and Facebook, make heavy use of JavaScript to enhance their service. Nevertheless, the execution of JavaScript at the client side allows the language to directly communicate with the browser, and other third-party plugins, which makes it vulnerable for exploitation. In fact, the JavaScript malware represents one of the most successful mass-scale cyber attacks happening nowadays.

Unfortunately, the security of JavaScript malware is not so concerned and well-studied as the traditional end-host malicious executables. Knowing the emerging attack models in the new Internet environment is vital to malware detection, privacy protection, and cybercrime prevention. Another new threat that aggravates the spread and infection of malware is the outburst of malware variants. Just like the biological virus, the transient and various variants make the vaccine almost impossible. The variants of malware could deeply hide its malicious behaviors and evade the detection of anti-virus products due to obfuscation.

There are two categories of JavaScript malware detection approaches: dynamic approaches and static approaches. Dynamic approaches are mostly based on low interaction honey clients [1] or high-interaction honey clients [2, 3]. The low-interaction honey client emulates a normal browser. They make use of specifications to detect malicious JavaScript. The low-interaction honey client approach is restricted by the provided specifications. An attack that is unspecified would be missed.

A high-interaction honey client uses a web browser running in a virtual machine to record all changes a web page caused to the environment. The changes include process

creation, modification of files and so on. If unexpected changes occurred, it is considered as the manifestation of an attack, and the corresponding web page is classified as a malicious web page. The problem of high-interaction honey client lies in the difficulty of setting up a virtual environment that captures the exploitations of all possible plugins. It requires all plugins to be installed properly. Given the vast amount of possible plugins, and each of them has specific requirements on the browser and operating system, it is almost impossible to set up a high-interaction honey client, that could capture the exploitations of all possible plugins (for each of their configurations). Therefore, this approach could easily miss a genuine attack. Most importantly, honey clients are extremely resource intensive, and it is not affordable for individuals or small research groups.

Static approaches do not require the actual execution of samples. Most users typically rely on signature-based anti-virus products to detect malware. The anti-virus tools normally extract features that discriminate malicious web pages from benign web pages for classification. To evade the detection of anti-virus, JavaScript malware makes use of sophisticated obfuscation techniques, such as dynamic code generation, to fail the static signature-based detection methods.

Static and dynamic approaches both have merits and drawbacks. Dynamic approaches normally are accurate but not efficient. They are usually designed for some specific kinds of attacks. No general approach can detect all kind of JavaScript malware. The detection also takes longer time than machine learning approaches because dynamic approaches need to executing the samples and execution normally requires a considerable time overhead. Machine learning or code similarity analysis based static approaches are efficient but have a high false negative ratio. Even worse, machine learning based approaches can neither express attack behaviors nor identify new attacks from emerging malware.

To protect end-users from JavaScript malware, we made our efforts from two aspects: identifying JavaScript malware and detecting vulnerabilities in browsers. Malware exploits vulnerabilities to compromise end-users and vulnerabilities facilitate malware.

They are indispensable parts of web-based attacks. To sum up, we have the following main goals:

- **G1. Effectively and efficiently detect JavaScript malware.** We aimed at combining advantages of static and dynamic approaches to achieve both accuracy and speed.
- **G2. Analyze and understand the behaviors of detected JavaScript malware.** Beyond mere detection, we also aim at understanding behaviors and targets of JavaScript malware, which is helpful to security experts. We aim to automatically extract and classify the JavaScript malware, which will significantly reduce our response time to new JavaScript malware variants.
- **G3. Detect vulnerabilities in browsers to reduce the likelihood of being attacked.** We believe that the root cause of the prevalence of malware is the existence of vulnerabilities in browsers. Therefore, finding vulnerabilities in browsers and improving mitigation is of significant importance.

1.2 Main Works and Contributions

To achieve three main goals in Section 1.1, we conducted four correlated works. The first two works are to identify JavaScript malware, as shown in Figure 1.1, the last two works are to detect vulnerabilities in browsers, as shown in Figure 1.2. We first detect JavaScript malware using machine learning combined with dynamic confirmation. Then we further model JavaScript malware according to their behaviors and targets. To eliminate threats fundamentally, we first generate more well-distributed seeds to test browsers, and then we improve browser fuzzing by making them grammar-aware.

1. JavaScript Malware Detection via Machine Learning

Generally, a JavaScript malware detection approach needs to solve the following challenges. **Obfuscation:** both benign and malicious JavaScript code adopt some obfuscation techniques. Purely static detection is incapable of identifying effective features

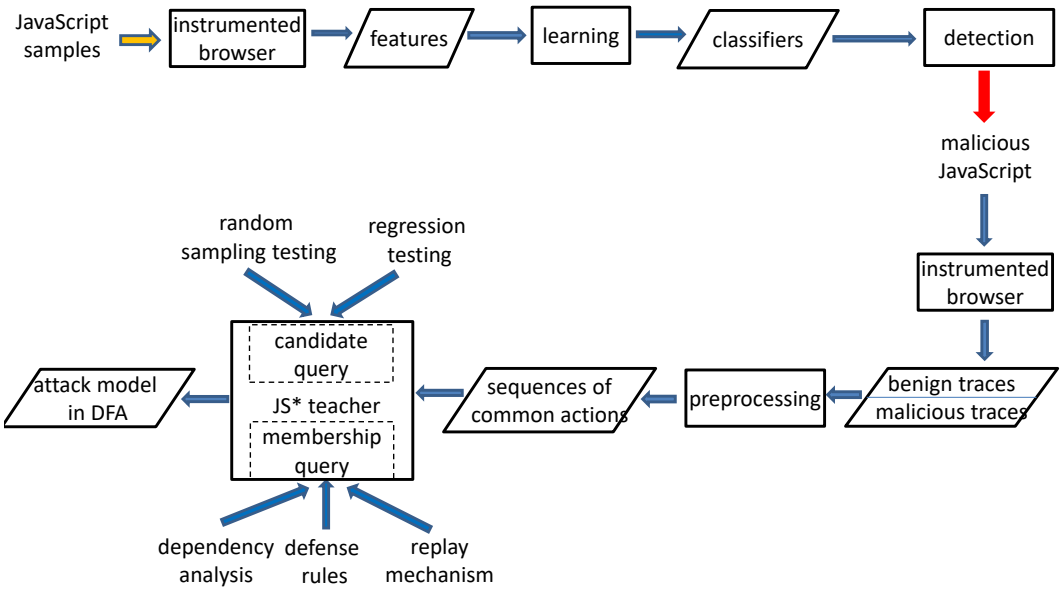


Figure 1.1: The Overview of JavaScript Malware Detection

without exposing the original code. **Transience:** Any off-line detection needs to deal with the transience of the JavaScript malware. Malware will be removed by site administer or go off-line after some period of time. To compute the transience of malicious sites, Literature [4] re-detected the set of malicious URLs detected by Nozzle on the previous day. This procedure is repeated for three weeks (21 days). Authors found that nearly 20% of malicious URLs were gone after a single day and after 13 days 80% of them were gone. **Dynamics:** The dynamic feature of JavaScript allows it to execute a portion of code generated at runtime. **Cloaking:** Certain JavaScript attacks are environment-specific, targeting a particular browser and often attacking specific versions of installed plugins. **Scalability:** Dynamic analysis is accurate but it is not as efficient as static analysis. For real-time or large-scale malware detection, scalability is a key factor. **Accuracy:** For detection of general JavaScript malware (not certain specific types), it is not easy to achieve both a low false positive ratio ($< 5\%$) and a low false negative ratio ($< 5\%$) at the same time [5].

In Chapter 3, we combine static analysis and dynamic analysis to balance their advantages and disadvantages. We propose a two-phase approach and in the first phase, we apply machine learning to detect malicious JavaScript fragments. In the second phase, we classify detected malicious JavaScript fragments into eight attack types. For those

suspicious JavaScript fragments which fall into the grey zone, we employ dynamic confirmation to check whether they are new attacks or false positives.

To sum up, the first work makes the following main contributions:

- We combine machine learning with dynamic analysis to get better accuracy and scalability. The features we propose include textual information and function call patterns. We also label each function call with the security level.
- We use our approach to detect 20,000 websites, which include about 1,400,000 JavaScript fragments. The results demonstrate the effectiveness and efficiency of our approach.

2. JavaScript Malware Behaviors Modelling

Besides mere detection, none of the previously mentioned work from security domain targets at attack analysis and modeling. Knowing attack details helps to understand the characteristics of various attacks. We first label each JavaScript malware sample according to their attack behaviors, then learn a model in the form of DFA from each attack type. We classify JavaScript malware into eight types, therefore, we learn eight different DFAs.

There are two challenges to JavaScript behaviors modeling:

- JavaScript can dynamically generate source code and change object type, which toughen the static approach. To model JavaScript malware, we need a compact representation of execution trace and effective learning approach.
- Current JavaScript malware family name is not based on malware's behavior. For example, *Trojan.js.iframe.** family launches various types of attacks. A meaningful model that can express the common behavior of JavaScript malware is of great value.

In Chapter 4, to overcome the above-mentioned challenges, we propose a dynamic analysis approach, named JS*, to automatically learn a DFA for each JavaScript attack type,

which is inspired by the research work on program behavior modeling [6]. The input accepted by DFA includes possible malicious execution traces of the same type of attack, which models the variants of the same attack. First, given a set of (both malicious and benign) execution traces of JavaScript malware, we conduct a preprocessing step to reduce these traces by removing security-irrelevant browser-level system calls. Then, the left security-relevant system calls are grouped into high-level *actions* that represent a meaningful and security-relevant behavior of JavaScript code. Subsequently, we extract the common actions that appear in the malicious execution traces and extract the system call based execution traces as action sequences. Second, we develop an online learning algorithm based on the L^* algorithm [7], which employs dynamic execution traces to learn a DFA of the JavaScript malware. The DFA accepts concrete malicious behaviors of variants of the same attack type.

Our main contributions are listed as follows:

- We treat detection and classification of JavaScript malware as a behavior modelling problem. We propose to use DFA to model distinct attack types instead of behaviors of various JavaScript malware families.
- We propose JS* as a dynamic tool to automatically learn an abstracted DFA from hundreds of malicious JavaScript samples on the fly. The learned DFA can be used for detecting JavaScript malware variants.
- Based on real-world JavaScript malware, we summarize eight different attack types involving 120 malicious JavaScript samples and learn their attack models. We evaluate JS* with 10,156 real-world JavaScript samples.
- The implementation of our *polynomial-time* teacher is different from that in previous work [8] on behavior modeling of Java programs using L^*.

Figure 1.2 shows the workflow of our two vulnerability detection works in sequence. The third work includes the learning of Probabilistic Context-Sensitive Grammar (PCSG for short) model and using it to generate well-distributed seeds. The fourth work mainly mutates seeds at AST level and tackles some problems in fuzzing stage.

Fuzzing is an automatic testing technique. It was first introduced in the early 1990s to analyze the reliability of UNIX utilities [9]. Since then, it has become one of the most effective and efficient testing techniques to find bugs or crashes in thousands of software. It is also widely used by plenty of companies such as Microsoft [10], Google [11], and Adobe [12] to ensure the quality of their software. File formats and network protocols are the most common targets for fuzzing, however, any inputs can be fuzzed, even data in a database or shared memory.

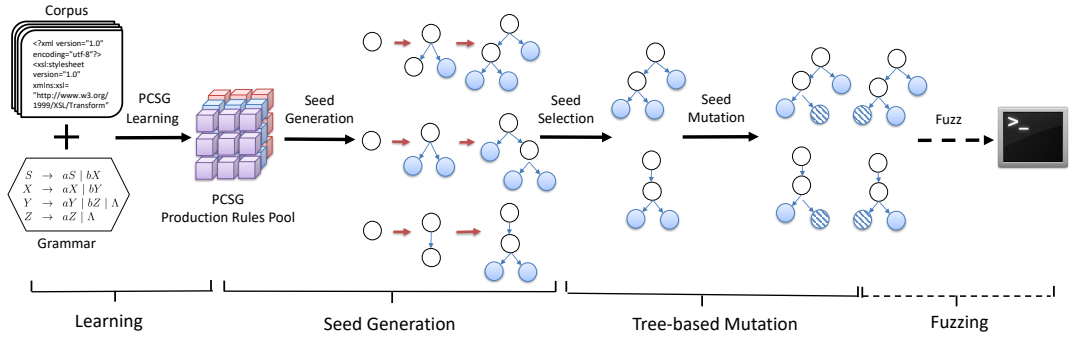


Figure 1.2: The Overview of Vulnerability Detection

3. Data-Driven Seed Generation for Fuzzing

The guided mutation-based approaches can efficiently fuzz programs that process compact and unstructured input formats (e.g., images and videos). However, they are less suitable for programs that take highly-structured inputs (e.g., XSL and JavaScript). Such programs often process the inputs in stages, as shown in Figure 1.3, i.e., *syntax parsing*, *semantic checking*, and *application execution*. Most invalid inputs generated by mutation-based approach fail to pass the syntax parsing and thus rejected at an early stage of processing, which makes the fuzzing spend a large amount of time struggling with syntax correctness and heavily prevents them from finding significant bugs.

In the third work, we propose a novel data-driven seed generation approach, named Skyfire[1]. It leverages a large number of samples (i.e., corpus) to automatically extract the knowledge of semantic rules, and utilizes such knowledge to generate well-distributed semantic-valid seeds for fuzzing programs that process highly-structured inputs. In this

[1] An Autobot scientist in the film *Transformers*.

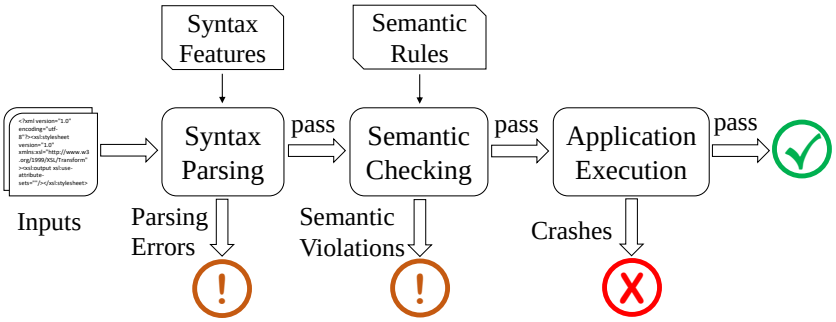


Figure 1.3: Stages of Processing Highly-Structured Inputs

sense, Skyfire is orthogonal to mutation-based fuzzing approaches by providing high-quality seeds for them and improving their efficiency and effectiveness. Besides, Skyfire advances the existing generation-based fuzzing approaches by carrying the fuzzing progress beyond the semantic checking stage to reach the execution stage to find significant bugs without any manual specification tasks.

Skyfire makes the following contributions:

- We propose to learn a PCSG from existing corpus to express semantic information for highly-structured inputs.
- We propose to leverage PCSG to generate seeds with diverse syntax structures, apply seed selection to reduce seed redundancy, and perform grammar-aware mutation to introduce minor changes diversely, which aims to generate correct, diverse, and uncommon seeds for fuzzing programs that process highly-structured inputs.
- We evaluated Skyfire by using the seeds generated by Skyfire to test several XSLT, XML engines. Skyfire greatly improved the code coverage and discovered 16 new vulnerabilities.

4. Grammar-Aware Greybox Fuzzing

Although we generate well-distributed seeds for fuzzing, the fuzzing procedure still suffers from the grammar-unaware problem. Grammar-blind mutations, such as bit flip, byte flip and so on, soon destroy the structure of seeds. Then fuzzers will struggle to

produce a meaningful test case that triggers a new path. Since seed generation can be grammar-aware, the mutation can also be guided by the grammar.

Fuzzers that have no knowledge of the structure of inputs suffer from the following problems:

- The trimming strategies adopted in traditional greybox fuzzing are grammar-blind, and hence can easily violate the grammar or destroy the input structure.
- The mutation strategies of traditional greybox fuzzing are grammar-blind, and hence most of the mutated test inputs fail to pass the syntax parsing and are rejected at an early stage of processing.
- Programs that process structured inputs often have intrinsic indeterminate behaviors which lead to the low stability of traditional greybox fuzzing. Such low stability makes it difficult for traditional greybox fuzzing to decide whether new coverage is achieved.

To address the above challenges, we propose a new grammar-aware coverage-based greybox fuzzing approach. Our approach takes as inputs a *target program* and a *grammar* of the test inputs. Based on the grammar, we parse each test input into an abstract syntax tree (AST). Using ASTs, we introduce a grammar-aware trimming strategy that can effectively trim test inputs while keeping the input structure valid. Besides, we propose two grammar-aware mutation strategies that can quickly carry the fuzzing exploration beyond syntax parsing. In particular, we first enhance AFL's dictionary-based mutation strategy by inserting and overwriting tokens in a grammar-aware manner and then propose a tree-based mutation strategy that replaces one subtree in the AST of a test input with the subtree from the test input itself or another test input in the queue. Furthermore, we leverage block coverage to mitigate the stability problem caused by multi-threads, and we also propose a selective instrumentation strategy to preclude the source code which is responsible for indeterminate behaviors for enhancing the fuzzing stability.

In summary, this work makes the following main contributions.

- We propose a novel grammar-aware coverage-based greybox fuzzing approach for programs that process structured inputs.
- We implement our approach and made it open-source (https://github.com/zhunki/Superion) and will improve it on follow-up feedback. We conducted experiments using several XML and JavaScript engines to demonstrate the effectiveness of our approach.
- We found 34 new bugs, among which we discovered 22 new vulnerabilities with CVEs assigned and received 3.2K USD bug bounty rewards.

1.3 Thesis Outline

Figure 1.1 and Figure 1.2 show the roadmap for this dissertation, which describes the sequence of our research work and thereby reveals the train of our thoughts on web-based security. The remaining of the thesis is organized as follows:

In **Chapter 2**, we present the background and preliminaries used in this thesis. We will introduce JavaScript malware types, obfuscation of JavaScript malware, some preliminaries about fuzzing and a representative of coverage-based greybox fuzzing.

In **Chapter 3**, we address the first goal of the thesis, i.e., being able to detect JavaScript malware efficiently. This is achieved using machine learning techniques. our approach is to benefit from both static analysis and dynamic analysis with their respective advantages (static analysis is more scalable while dynamic analysis is more accurate). First, machine learning techniques are used statically to classify the JavaScript programs, using notably function call patterns and timed of external references; second, dynamic analysis is used to improve the accuracy of the classification. Experiments are performed on a controlled set (20,942 labeled programs) and an uncontrolled set (1,400,000 wildly crawled JavaScript fragments). The method proposed in this chapter successfully detects 99.68% of malware, while the second best method (the Avast! antivirus) detects only 93.84%. The worst commercial antivirus detects even as low as 5% of malware.

Then **Chapter 4** is concerned with JavaScript malware behavior modeling and addresses the second goal of the thesis. To prevent future attacks, a model of a malware is sought, and the method proposed is to build a DFA modeling the attack behavior using system calls. This is obtained using a learning algorithm inspired by Angluin's L^* algorithm. First, malware samples are executed multiple times, with multiple execution traces collected. Then, irrelevant actions are filtered out. Then, the learning phase can be done: as in L^* , the teacher answers membership and candidate queries. Here, membership denotes the possibility for a trace to actually represent an attack, and it is implemented based on defense rules. Candidate query denotes the check whether the candidate DFA is indeed (equivalent to) the one accurately modeling the attack. Here, the techniques used are regression testing and random sampling testing. Experiments are applied on a set of 276 malware samples, as well as 10,000 benign samples. Again, experiments are particularly impressive, with a relatively fast learning, and a high accuracy: 95.51% for the proposed approach against 81.41% for the best existing approach.

Chapter 5 and **6** are two works on vulnerabilities detection. Among them, **Chapter 5** is concerned with vulnerability detection via data-driven seed generation, and addresses the third goal of the thesis. This chapter is relatively orthogonal to the first two contributions, as it deals with fuzzing, i.e., an efficient automated testing used to exhibit bugs or vulnerabilities in programs. The idea is to propose well-distributed seed inputs for fuzzing programs. The approach is orthogonal to mutation-based approaches (that start from a program, and apply modifications such as bit-flipping) Starting from a corpus(a set of sample programs) and a grammar, the method learns a probabilistic context-sensitive grammar (PCSG). From the PCSG, a seed (i.e., a concrete program) is generated, by applying left-most derivation from the grammar. Various heuristics are used. Notably, low-probability production rules are favored so as to obtain programs less likely to be generated, and therefore more likely to contain undiscovered bugs. This choice is itself probabilistic: most of the time (90%), low-probability production rules are chosen, but sometimes (10%) high-probability production rules are favored too. The framework is implemented in an open-source Java tool, and has been applied to XSL and XML bug detections. Several unsuspected bugs were discovered in existing open-source projects and libraries.

Chapter 6 is again concerned with the third goal of the thesis. It improves the contribution of Chapter 5 by addressing fuzzing for high-structured inputs for which the previous approaches do not succeed. The issue in previous approaches is that generated seeds very often do not pass the syntactic or semantic checks if the language is too elaborated, i.e., too constrained by its grammar. The main idea is to propose mutations at the abstract syntax tree level instead of at the concrete level. Therefore, the seeds remain grammar-aware, and pass the checks. Various heuristics are proposed so as to keep the subtree at a reasonable size. The implementation has bee applied to XML and JavaScript programs. It shows a significant improvement in both line coverage and function coverage when compared to state-of-the-art techniques (AFL and jsfunfuzz). However, the time necessary is here different from the previous approaches, and requires months of computation. Again, unsuspected new vulnerabilities have been discovered using this technique.

At last, **Chapter 7** concludes the thesis and proposes future general perspectives to improve the four pieces of work.

1.4 Publication List

All publications of the Ph.D. candidate are listed as follows,

- **Junjie Wang**, Yinxing Xue, Yang Liu, and Tianhuat Tan. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification, In Proceedings of the 10th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2015), 14-17 April 2015, Singapore.

- Yinxing Xue, **Junjie Wang**, Yang Liu, Hao Xiao, Jun Sun and Mahinthan Chandramohan, JS*: Detection and Classification of Malicious JavaScript via Attack Behavior Modelling, In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 12-17 July 2015, Baltimore, Maryland.

- **Junjie Wang**, Bihuan Chen, Lei Wei, and Yang Liu, Skyfire: Data-Driven Seed Generation for Fuzzing, In Proceedings of 38th IEEE Symposium on Security and Privacy (S&P 2017), May 22-24 2017, San Jose, CA.
- **Junjie Wang**, Bihuan Chen, Lei Wei, and Yang Liu, Superion: Grammar-Aware Greybox Fuzzing, In Proceedings of 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019), May 25-31 2019, QC Canada.

2

Background and Preliminaries

In this chapter, we present the background and preliminaries involved in the thesis. We first introduce JavaScript malware types, which is the base of our JavaScript malware detection approach. After that, we introduce one challenge of JavaScript malware detection, obfuscation.

We also introduce some preliminaries of fuzzing in this chapter. Finally, we introduce AFL, as a representative of coverage-based greybox fuzzing tools. These are vital to understanding our vulnerabilities detection work.

2.1 JavaScript Malware Types

According to Kaspersky security bulletin [13], attacks targeting JRE, attacks targeting Adobe PDF Reader, browser, Adobe Flash and so on, account for more than 95% of attacks launched by JavaScript malware. Attack surface could be a good standard for classification since it indicates the root cause of vulnerability and the same attack type shares a similar behavior pattern. We list eight most commonly-seen JavaScript attack types according to attack surface as follows:

- *Type I: Attack targeting browser's vulnerabilities.* The browser is made up of dozens of components and for speed considering, they are normally implemented using C/C++. There are hundreds of vulnerabilities reported in the source code of web browsers. This type of attack targets vulnerabilities (e.g., CVE-2014-1567) of a browser of a certain version or with certain plugins, and then exploits memory corruption vulnerabilities to get control of execution flow. Consequently, it will lead to the arbitrary code execution.

- *Type II: Browser hijacking attack.* Browser hijacking is a type of attack that modifies a web browser's settings without the user's permission. A browser hijacker may stealthily replace the original home page, error page, or search engine page with its own. This is generally used to force hits to a particular website, increasing its advertising revenue. The home pages that are set by hijackers are often search engine pages, and many of these programs are spyware programs that track personal data. For instance, CVE-2007-2378 in the Google Web Toolkit (GWT) framework is a JavaScript hijacking attack, which illegally tracks personal data.

- *Type III: Attack targeting Adobe Flash.* ActionScript is a language very similar to JavaScript and used in Adobe Flash and Adobe Air. It also allows memory allocation in the heap. A malicious input exploits vulnerabilities like CVE-2012-1535 to manipulate the heap layout, finally leading to code execution.

- *Type IV: Attack targeting JRE.* Attacks take advantage of vulnerabilities in Oracle's J2SE JDK and JRE to execute arbitrary code on a victim's computer. Users

of Java who is not using the latest patched version of Java may be vulnerable to arbitrary code execution by a remote attacker. The JVM security depends on the bytecode verifier, the class loader, and the security manager. For instance, the vulnerability in Java 7 Update 11 (CVE-2013-1489) allows attackers to bypass all security mechanisms in the Java browser plugin.

- *Type V: Attack based on multimedia.* Attacks are carried by multimedia files or formats supported by browsers, e.g., CSS based attacks. Generally, malicious downloadable resources (e.g., image and font files) can be deliberately crafted to exploit vulnerabilities like CVE-2008-5506 to consume the network capacity of visitors, launch Cross-Site Request Forgery (CSRF) attacks, spy on visitors or run distributed denial-of-service (DDoS) attack.

- *Type VI: Attack targeting PDF reader.* Attackers exploit vulnerabilities in PDF Reader to take control of a victim's computer. For example, vulnerabilities like CVE-2011-2462 could cause a crash and potentially allow an attacker to take control of the affected system.

- *Type VII: Malicious redirecting attack.* Cyber-criminals are constantly thinking up new approaches to redirect unsuspecting visitors to their drive-by landing pages. It includes "Meta Refresh redirecting", "JavaScript Redirect", "CSS redirecting", "OnError redirecting" and so on. This attack type generally belongs to CWE-601.

- *Type VIII: Attack based on Web attack toolkits, e.g.Blacole.* Exploit kits are getting increasingly popular in an unskilled underground market. This is not only because they are easier to use, but also advanced enough to evade detection. Developers of toolkits are selling a product that is fueling the growth of a self-sustaining, profitable, and increasingly organized global underground economy. According to a recent report [14], Symantec reports that 61% of observed Web-based attack activities could be directly attributed to attack kits. The magnitude of these attacks and their widespread usage are concerns for end users as well as the enterprise and everyone in between.

2.2 Obfuscation in JavaScript Malware

Obfuscation is very common in JavaScript language. It changes the original JavaScript application by making semantics-preserving modifications to make the program difficult to understand. Both benign and malicious JavaScript applications make use of obfuscation to achieve different purposes. The benign JavaScript application makes use of obfuscation to protect intellectual property, while the malicious JavaScript uses obfuscation to hide its malicious intent.

The commonly used JavaScript obfuscation techniques can be categorized into the following types [15]: Data obfuscation (a variable or a constant is decomposed into multiple variables or constants, e.g., string splitting); Unicode or hexadecimal encoding (malicious JavaScript code is transformed into unreadable Unicode or hexadecimal encodings); customized encoding functions (a customized encoding and decoding functions are used to obfuscate and de-obfuscate malicious JavaScript code); standard encryption and decryption (using standard cryptography functions to encrypt and decrypt the malicious JavaScript code); logical structure obfuscation (changing the logical structure of JavaScript code by adding redundant instructions that are independent to the core functionality of the malicious code, or modifying the conditional branches).

Obfuscation makes the signature-based anti-virus software incapable. Although the usage of obfuscation is not the dominant factor determining the maliciousness of JavaScript code, malicious code indeed exhibits some abnormal obfuscation features such as multiple times of obfuscation [15] and the distinct encoding scheme from that in benign code.

We randomly picked up 100 (out of 287) malicious samples from our malware collection and 100 benign samples. Then we manually examined the obfuscation techniques adopted by these samples. We found 90% of malicious and 30% of benign JavaScript code use data obfuscation. Most of the state-of-art anti-virus products can resist such obfuscation [15], as the textual and structural information are generally not obscured to an unreadable level.

Encoding obfuscation usually adopts three ways to encode the original code: in ASCII, Unicode or hexadecimal representations. These three types of standard encoding schemes are mainly used in the benign JavaScript code. In our selected samples, we found about 45% of benign scripts and 15% of malicious ones use standard encoding schemes. To evade the decoding, instead of standard encoding schemes, customized encoding functions are often defined in malicious scripts (82% of malicious ones). The standard encoding can be handled, but the customized encoding is hard to address unless dynamic analysis or at least the interpretation (without execution) is used.

The obfuscated code can be increasingly unreadable and complex by applying multiple levels of obfuscation. Thus, to analyze such obfuscated code, multiple times of unpacking is required for de-obfuscate, e.g., the sample in Figure 2.1 uses multiple levels of obfuscation to hide its malicious intent.

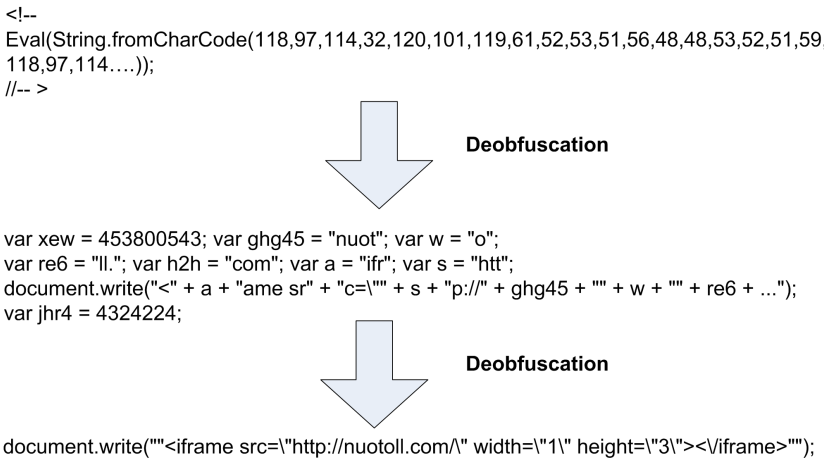


Figure 2.1: Example of Multi-Level Obfuscation

2.3 Preliminaries about Fuzzing

Fuzzing data that crosses the trust boundary is interesting. For example, fuzzing the source code that processes files uploaded by arbitrary users is more important than fuzzing server configuration file. This is because the configuration file can only be modified by the limited users with privilege.

Fuzzing feeds a large number of inputs to the target program in the hope of triggering unintended behaviors, such as crashes or assertion failures. The quality of the inputs is one of the most significant aspects that affect the success of fuzzers [16, 17]. Inputs are usually generated by *mutation-based* and/or *generation-based* approaches. Mutation-based approach generates inputs by mutating existing inputs through bit flipping, byte flipping and so on. Such mutations can be random [9], or guided by coverage [18], taint analysis [19, 20], symbolic execution [21–23] or a combination of taint analysis and symbolic execution [24, 25]. Taint analysis can identify interesting bytes to mutate and symbolic execution relies on constraint solving to systematically explore execution paths.

Fuzzing can also be coverage-guided or totally blackboxed. Coverage-based fuzzing (also known as greybox fuzzing) is one kind of prevalent fuzzing approach which requires no program analysis and widely used in practical security, where a seed input is picked up from the seed sets and slightly mutated into new tests which then be executed. If the execution discloses any new interesting paths, the generated test will be added into seed sets and wait for further mutation, otherwise, it will be discarded. For example, AFL is one of the most well-known greybox fuzzing tools.

On the other hand, generation-based fuzzing constructs inputs from a specification, e.g., input model [26–28] that specifies the format of input data structure and integrity constraints, and context-free grammar [29–32] that describes the syntax information. Naturally, these model-based or grammar-based approaches can generate inputs that can easily pass the integrity checking (e.g., checksum) or syntax checking, quickly advancing the fuzzing progress beyond the syntax parsing stage. In that sense, these approaches can significantly narrow down the search space of fuzzing than those mutation-based approaches.

However, the majority of the inputs generated from the generation-based approach are usually unable to pass the semantic checking. This is a common difficulty to develop a smart fuzzer. For example, an XSLT engine will check if an $attribute$ can be applied on a certain $element$. If this semantic checking is violated, an “unexpected attribute name” message will be prompted, and the program will abort further execution. There

are hundreds of such semantic rules implicitly implemented in a program. As a result, only a small portion of inputs generated from generic generation-based approach can successfully reach the execution stage, where the significant bugs normally hide; and a large part of the source code is untouched. It is needed to notice that generating a small portion of semantic invalid samples can get unexpected results.

To generate semantically valid inputs, some fuzzing work [33–35] proposed to manually specify semantic rules that the inputs generated should follow. However, different programs often implement different sets of semantic rules even for the same input (e.g., for XSLT 1.0 and 2.0); and it is daunting and labor-intensive, or even impossible to manually specify all semantic rules.

One of the most successful mutation-based greybox fuzzing techniques is coverage-based greybox fuzzing, which uses the coverage information of each executed test input to determine the test inputs that should be retained for further incremental fuzzing. AFL [18] is a state-of-the-art coverage-based greybox fuzzer, which has discovered thousands of high-profile vulnerabilities. Thus, without the loss of generality, we consider AFL as the typical implementation of coverage-based greybox fuzzing.

As shown in Figure 2.2, AFL takes the target program as an input and works in two steps: instrumenting the target program and fuzzing the instrumented program. The instrumentation step injects code at branch points to capture branch (edge) coverage together with branch hit counts (which are bucketized to small powers of two). A test input is said to have new coverage if it either hits a new branch or achieves a new hit count for an already-exercised branch. The fuzzing step can be broken down into five sub-steps. Specifically, a test input is first selected from a queue where the initial test inputs, as well as the test inputs that have new coverage, are stored. Then the test input is trimmed to the smallest size that does not change the measured behavior of the program, as the size of test inputs has a dramatic impact on the fuzzing efficiency. The trimmed test input is then mutated to generate new test inputs and the program is executed with respect to each mutated test input. Finally, the queue is updated by adding those mutated test inputs to the queue if they achieve new coverage, while the mutated test inputs that

achieve no new coverage are discarded. This fuzzing loop continues by selecting a new test input from the queue.

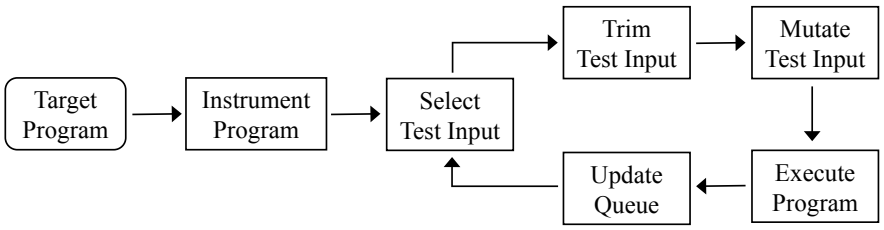


Figure 2.2: The General Workflow of AFL

3

JavaScript Malware Detection Using Machine Learning

In this chapter, we will introduce our JavaScript malware detection approach which uses machine learning. This is the first part of JavaScript malware detection work and Chapter 4 introduces the second part of the work.

3.1 Introduction

Towards an insightful analysis of JavaScript malware evolution trend, it is desirable to detect JavaScript malware and further classify them according to the exploited *attack*

vector and the corresponding *attack behaviors*. Considering the emergence of numerous new malware or their variants, knowing the attack type and the attack vector will greatly help domain experts for further analysis. Such an automated classification can significantly speed up the overall response to the malware and even shorten the time to discover the zero-day attacks and vulnerabilities.

In this work, we propose an automatic approach to perform JavaScript malware detection and classification by combining static analysis (via machine learning) and dynamic analysis to achieve both scalability and accuracy. To efficiently detect and classify JavaScript malware, we propose a two-phase classification. The first phase is to classify malicious JavaScript samples from benign ones, while the second phase is to discover the attack type that a JavaScript malware belongs to. In the first phase, we extract and analyze features that are generally predictive for malicious scripts. To address the obfuscation and dynamic code generation problems, we extend HtmlUnit [36] to obtain the final unpacked code. We extract features from document elements, source code, and sensitive function call patterns.

In the second phase, we focus on the features that are representative and unique for each attack type. To identify unique features relevant to a certain attack type, the inter-script analysis is adopted to find API usage patterns, which can serve as features of different attack types. For the suspicious candidates that fall into the grey zone (uncertain in classification), we adopt the dynamic analysis to unveil the practical behaviors of the attack and match it to existing attack behavior models. For dynamic analysis, we instrument Firefox to capture the attack behaviors of JavaScript malware. Holistically, static analysis assures the scalability while dynamic analysis of uncertain scripts improves the accuracy.

To sum up, this work makes the following contributions:

- We propose a combined approach for JavaScript malware classification by firstly using the machine learning approach with predictive features like function call patterns and the times of external references, and then further improving the accuracy using dynamic program analysis. Our approach achieves the scalability and accuracy.

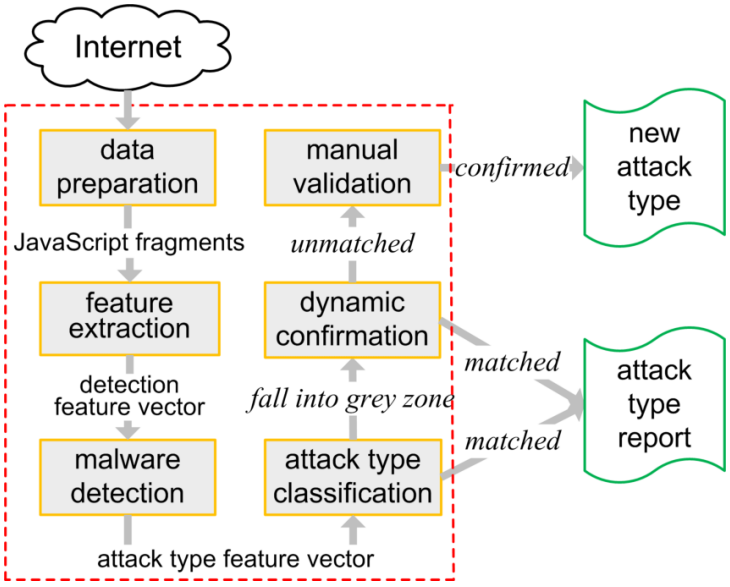


Figure 3.1: The Work Flow of JSDC

- We train the classifiers with the JavaScript malware collection from famous malware repository VXHeaven [37] and OpenMalware [38]. We use publicly available JavaScript malware collections as the oracle to observe the features and to train the classifiers. We also conduct experiments to compare our approach with the mainstream anti-virus products and research prototype JSAND. The evaluation shows our method is far ahead of many anti-virus tools.

- We combine static and dynamic analysis to have a complete detection loop for identifying attack types, even new malware variant.

3.2 System Overview of JSDC

Our general workflow is as shown in Figure 3.1. The workflow includes two classification steps. The first classification detects malicious JavaScript fragments and the second one label detected JavaScript fragments into eight attack type. If there is any malicious JavaScript fragments cannot be classified into any of eight attack types, we apply dynamic execution on it to test whether they are new attacks. To do the classification, we also need to perform dataset preparation, features extraction steps and so on. In the following subsections, more details about each step will be given.

3.2.1 Data Preparation and Preprocessing

Heritrix [39] is an open-source, extensible, scalable and archival-quality Web crawler. We leverage it to crawl JavaScript samples as the training data set and detection source. The malicious JavaScript fragments training data is crawled from the malicious JavaScript domain list provided by Web Inspector [40]. Some JavaScript fragments are in form of separate JavaScript files, while some are embedded in the HTML files. There are also JavaScript files link to the external source. We carefully setup Heritrix to download those JavaScript fragments as much as possible.

Most of the downloaded JavaScript fragments are obfuscated, packed or compressed. This poses difficulty in our feature extraction step. Here we turn to HtmlUnit [36] for help. HtmlUnit is a lightweight GUI-Less emulator for JavaScript.

3.2.2 Feature Extraction

Our training dataset includes samples labeled malicious and benign. We extract features from those two datasets. Firstly, textual features of the JavaScript code fragments based on textual analysis, word size, an n-gram model, the frequency of characters, commenting style, and entropy are extracted. No matter if the code is obfuscated or not, such textual features are extracted to unveil the distinct textual pattern due to the obfuscation of malicious code.

To address the issue of obfuscation, Zozzle [4] relies on the Detours binary instrumentation library, which allows intercepting calls to the function COleScript::Compile in the JavaScript engine located in the jscript:dll library. This process is called unpacking, which can help to handle the simple obfuscation such as data or standard encoding.

De-obfuscation exposes the original code, but the original code does not exhibit the exact behavior of the code executed at runtime due to JavaScript dynamic features. Dynamic execution trace can capture the actual behavior of the code, but collecting dynamic execution traces to get the complete behavior model can be computationally costly. To overcome the dynamics, Revolver [41] adopts a browser emulator based on

HtmlUnit to virtually executing the code. In such way, the actual and accurate AST of dynamically-generated code is obtained. Thus, a dynamic approach based on (virtual) execution can ideally address issues of obfuscation and dynamics of JavaScript code. The only drawback is that dynamic approaches might be time-consuming and not efficient for large-scale malware detection.

In this study, to conduct a large-scale detection, our focus is to explore effective features of the obfuscated code as well as those of the original code. Our approach also takes features of dynamically generated code into account. To mitigate the performance problem, we extend HtmlUnit to only obtain the dynamically generated code, but the actual rendering of the JavaScript and execution of the redirected pages are omitted.

We extract features about the program information from the dynamically generated code. The program information includes HTML properties and Document Object Model (DOM) operations, and the API usage patterns in the JavaScript snippet. The API usage pattern refers to the pattern of function calls in the analyzed script. The inner-script API usage pattern analysis can show the semantics of JavaScript application to some extent [42].

Then, we also extract AST-relevant features, e.g., the tree depth, the pair of type and text from some AST nodes [4]. Features from AST, such as the tree breadth, the biggest difference among the subtrees' depth, are used in [43]. The rationale is that the injected code may take advantage of dynamic code generation provided by functions like `eval`. The malicious code may adopt several levels of `eval` calls to inject the final attack, which could be easily manifested from the AST-based features.

3.2.3 Normalizing Features Vector

Directly using the raw or unnormalized feature values would not perform well due to the fact that the original features have a different range of values. For classifiers like SVM that consider the distance among feature vectors in vector space, the larger range of a feature gives more weight, compared with those with smaller values. Thus, to treat features fairly, we normalize the feature values via range and z-normalization. For

each feature value x of a feature f, we define a value normalization function $r(x) = \frac{x-min}{max-min}$, where min and max denote the minimum and maximum value of f. To further standardize the value, we calculate $z\text{-}score$ by $z(x) = \frac{x-\mu}{\sigma}$, where μ and σ are the mean and standard deviation for f.

3.2.4 Feature Selection

Before we actually train the classifiers, we need one more operation to select the features. There are too many features, which will cause overfit. In statistics, overfit is the production of an analysis that corresponds too closely or exactly to a particular set of data, and may fail to fit additional data or predict future observations reliably. To prevent overfitting, we pre-select features.

In the first classification phase (malware detection phase), the problem is essentially a binary classification problem. Thus, we try to keep the number of features, which distinguish malicious scripts from benign ones, as large as possible. The dominant features (that are high-correlated with the malicious attack at a probability > 0.8) or non-dominant features are both considered for malware detection.

The second step is to classify detected malicious JavaScript fragments into eight attack types. Part of the features used in the first step can be re-used in this classification. Except for those features, we need to propose new features to classify attack types. This is because the features used in the first step is common to malicious behaviors. To distinguish attacks type, we need some features more unique to each attach type. We mostly relied on function call patterns in this step.

In the second classification phase (attack type classification phase), the problem becomes a multinominal classification problem, for which finding high-correlated feature(s) for each class is critical. To eliminate the noise of attack-independent characteristics (like those due to obfuscation), we need to filter out features that are common to benign scripts and also to all the malicious scripts. In other words, we will try to select those features that are predictive and unique to different attack types. Given an attack

type t and a feature s, to check whether s is high-correlated with t, we use χ^2 test (for one degree of freedom), which is described below:

$$A\text{: attack belonging to } t \text{ with } s$$
$$B\text{: attack belonging other types with } s$$
$$C\text{: attack belonging to } t \text{ without } s$$
$$D\text{: attack belonging other types without } s$$
$$\chi^2 = \frac{(A+B+C+D)(A*D-C*B)^2}{(A+C)*(B+D)*(A+B)*(C+D)}$$

After defining an initial feature set, we do a filtering process to select those features that are more predictive. We select features with $\chi^2 \geq 2.71$, which represents a 90.0% confidence that the two values (presence of s and belonging to t) are correlated.

3.2.5 Classifiers Training

Before the actual application of our approach in detection, we need to train the classifiers for prediction. We used several publicly available JavaScript malware collections as the oracle for training, namely VXHeaven [37], OpenMalware [38] and Web Inspector [40]. We verified our classifiers using k-fold Cross Validation (CV).

3.2.6 Dynamic Confirmation

For the suspicious samples that fall into none of eight attack types, we use dynamic execution to confirm its attack behaviors. We learn an attack model for each attack type and check its execution trace with a model of eight attack types. If any model can accept the execution trace, the sample is confirmed belonging to this attack type. If still there is not any model can accept the sample, we manually check whether it is a new attack type or a false positive.

3.3 Details of Feature Extraction

In this section, we elaborate on the features that are used to do malicious JavaScript fragment detection and classification. These features can be categorized into three types: textual features, inner-script features and inter-script features as shown in Figure 3.2. Textual features are detailed in Section 3.3.1. Inner-script features are extracted in Section 3.3.2, while inter-script features are discussed in Section 3.3.3.

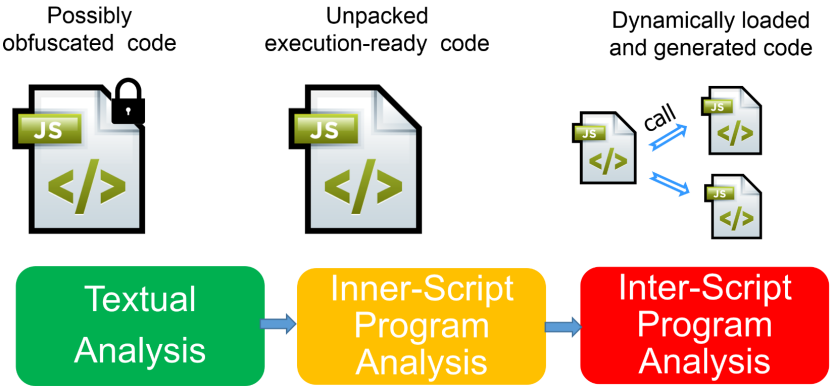


Figure 3.2: Feature Extraction Process

We take the JavaScript code fragment in Figure 3.3 as the running example to elaborate our approach. In the following part, the feature only used in JavaScript malware detection is annotated with †, and the features used in both malicious JavaScript detection and attack type classification are annotated with *.

3.3.1 Textual Analysis

We invested a considerable effort to examine malware by hand and categorize them. Most of JavaScript malware are obfuscated prior to deployment, either by hand or using off-the-shelf obfuscation toolkits [44].

We first train our cluster based on collected samples after manual categorizing. We concern the following features of each category. Most shellcode and nop sleds are written as hexadecimal literals using the '\x' prefix. Some other malware uses the '\u' Unicode encoding which converted to binary data using JavaScript unescape function. We

```
<!--
eval(String.fromCharCode(118,97,114,32,120,101,119,61,52,53,51,
56,48,48,53,52,51,59,118,97,114,32,103,104,103,52,53,61,34,110,
117,111,116,34,59,118,97,114,32,119,61,34,111,34,59,118,97,114,
32,114,101,54,61,34,108,108,46,34,59,118,97,114,32,104,50,104,
61,34,99,111,109,34,59,118,97,114,32,97,61,34,105,102,114,34,59,
118,97,114,32,115,61,34,104,116,116,34,59,100,111,99,117,109,
101,110,116,46,119,114,105,116,101,40,39,60,39,43,97,43,39,97,
109,101,32,115,114,39,43,39,99,61,34,39,43,115,43,39,112,58,47,
47,39,43,103,104,103,52,53,43,39,39,43,119,43,39,39,43,114,101,
54,43,39,39,43,104,50,104,43,39,47,39,43,39,34,32,119,105,100,
39,43,39,116,104,61,34,49,34,32,104,39,43,39,101,105,103,104,
116,61,34,51,34,62,60,47,105,102,39,43,39,114,97,109,101,62,39,
41,59,32,118,97,114,32,106,104,114,52,61,52,51,50,52,50,50,52));
//-->
```

(a) The original obfuscated version

```
var xew = 453800543;
var ghg45 = "nuot";
var w = "o";
var re6 = "ll.";
var h2h = "com";
var a = "ifr";
var s = "htt";
document.write("<"+a+"ame sr"+"c=\""+s+"p://"+ghg45+""+w+""+re6
+""+h2h+"/"+"\"wid"+"th=\"1\"h"+"eight=\"3\"></if"+"rame>");
var jhr4 = 4324224;
```

(b) The HtmlUnit unpacked version

Figure 3.3: The Running Example of Feature Extraction

seek features of using hexadecimal literals or Unicode encoding. While some literature [45, 46] conflates obfuscation with maliciousness, literature [47] reveals that the correlation between obfuscation and maliciousness is weak. We cannot assume that detecting obfuscation implies maliciousness.

iFrame. An inline frame (iFrame) is used to embed another source within the current web page. It can be used for several powerful and good intent, but attackers also make use of it as a way to embed malicious JavaScript fragments into otherwise healthy web sites. Looking back to 2012, in many scenarios, we could find the same injection to thousands of innocent sites. For example, an iFrame link to ttl888.info was injected into more than 15,000 benign sites. Malicious iFrames often are invisible. We concern their attributes, including frameborder, border, width, height, display: none and visibility: hidden. From an iFrame element, we extract seven features as listed in Figure 3.4.

JavaScript injections. Malicious injections are similar to iFrame attack, but it rather than loading another site's HTML documents, it just loads the JavaScript files. It can do similar damage as an iFrame injection (redirecting the browser to exploit kits, to SPAM, to fake AV, and other similar nefarious attacks). Almost 10,000 sites were doing

Features of iframe:	**Features of JavaScript injections:**
$if_{f1} : \langle frameborder : 0 \rangle$	$js_{f1} : \langle word_size > 300 \rangle$
$if_{f2} : \langle border : 0 \rangle$	$js_{f2} : \langle entropy < 3.0 \rangle$
$if_{f3} : \langle width < 10 \rangle$	$js_{f3} : \langle most\_frequent\_character \rangle$
$if_{f4} : \langle height < 10 \rangle$	$js_{f4} : \langle second\_frequent\_character \rangle$
$if_{f5} : \langle display : none \rangle$	$js_{f5} : \langle third\_frequent\_character \rangle$
$if_{f6} : \langle visibility : hidden \rangle$	$js_{f6} : \langle sensitive\_function\_usage \rangle$
	$js_{f7} : \langle misuse\_of\_annotation \rangle$
	$js_{f8} : \langle hexadecimal\_literals \rangle$
	$js_{f8} : \langle Byte\_occurrence\_frequency \rangle$
	$js_{f8} : \langle Duplicated\_code \rangle$

Figure 3.4: Features of iFrame and JavaScript Injections

conditional redirections to wayoseswindows.ru at some point in 2012. We also list ten features of JavaScript injections in Figure 3.4.

Longest word size†**.** Obfuscated JavaScript code fragments usually employing super long words, which is the signal the existence of encoding or encryption that is used for the content of the function call like `eval`. A JavaScript code fragment with word size (e.g., larger than 350 words [45]) is highly likely to be obfuscated. For example, as shown in the obfuscated sample in Figure 3.3a, the size of the longest word is 814 bytes. Different from normal JavaScript code, malicious may use very long string, for example, 112,640 bytes in a word. After tokenization, the longest word size is calculated.

Entropy†**.** Entropy is a commonly-used feature of the unpredictability of information, and in this work, it is used to analyze the distribution of different characters. Entropy is calculated as follows:

$$H(X) = -\sum_{i=1}^{N}(\frac{x_i}{T})log_{10}(\frac{x_i}{T}) \begin{cases} X = \{x_i, i = 0, 1, ..., N\} \\ T = \sum_{i=1}^{N} x_i \end{cases} \tag{3.1}$$

where x_i is the number of each character and T is the number of all characters. Here we ignore whitespace character (e.g.character tabulation, line tabulation, next line character) since it is not part of the JavaScript source code. The obfuscated code typically

has lower entropy than normal source code, since it often contains repeated characters. It has experimented that the entropy for obfuscated code is usually lower than 1.2, while the standard code has entropy from 1.2 to 2.1 [45]. The obfuscated code in Figure 3.3a has an entropy of 1.1.

The byte occurrence frequency of particular character[†]**.** To encode the JavaScript source code, some particular character will be repeatedly used. We calculated the occurrence of characters as a feature. For example, comma character in 3.3a repeated 232 times, which account for 25% of the whole samples.

Commenting style[†]**.** JavaScript has two kinds of annotations, one is for single line annotation and the other one for block comments or called multi-line annotation. To evade from manual auditing and some simple automatic detection approaches, the malicious will misuse those two types of annotations, which can mislead auditor that the code fragment is commented out while due to browser's error-tolerance feature, it will get executed anyhow, as shown in Figure 3.3a.

Duplicated code[†]**.** It is common for malware writers, who tend to include multiple exploits within the same page to increase their chance of successful exploitations. The most common way adopted is injecting the same iframe into different locations of the web page. Generally, each of these exploits is packaged in a similar fashion.

3.3.2 Inner-Script Program Analysis

In this section, we extract features from internal JavaScript fragments. The inner script program features are extracted from unpacked code fragments instead of obfuscated ones, as shown in Figure 3.3b. The inner script features consist of function call patterns and AST features.

Function calls with security risks[*]**.** The malicious behavior of JavaScript malware normally includes dynamic code generation, hijacking users to crafted websites, checking user's platform, stealing information from the cookie, faking user's operation, embedding malicious as well as invisible contents in a web page, string operation to evade

Table 3.1: Functions with Security Risks

Function Name	Function Type	Possible Threats
eval() window.setInterval() window.setTimeout()	Dynamic code execution	Dynamic code generation
location.replace() location.assign()	Change current URL	Redirect to malicious URL
getUserAgent() getAppName()	Check browser	Target specific browser
getCookie() setCookie()	Cookie access	Manipulate Cookie
document.addEventListener() element.addEventListener()	Intercepting Events	Block user's operation or emulating
document.write() element.changeAttribute() document.writeln() element.innerHTML() element.insertBefore() element.replaceChild() element.appendChild()	DOM operation	Embed malicious script, invisible java applets, invisible iframe, invisible silverlight, etc.
String.charAt() String.charCodeAt() String.fromCharCode() String.indexOf() String.split()	String operation	Hide intension, by encoding and encryption

detection of anti-virus. The detailed functions we considered are listed in Table 3.1. The first column in the table is the function name we care about, the second column is the description of behaviors of the function call in the first column, and the last column lists the possible threats caused by functions in the first column.

The running example in Figure 3.3b contains function calls to String.fromCharCode() and eval() as in Table 3.1. The value of function call features are the number of times function calls in the code fragments. Therefore, the value of String.fromCharCode() and eval() in Figure 3.3b are both 1.

AST features*. For AST of JavaScript samples, we consider the following features: the depth of the AST; the breadth of the AST; the difference between the deepest depth of its subtrees and the shortest depth of its subtrees.

There is one particular operation we need to notice about AST. One JavaScript fragment can have several versions of AST due to the unpacking. Each iteration of unpacking can generate an AST and each AST has the corresponding features. For example, the depth of AST of the JavaScript code fragment in Figure 3.3b is 21. The width of the AST of Figure 3.3b is 20 and the difference of depth is 5.

Function call patterns[‡]. Besides times of separate function calls, we also consider function call patterns as our features. Some function call itself is not an indicator of malicious behaviors, but the sequence of several function calls can distinguish whether a sample is malicious or not. For example, the function unescape() employs obfuscation to evade simple checking; function eval() helps to dynamically generate malicious code; the function GetCookie() checks for cookie to distinguish the target victims; function dateObject.toGMTString() checks for time used in cookie and function SetCookie() marks the cookie with attacker's information; function document.write() generate dynamic content to current HTML document and function document.createElement() creates a new document element and next function call append the generated element to current HTML document tree; function newActiveXObject() launch a new Active object and next function call createXMLHTTPRequest() establish a remote network connection and download the exploit files into the victim computer.

3.3.3 Inter-Script Program Analysis

Some JavaScript code fragments are self-contained and can be executed as a whole. While there are some other JavaScript code fragments call functions or variables of external JavaScript files. We refer analysis of external JavaScript as the inter-script program analysis.

Miscellaneous and derived features[‡]. We treat the times of external JavaScript reference as a feature. Besides, we also extract the following features. Feature *changeSRC* is the times of redirecting. We count the number of changing *src* attribute of iFrame or navigator.userAgent. The *domAndDynamicUsageCnt* counts the number of function

calls of the seventh row of Table 3.1. Feature *referenceError* counts the number of reference error, for example, the referred external JavaScript file does not exist anymore.

3.4 Dynamic Confirmation

For detected malicious JavaScript fragments, we try to classify them into one of eight attack types. For those who failed to be classified into any of eight attack types, we propose a dynamic confirmation approach to accurately analyze their behaviors. The workflow is as shown in Figure 3.5. We instrument the browser to capture browser-level system call of JavaScript. The instrumented browser hooks can give us an execution trace, which includes benign parts and malicious parts. We preprocess the execution trace to reduce those system calls that are less relevant to security. The common actions left in the execution trace are used to train the corresponding model of eight attack types.

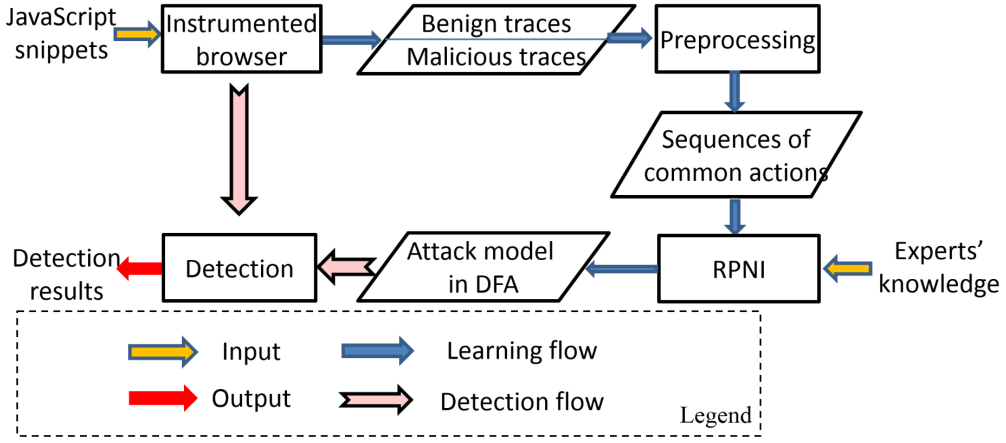


Figure 3.5: The Work-Flow of Dynamic Confirmation

We apply regular positive negative inference (RPNI) [48] to infer the model of each attack type. The model is in the form of DFA and used to check whether a given trace is acceptable to any of the eight models. Some other works [49] use Finite State Automate (FSA) to model an attack. In this work, an FSA is inferred from execution trace of binary executables. The execution trace is intercepted by instrumenting the Mozilla Firefox and hooking the function call to Cross Platform Component Object Model (XPCOM) [50].

3.5 Implementation

This section illustrates our implementation details. We first apply Heritrix [51] to crawl JavaScript samples. Heritrix is fast and easy to configure. For performance reasons, we configure it to download only JavaScript fragments and ignore pictures and multimedia resource. For JavaScript fragments embedded in the web pages, we extract them and write it into a separate file. For external JavaScript files, we parse the link and try our best to download them.

Unpacking JavaScript using HtmlUnit. We use HtmlUnit to unpack the obfuscated JavaScript fragments. HtmlUnit is GUI-less and efficient without rendering. It models HTML document objects and provides API to change the document tree. HtmlUnit has fairly good support for JavaScript. It also can be configured to emulate various browsers, such as Chrome, Firefox or Internet Explorer.

AST Generation and Function Call Pattern Extraction. We use Mozzila Rhino 1.7 [52] to parse JavaScript source code to get AST. Rhino is open-source JavaScript parser written in Java. Similar to browsers, Rhino also has some extent of fault-tolerance.

Classifiers. For classification, we used Weka [53], which provides API for Java. We tried out four different popular classifier algorithms: the "Ranking+Threshold" classifier, Random Forest (RF), J48 and Näive Bayes (NB).

- RT: A classifier that builds a decision tree with K randomly chosen attributes at each node. It supports the estimation of class probabilities based on a hold-out set.
- RF: A non-probabilistic classifier that uses multiple decision trees for training, and predicting the classes by checking the most votes over all decision trees. Random Forest is an extension of bagging for decision trees that can be used for classification or regression.
- J48: A non-probabilistic classifier that uses a single decision tree for classification.

- NB: A probabilistic classifier based on Bayes' theorem with a strong (naive) independence assumptions between the features.

3.6 Evaluation

To evaluate our approach, we designed a controlled experiment and an uncontrolled experiment. The controlled experiment is conducted on 20,942 labeled samples and uncontrolled experiment is conducted on 1,400,000 wildly crawled JavaScript fragments.

3.6.1 Experiment Setup

The controlled experiment samples include 20,000 benign JavaScript fragments, which is downloaded from Alexa top 500 websites. Since these top sites normally belong to big enterprise and have specific administers who are maintaining them, we consider them are benign in large probability. The malicious samples are collected from three sources: 200 from well-known malware repositories VXHEAVEN [37] and another 200 samples from OPENMALWARE [38]. The rest of the 542 samples are from malicious websites reported by WEB INSPECTOR [40], as shown in Table 3.2.

To confirm that the malicious samples are JavaScript malware indeed, we turn to Avast! (version 2014.9.0.2021) and TrendMicro (version 10.4) for confirmation. These two anti-virus products are considered to be accurate. The detection results of these two tools also can give a rough description of malicious samples which give us some hints on labeling them. The benign samples also confirmed by these two anti-virus tools to make sure they are benign indeed.

The unlabelled experiment data includes 1,400,000 JavaScript fragments which are downloaded by Heritrix. The sources are randomly selected by Heritrix. There are normally dozens of links to other URLs in a single web page. When Heritrix encounters such links, it will add it into its download queue for further crawling. Thus, Heritrix can crawl web pages itself.

Table 3.2: Data Sets Used in Controlled Experiments

Benign data set	**#samples**
Alexa-top500 websites	20,000
Malicious data sets	**#samples**
Attack targeting browser vulnerabilities (type I)	150
Browser hijacking attack (type II)	28
Attack targeting Flash (type III)	81
Attack targeting JRE (type IV)	191
Attack based on multimedia (type V)	190
Attack targeting PDF reader (type VI)	101
Malicious redirecting attack (type VII)	92
Attack based on Web attack toolkits (type VIII)	109
Total	942

Our experiment is conducted on a Dell optiplex 990 workstation with Intel i7-2600 3.40GHz core installed. There are 8 GiB memory and running a Windows 7 system.

3.6.2 Evaluation of Malware Detection

3.6.2.1 Controlled Experiments

In the controlled experiment, the malicious and benign samples are randomly mixed together. We adopt ten-fold cross-validation instead of percentage partition, for example, 70% percent for training and 30% percent for prediction. We trained four different classifiers.

Table 3.3 shows the accuracy achieved by different classifiers. In the first column, we list the names of four classifiers. In the second column, we list the accuracy of four classifiers. We can see that the RandomForest classifier achieved the best accuracy among four classifiers. J48 follows next. The third column is the false positive rates of four classifiers. We can see that RandomForest still get the lowest false positive rate and the lowest false negative rate. Therefore, RandomForest outperforms other three classifiers and we use it in the next stage classification.

Table 3.3: The Accuracies of Different Classifiers

ML classifier	Accuracy	FP rate	FN rate
RandomForest (RF)	99.9522%	0.2123%	0.8492%
J48	99.8615%	0.7431%	2.335%
Näive Bayes (NB)	98.2237%	1.127%	4.5280%
RandomTree (RT)	99.8758%	0.3609%	1.4862%

The classifier that performs the worst is the Näive Bayes, with worst accuracy, false positive rate, and false negative rate. We checked the samples missed by all of the four classifiers, as shown in Figure 3.6. The sample is malicious but missed by all classifiers. This sample is a redirecting attack sample belongs to attack type VII, and its target is a mobile device. It first checks the platform, if the system is not the mobile device, it does nothing. Since our analysis tool is based on HtmlUnit, the case cannot be successfully triggered.

```
<!--
if(navigator.userAgent.match(/(android|midp|j2me|symbian|
series60|symbos|windowsmobile|windowsce|ppc|smartphone|
blackberry|mtk|bada|windows phone)/i)!==null){
 window.location = "http://mirolend.h19.ru/away.php?sid=5";
} //-->
```

Figure 3.6: An Exemplar False Negative Case

Comparison with other tools.

We compared our tool with a similar research tool, JSAND [54], and 11 well-known anti-virus products. These 11 anti-virus products include: CLAMAV, AVAST!, AVG, F-SECURE, BITDEFENDER, KASPERSKY, GDATA, MCAFEE, PANDA, SYMANTEC and TRENDMICRO. These tools all are available in VirusTotal [55].

The detection results are listed in Table 3.4. We listed the name of the detection approach and the detection ratio. We can see that JSDC outperforms all other detection approaches, reaches the best detection ratio of 99.68%, that is 939 out of 942. The second best detection tool is AVAST! and its detection ratio is 93.84%. The worst detection tool is PANDA, which only detects 5.31% of malicious JavaScript code fragments.

Table 3.4: Detection Ratio of JSDC and Other Tools on Labelled Malicious Data Sets

Tool	Detection %	Tool	Detection %
JSDC	99.68%	MCAFEE	59.87%
AVAST!	93.84%	SYMANTEC	27.28%
KASPERSKY	86.31%	JSAND	25.58%
GDATA	85.88%	TREND	22.08%
BITDEFENDER	83.23%	CLAMAV	9.24%
F-SECURE	82.38%	PANDA	5.31%
AVG	76.22%		

3.6.2.2 Uncontrolled Experiments

We would like to try out our approach to the samples crawled in the wild. In total, we collected 1,400,000 JavaScript fragments and from which, the best classifier, RF, classify 1,530 samples as malicious. We manually verified 100 of these 1,530 malicious samples, which is randomly selected. We only find 1 false positive case and we feed these 100 samples to 11 detection approaches in Section 3.6.2.1, 11 of them are missed by all the anti-virus products.

We listed the detection result of 12 detection tools in Table 3.5. We can see that KASPERSKY outperforms all other tools and achieved a detection ratio of 48.49%. PANDA still is the worst detection tool and it finds none of them as malicious.

Table 3.5: Detection Ratio of Other Tools on the 99 Unique Samples Reported

Tool	Detection%	Tool	Detection%
KASPERSKY	48.49%	MCAFEE	22.22%
AVAST!	46.47%	JSAND	10.10%
GDATA	34.34%	TREND	6.06%
BITDEFENDER	30.30%	SYMANTEC	3.03%
F-SECURE	30.30%	CLAMAV	2.02%
AVG	27.27%	PANDA	0.00%

We find an exploit to CVE-2014-1580 vulnerability which is missed by all detection tools other than ours. This is a vulnerability of Mozilla Firefox before version 33.0, which improperly initialize memory for a gif image, which allows an attacker to take control of victim machine using a crafted web page.

3.6.2.3 Performance

If a detection approach is fast enough, it can serve as an online scanner to the browser; while it will slow the browser down, it can only work as an off-line detection service. In this section, we want to examine the potential of our approach as an online tool. Therefore, we listed the performance index in Table 3.6.

Table 3.6: The Running Time for Different Classifiers

Operation	**Num**	**Time(s)**	**Avg(ms)**
Feature extraction	20942	1660.7	79.3
Training(RandomForest)	20942	0.785	0.037
Training(J48)	20942	0.364	0.017
Training(Näive Bayes)	20942	0.124	0.006
Training(RandomTree)	20942	0.275	0.013
Detection(RandomForest)	1,400,000	57.4	0.041
Detection(J48)	1,400,000	26.6	0.019
Detection(Näive Bayes)	1,400,000	8.4	0.006
Detection(RandomTree)	1,400,000	19.6	0.014

The first column lists the different operations of our approach. It includes feature exaction, which is executed once per sample. The second operation is the training of classifiers. This operation only needs to be done one time, later we can reuse the model as many times as possible. The third operation is the prediction, or called detection, which also needs to be executed once per sample.

We can see the average feature extraction time is 79.3ms per sample and the training time for the model is about 0.785s for the RandomForest classifier. The detection time

of one sample is 0.041ms on average. The total time for samples is 79.341ms per sample, which is very fast. That means our approach has the potential to serve as an online scanning tool.

3.6.3 Attack Type Classification

We have proved the detection of malicious JavaScript samples is effective and accurate. In addition, we want to know the effectiveness of our approach to classifying the attack types. The accuracy of these four classifiers is 92.14% (RF), 90.22% (J48), 83.44% (NB) and 90.13% (RT), respectively. We listed the classification confusion matrix results in Table 3.7. The samples in the main diagonal are the samples which are correctly classified. Other samples are wrongly classified. There are 74 samples which are wrongly classified as other attack types.

Table 3.7: The Confusion Matrix of RandomForest on Labelled Malicious Samples

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**<–classified as**
139	0	0	0	9	2	0	0	a = type I
0	23	4	0	0	0	0	1	b = type II
1	1	74	1	0	0	1	3	c = type III
0	0	2	179	9	0	1	0	d = type IV
1	0	0	0	179	10	0	0	e = type V
0	0	0	0	19	82	0	0	f = type VI
0	0	0	1	1	0	87	3	g = type VII
0	0	0	1	0	0	3	105	h = type VIII

The most significant wrong classification is 19 samples of attack Type VI are classified as the attack Type V. We found that these two attack types share very similar behaviors and the classifier has the difficulty to tell them apart. We may need more samples to train these two classifiers.

3.6.3.1 Prediction Results

We used our best classifier, RandomForest to classify 1,530 malicious JavaScript samples in the wild. The result is listed in Table 3.8.

Table 3.8: Attack Classification on the 1,530 Malicious Samples

Type	**type I**	**type II**	**type III**	**type IV**
Num	113 (7.39%)	10 (0.65%)	75 (4.90%)	253 (16.54%)
Type	**type V**	**type VI**	**type VII**	**type VIII**
Num	202 (13.20%)	101 (6.60%)	350 (22.88%)	426 (27.84%)

We can see the malicious JavaScript samples detected that belong to attack Type VIII is the most. To verify the result, we manually selected 10% of the samples, which evenly distributed in eight attack types. Among the 164 samples we investigated, 87.8% of them are correctly classified, while others are wrongly classified into other attack types. We found four of them fall into the grey zone and the classification confidence is low.

3.6.4 Combining Dynamic Confirmation and Machine Learning

For those fall into the grey zone, we dynamically execute them and investigate their behaviors. In Table 3.9, we list the certainty of the classification of attack types. For example, 764 of 1,530 samples are classified with certainty = 1. When we set the certainty level as 0.7, 234 samples fall into the grey zone. For these 234 uncertain samples, we apply the dynamic confirmation and executed them 10 times per sample. Then we feed these 10 execution traces to learned models. If any of the 10 traces is accepted by any model, we consider this sample belongs to the corresponding attack type.

Combining the classification results of machine learning and dynamic confirmation, the accuracy of our approach reaches 95%.

Table 3.9: The Certainty Value and the Number of Samples that Fall into Grey Zone

certainty	certain#	total	uncertain#	uncertain %
1	764	1,530	766	51.31%
$\geqslant$0.9	854	1,530	676	50.07%
$\geqslant$0.8	994	1,530	536	44.18%
$\geqslant$0.7	1,296	1,530	234	15.29%
$\geqslant$0.6	1,311	1,530	219	14.31%

Performance of Attack Type Classification. We list the performance of attack classification in Table 3.10. The average training time for each classifier is about 0.11 ms per sample and the detection speed is about 0.0033 ms per sample for best classifier RandomForest. The classifiers are very efficient.

Table 3.10: The Running Time of Different Classifiers in Type Classification

Operation	Num	Time(s)	Avg(ms)
Training(RandomForest)	942	0.1	0.11
Training(J48)	942	0.12	0.13
Training(Näive Bayes)	942	0.04	0.04
Training(RandomTree)	942	0.01	0.01
Detection(RandomForest)	1,530	0.05	0.0033
Detection(J48)	1,530	0.01	0.0006
Detection(Näive Bayes)	1,530	0.001	0.00006
Detection(RandomTree)	1,530	0.02	0.001

3.7 Related Work

Since the past few years, JavaScript malware detection has intrigued the interest of security researchers and many approaches have been proposed. Existing approaches to JavaScript malware detection mostly rely on static analysis with machine learning techniques or dynamic analysis with behavioral abnormality checking. However, beyond

mere detection, there are few studies that focus on malware classification according to vulnerability or attack behaviors.

In 2009, Likarish *et al.* [46] adopted four classifiers (i.e., NB, ADTree, SVM and RIPPER) to detect obfuscated JavaScript malware. The assumption of their approach is that malware utilizes obfuscation to hide the exploits and to evade the detection.

In 2010, JSAND (JavaScript Anomaly-based aNalysis and Detection) [56] was presented to detect drive-by download attacks. JSAND identifies ten features characterizing intrinsic events of a drive-by download attack, and then uses these features for machine learning technique (NB) to detect malicious samples. These ten features are extracted from four aspects (redirection, de-obfuscation, environmental context, and exploitation) by dynamic anomaly detection with emulation in HtmlUnit [36].

Also in 2010, CUJO [57] uses hybrid analysis to on-the-fly capture program information (by static analysis) and execution traces (by dynamic analysis) of the JavaScript program. The results of the dynamic and static analysis are subsequently processed by *q-grams* to extract features used for SVM-based classification. The authors explored the distinct features for heap-spraying attacks and obfuscated attacks. But they failed to examine features for obfuscated but benign scripts and overlooked classification according to vulnerabilities or attack behaviors as we do in this work.

Later in 2011, Canali *et al.* [5] proposed a filter, called PROPHILER, to quickly filter out benign pages and forward to the costly analysis tools only the suspicious pages that are highly malicious. The filter's detection models are learned based on 70 features extracted from HTML contents of a page, from the associated JavaScript code, and from the corresponding URL. Then, multiple classifiers, BN, J48, Logistic, RT, and RF, are used to classify the malicious and benign web pages. For JavaScript, the FP of these classifiers is good, ranging from 0.0% to 1.7%, but the FN ranges from 18.1% to 81%, which is worse than our approach.

In the same year, AST is used to extract characteristics for malicious JavaScript detection. Curtsinger *et al.* [4] presented the tool ZOZZLE that predicates the benignity or maliciousness of JavaScript by leveraging features associated with AST contextual

information. Around one to two thousand of features are extracted from the hierarchical structure and texts in ASTs, and then the classifier NB is applied. To remove the noise due to obfuscation, ZOZZLE uses Detours binary instrumentation to get the final, unpacked code for accurate AST generation. After tuning up with different setups, the authors reported the FP ranges from 0.0003% to 4.5%, and the FN from 1.26% to 11.08%.

To sum up, the mainstream JavaScript malware detection tools mostly learn the features of each malware. While our approach group malware samples together according to their behaviors and then learn a representative model for each malware type. As a result, our approach can detect unseen malware variants.

3.8 Conclusion

A key to protecting the user from exploitation is the rigorous eliminating of malware on the Internet. To this end, we have proposed a method for accelerating the process of manual malware detection by suggesting potentially malware and their attack types to an analyst or an end-user. Our method not only learned features of maliciousness but also of attack type. Therefore, we can tell not only presence of malware, but malicious behaviors with attack approach information by virtue of dynamic attack confirmation. We also demonstrated our effectiveness and efficiency through empirical wild prediction. Among over 1,400,000 scripts, we find over 1,500 malware with eight attack types. Our detection speed is also scalable with below 80ms per script. The attack type classification also takes around one second for each malware, combining machine learning classifiers and dynamic attack confirmation.

4

JavaScript Malware Behavior Modelling

In the previous chapter, we introduced our machine learning-based JavaScript malware detection work. In this chapter, we further analyze the detailed behaviors of JavaScript malware and try to assist analysts to understand the behaviors of attacks launched by malware.

4.1 Introduction

The challenges in JavaScript behavior modeling stem from two aspects: (1) JavaScript is a dynamic language with runtime generation and dynamic typing, which rule out the static methods. To effectively analyze behaviors of JavaScript, we need a compact representation of JavaScript dynamic executions and a powerful framework for dynamic

analysis. (2) A meaningful classification according to attack types is important, especially for identifying new attacks. Such classification usually requires human expertise and is error-prone. Thus, an automated way is needed to learn attack behaviors and do the classification.

In this work, to overcome the first challenge, we propose to analyze JavaScript program behaviors by focusing on *browser-level system calls*. As system calls or actions are the interactions of a program (i.e., the web page in this work) with its environment (i.e., the browser in this work), it is effective to model program behaviors based on system calls or actions [58, 59]. In this work, we propose to use the DFA of the systems calls to model attack behaviors, which is inspired by the work on program behavior modeling [6]. The language accepted by the DFA includes all possible malicious executions of this type of attack, which captures the variants of the same attack.

To address the second challenge, we propose a dynamic analysis framework, named JS*, to automatically learn a DFA for each JavaScript attack type. First, given a set of (both benign and malicious) execution traces of malicious JavaScript, we perform a preprocessing to simplify these traces by removing security-irrelevant system calls. The simplified (abstracted) traces are called action sequences. Second, we develop an online learning algorithm based on the L* algorithm [7], which uses dynamic execution sequences to learn a DFA of the JavaScript malware.

Lastly, the inferred DFA serves as an abstract behavior model to identify variants of the modeled attack type, which can be used for both malware detection and classification. Given a suspicious JavaScript sample, the execution of this sample produces a concrete execution trace. Checking if this trace is accepted by the inferred abstract behavior model of each attack type indicates whether this trace belongs to this certain type of attack.

4.2 JavaScript Malware Behavior Modelling

To model the behavior of a JavaScript malware in dynamic analysis, system calls are usually gathered and used [60]. As system calls are the interactions of a program with the contextual system, it is effective to use system calls as the sources for identification of any security-relevant violation. Thus, the characteristics of visible system calls unveil unintended interactions, especially those malicious ones. In the implementation, we instrumented open-source Firefox. Concretely, we hooked the XPCOM [50] layer. XPCOM is a cross-platform component model for Mozilla, similar to the Component Object Model of Microsoft.

A system call is a tuple $sc = (I, M, N_p, S_p, T_r,)$, where I is the interface name of sc, M is the method name, N_p is the number of parameters, S_p is the list of arguments, and T_r is the return type. A JavaScript execution trace is defined as a sequence of system calls $\pi = \langle sc_1, sc_2, \ldots, sc_n \rangle$, occurring in a chronological order.

Example 4.1. ***A running example of a web-based attack***. *We list the JavaScript code fragment of an attack in Figure 4.1. Due to the vulnerability of CoolPreviews (Mozilla Firefox Extension) [61], via a link pointing to a data URI which embeds the Cross-Site scripting payload, the malicious page can inject exploiting code that is rendered and executed in the Chrome privileged zone*[1]*. Although a calculator application is used in Figure 4.1, arbitrary code can be executed. It is forbidden for normal JavaScript code to access the file system and create a process. However, due to the above vulnerability of CoolPreviews, the forbidden behaviors are made possible. Such code can be executed in the chrome privileged Firefox zone.*

Figure 4.2 shows the related system calls in a malicious trace of the attack in Figure 4.1. Each row represents a security relevant system call (see Section 4.3.2), where the five elements inside a system call are split by "|". The trace is shown on the right column only contains 12 security relevant system calls after filtering out the security irrelevant ones from the original hundreds of system calls. nsIIOService2 *is the interface name;*

[1] The Chrome privilege grants the JavaScript code the permission to do everything in the browser, which is similar to the root permission in OS. By default, JS is not allowed to create a file or a process outside the sandbox with Chrome privilege.

newURI is the method name; 3 means parameter number; the next part prints the values of parameters; void shows the type of the return value.

```
<a href='javascript:
var file=Components.classes["@mozilla.org/file/local;1"].
       createInstance(Components.interfaces.nsILocalFile);
var path = "/usr/bin/gnome-calculator";
file.initWithPath(path);
var proc=Components.classes["@mozilla.org/process/util;1"].
         createInstance(Components.interfaces.nsIProcess);
proc.init(file);
proc.run(true,[path],1);
'></a>
```

Figure 4.1: An Exploit to the Vulnerability of CoolPreviews

System calls sc_i**:** $(I \mid M \mid N_p \mid S_p \mid T_r)$	**Actions** a_i
nsIIOService2 | newURI | 3 | data:text/html;base64,PHN...; | void	$\Rightarrow a$
nsIURI | scheme | 0 | | void	$\Rightarrow b$
nsIPrefBranch | getComplexValue | 2 | intl.ellipsis;object; | void	$\Rightarrow n_1$
nsIPrefLocalizedString | data | 0 | | void	$\Rightarrow n_2$
nsIPrefBranch | getBoolPref | 1 | devtools.inspector.enabled; | void	$\Rightarrow c$
nsIPrefBranch | getCharPref | 1 | preview.enable; | void	$\Rightarrow c$
nsIIOService | newURI | 3 | data:text/html;base64,PHN...; | void	$\Rightarrow a$
nsIURI | scheme | 0 | | void	$\Rightarrow b$
nsILocalFile | initWithPath | 1 | /usr/bin/gnome-calculator; | void	$\Rightarrow d$
nsIProcess | init | 1 | object; | void	$\Rightarrow e$
nsIProcess | run | 3 | true;object;1; | void	$\Rightarrow f$
nsISecureBrowserUI | init | 1 | object; | void	$\Rightarrow n_3$

Figure 4.2: A Concrete Execution Trace and the Common Actions in the Attack of Figure 4.1

4.3 Approach

In this section, we present the overview of the proposed approach JS*, and then elaborate on trace collection and preprocessing, with the running example in Figure 4.1. We describe how an attack is modeled in the form of a DFA and how the DFA is used to detect variants of this attack.

4.3.1 Overview

The work-flow of our approach shown in Figure 4.3 contains several steps, each of which requires its own input and produces the corresponding output.

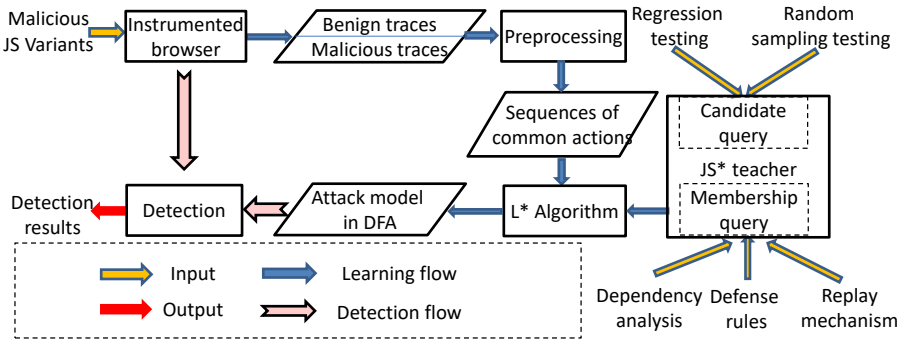


Figure 4.3: The Work-Flow of JS*

The first step is to collect execution traces for training model. This is done by instrumenting browser, in this work, Firefox. We use instrumented browsers to visit malicious samples. Each sample is executed several times and thus several execution traces are collected.

Given execution traces collected, we preprocess them to filter out system calls in execution trace which are irrelevant to security behaviors. Left core system call traces are used to train a DFA. The alphabet Σ of DFA $\mathscr{D}$ is the common actions. The DFA $\mathscr{D}$ should be as small as possible. Our learning algorithm is similar to L*. The main component includes a teacher who can answer membership and candidate queries. How our approach answer membership queries are illustrated in Section 4.4.1. And the answer to candidate queries is shown in Section 4.4.2. The learned DFA is then used to categorize execution trace of suspicious JavaScript fragments.

The final output of JS* is a report that judges whether a suspicious JavaScript sample is really a malware variant. Some other important intermediate result is the learned DFAs that describe accurate attack models and capture the core events. Such learned attack models can be applied to other security purposes.

4.3.2 Trace Preprocessing

Given the suspicious samples, the dynamic analysis tool can be used to capture the execution traces of the malware at runtime in its hosted system. To have the careful and comprehensive observation of the suspicious malware, all the system calls and the events relevant to the process of this malware should be traced down. Some human experts also need to be involved to judge which traces will lead to the successful attack and which ones will not. Furthermore, the dynamic traces collection for malware is not limited to operating systems. Under the current Internet environment, analysis of some malicious web applications with script attack entails capturing dynamic execution traces in a Web browser.

One actual collected execution trace during a time interval of several seconds can contain hundreds of or even up to thousands of system calls. We start our approach by capturing browser-level system call traces during loading and rendering a web page through instrumenting the browser (see Section 4.5.1 for implementation details). Each recorded trace contains thousands of system calls, most of which are irrelevant to browser security. To precisely and concisely model attacks, removing security-irrelevant system calls and mapping the rest similar ones into high-level actions prelude further analysis.

Such irrelevant system calls will bring in noise and affect the learning result of the L^* algorithm, as considering security-irrelevant system calls in DFA may complicate the attack model and fail to capture the essence of the problem as too many irrelevant system calls may greatly increase the number of minimum traces required for learning a reasonable DFA and complicate the learned attack model. Thus, it is necessary to remove such noise before the application of the L^* algorithm. Another problem in the direct analysis of execution traces is the existence of multiple similar system calls of the same action. Clustering the attack-relevant system calls into coarse-grained categories preludes further grouping of system calls to actions.

Most attack-irrelevant browser-level system calls at most perform some assistant operations to the security relevant system calls, rather than directly sabotage. These security

relevant system calls could do many operations with root permission, e.g. directly crashing the browser or even damaging the data in the computer. Thus, among the enormous XPCOM interfaces and system calls, we mainly concern the security-relevant ones. To infer a general and meaningful attack model represented by the learned DFA, ignoring the security irrelevant system calls is an optimized step before application of L^* algorithm.

In Linux based system, there are 190 important system calls[2], which are the operations on file management, process management, error handling, memory management, network access, message queue, time-related and system-wide issues[3]. In a Windows system, there are also about 100 important system calls, like NtCreateProcess and NtOpenFile. These system calls fall into the following categories: file system, registry, process/thread, synchronization, network, and section.

Hence, we adopt a statistical approach in this work. We have collected 5,000 malicious samples from three JavaScript malware sources (see Section 4.5.2 for details). We found that 667 out of the 1,948 interfaces in XPCOM [50] never appear in system call traces of the malicious samples at hand. Thus, system calls to methods in these 667 interfaces are considered as no security risks, i.e., *security-irrelevant system calls*, e.g., method nsIDOMWindow.GetscreenY that gets the Y coordinate of a window. In our study, methods frequently appearing in malicious samples are *security-relevant system calls*, e.g., method nsIProcess.run that executes a process.

All these interfaces that contain browser-level system calls can be classified into 54 categories according to their functionalities. According to the security severity, these 54 categories can be further associated with four levels of security shown in Table 4.1. For example, interfaces in *Process launching* could run a malicious executable if misused; interfaces in *Cookie* can leak the user privacy by the malicious script, and interfaces in *Dynamic code execution* can dynamically invoke the execution of some scripts with risks. Interfaces at None level are not deemed security risks, i.e., interfaces *Browser core* and *Cryptograph* are invulnerable to external JavaScript attacks.

[2]LINUX System Call Quick Reference: http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf

[3]Classification and Grouping of Linux System Calls: http://seclab.cs.sunysb.edu/sekar/papers/syscallclassif.htm

Table 4.1: Categorizing XPCOM Interfaces into Four Security Levels

Rating	Names of Interfaces
`High`	Arbitrary file access; Process launching; Download; XPInstall; Network access via XPCOM API; Update; DOM injection
`Medium`	Password; Login; Cookie; Network access via XMLHTTPRequest; Addons management; Changing firefox preferences; Profile
`Low`	History; Bookmark; Clipboard; Dynamic code execution
`None`	Accessibility; Browser core; Auto complete; Log to console; Searching; Spell checking; DOM; Editor; Internationalization; Offline cache; XML parser; Network utilities; RSS/RDF; Data types and structures; Streams; Memory management; Thread management; Component management; Additional XPCOM services; Javascript core; Javascript debugger; XPConnect; Authentication; Certification; Cryptograph; Additional security interfaces; Doc handling; Transaction management; Web worker; Window management; Print; DB access; Images; ZIP/JAR process; JVM; Plugins

However, not all the methods in the interfaces associated with High/Medium/Low are security relevant, and we only consider the methods that are *not* setter/getter/constructor as *relevant system calls*. The browser-level system calls in the other severity category are considered irrelevant to the attack.

Given the categorization according to function, permission and security severity level, we could find most actually attack-relevant system calls. However, there can be some extreme cases: some traces with the same set of relevant system calls and also the similar parameters lead to different results. Some are malicious but some are benign. Such cases indicate that the relevant system calls we collected may not be the minimum set of the actually relevant system calls required for the attack model. To solve this problem, we may need to further include those originally irrelevant system calls in the malicious back into the relevant system call set.

Action Abstraction. Some system calls provide similar or identical functionalities, e.g., in Figure 4.2, nsIPrefBranch.getBoolPref and nsIPrefBranch.getIntPref get the Bool and Int type preference data; nsIIOService1.newURI and nsIIOService2.newURI both construct a new URI, respectively. Treating these similar systems calls as atomic behaviors will be unreasonable and bring the noise to the following analysis. To simplify the further building of alphabet for L^*, we abstract system calls with similar functionalities as

Algorithm 1 extractCommonActions

Input: $S_\pi = \{\pi_1, \pi_2, ...\pi_n\}$, a set of malicious system call traces
Output: M, the map whose key is an action and the value is the set of system calls abstracted by this action. Initially empty.
Output: Σ, the set of common actions relevant to the attack

```
1:  for each trace π ∈ S_π do
2:      for system call sc ∈ π do
3:          if ∃ a_k ∈ M.keys • IsSameAction(a_k, sc) then
4:              M.get(a_k).add(sc)
5:          else
6:              create a new action a_n according to sc
7:              M ← M ∪ {(a_n, sc)}
8:          end if
9:      end for
10: end for
11: Σ ← GetAction(π_1, M)
12: for trace π ∈ {π_2, ..., π_n} do
13:     Σ ← Σ ∩ GetAction(π, M)
14: end for
```

the same type of *action* for the purpose of variants detection. We name the actions alphabetically starting from 'a' in this work. After the abstraction, malicious traces based on system calls become abstracted action sequences. Due to multitasking of the browser, each execution of the same JavaScript code may produce different action sequences but with the identical attacking route. Intersecting these action sequences extracts common actions, making visible common behaviors of malicious traces.

This step is to find the minimum common system calls that are attack-relevant. Note that the general security-relevant system calls are not necessarily relevant to the specific attack. Algorithm 1 illustrates how system calls are mapped into actions and then how common actions are extracted from the given set of malicious traces S_π. First, the map *M* is built from Lines 1 to 7. At Line 3, each system call sc in π from S_π is compared with each key a_k in *M* by calling *IsSameAction* to check the functionality similarity, at Line 3. The checking is based on the following strategy. If the fully qualified name of sc is similar to the fully qualified name of any system call in set $M.get(a_k)$, sc belongs to $M.get(a_k)$. Being similar is satisfied if string similarity according to Levenshtein Distance [62] is above a threshold, 80% in our study [63]. We also refine the results of this step via the manual check, considering the limited size of system calls involved in each type of attack. Method $M.get(a_k)$ returns those system calls abstracted by a_k.

If *IsSameAction* returns true, i.e., sc can be abstracted by action a_k, sc is added into

Malicious Action Sequences S_π:	**Benign Action Sequences S'_π:**
$\pi_{m1} : \langle c.a.b.d.e.f \rangle$	$\pi_{b1} : \langle c.b.d.e.f \rangle$
$\pi_{m2} : \langle a.c.b.d.e.f \rangle$	$\pi_{b2} : \langle a.c.d.e.f \rangle$
$\pi_{m3} : \langle a.b.c.d.e.f \rangle$	$\pi_{b3} : \langle a.b.c.e.f \rangle$
$\pi_{m4} : \langle a.b.d.c.e.f \rangle$	$\pi_{b4} : \langle a.b.d.c.f \rangle$
$\pi_{m5} : \langle a.b.d.e.c.f \rangle$	$\pi_{b5} : \langle a.b.d.e.c \rangle$
$\pi_{m6} : \langle a.b.d.e.f.c \rangle$	$\pi_{b6} : \langle c.b.a.d.e.f \rangle$
$\pi_{m7} : \langle c.a.c.b.c.d.c.e.c.f \rangle$	$\pi_{b7} : \langle c.a.b.e.d.f \rangle$
$\pi_{m8} : \langle a.b.c.c.a.b.d.e.f \rangle$	$\pi_{b8} : \langle c.a.b.d.f.e \rangle$

Figure 4.4: Representing Traces as Sequences of Common Actions

$M.get(a_k)$ (Line 4). Otherwise, a new action a_n is created according to the next available alphabet (ASCII letter in our implementation) at Line 6; and the pair (a_n, sc) is added to M at Line 7. Practically, action abstraction step is manually verified, as the size of the alphabet is small (see Section 4.5). After M is built up, method *GetAction*(π_1,M) at Line 11 collects the actions in π_1 and assigns to Σ. From Lines 11 to 13, all action sequences are iteratively used to build the set of common actions Σ, actions appearing in all traces in S_π.

In the running example, we have eight such concrete system call traces, and after applying the Algorithm 1, the common actions shown in the alphabet in Figure 4.2 listed in are extracted. Note that extracting common actions is only conducted on malicious traces. Only after that, the available benign traces are represented as sequences of common actions. Suppose we collected eight malicious and eight benign concrete system call traces, we apply action abstraction and intersection on these eight malicious traces to build the set of common actions, i.e., $\{a, b, c, d, e, f\}$ in Figure 4.2. Finally, in Figure 4.4 we represent these 16 traces in sequences of common actions, which serve as the input for the JS* learning approach.

4.4 JS* Learning Framework

This section is devoted to the explanation of the proposed JS* learning framework. Our approach is based on the L* algorithm, which learns a DFA from a set of strings [7]. For string learning, L* contains a teacher to answer membership queries and candidate queries via substring or regular expression matching. In our study, we combine domain knowledge (e.g., defense rules), program analysis (e.g., data dependency), and other techniques (e.g., replay mechanism, random sampling) to implement an effective teacher to answer these two queries (see Sections 4.4.1 and 4.4.2).

4.4.1 Membership Query

In our running example, we captured eight malicious traces. After these traces are preprocessed as described in Section 4.3.2, we can get the alphabet $\Sigma = \{a, b, c, d, e, f\}$ of the language that describes the exemplary *web-based* attack $\mathscr{L}$, given the sequences of common actions shown in Figure 4.4. Using the alphabet Σ, we can represent these eight traces in eight alphabetical sequences, which can be used for L* learning.

The L* algorithm assumes that the expected attack model M can be represented in the form of a DFA $\mathscr{D}$ with a fixed alphabet Σ. The DFA $\mathscr{D}$ should be with the minimal number of states that accepts the language $\mathscr{L}$, the simplest DFA on $\mathscr{L}$. To achieve this goal, the L* algorithm interacts with the implemented online teacher by asking membership and candidate queries to make the observation table closed and consistent.

An ideal implementation of the online teacher for membership queries should be able to correctly answer if the system call sequence $\pi_a \in \Sigma^*$ should be accepted (i.e., whether π_a represents an attack of the modeled type) in polynomial time. Practically, it is infeasible to extract all possible traces for a JavaScript program and judge their maliciousness. Here, we propose a feasible and efficient way to utilize defense rules, data dependency analysis, and JavaScript replay mechanism to answer membership queries on the fly, as elaborated below.

4.4.1.1 Browser Defense Rule

Defense rules refer to security policies commonly used by the mainstream browsers to detect malicious attacks. Violation of such rules indicates possible security risks in existing studies [64]. A commonly used rule is the *same origin policy*. The policy permits scripts running on pages from the same origin. A combination of the same scheme (or protocol), hostname, and port number. In [65, 66], this policy is employed for malware detection.

Mozilla also implements zone-based rules, *Configurable Security Policies* (CAPS) [67]. The idea of CAPS is to enable users to configure the general security policies as well as to customize the specific policies for different security zones. By manually modifying the *user.js* configuration file, users can assign three levels of permission, namely *noAccess*, *sameOrigin*, and *allAccess*, to different methods for various websites. For example, a user can set *noAccess* to *Window.confirm* for website `http://example.com/`, denying all scripts access from this website to *confirm* the property of an object of type *Window*.

Another rule is the *signed-script policy*. To exercise fine-grained control over activities beyond normally allowed JavaScript, a signed script can request expanded privileges to access the restricted information and functionality by using netscape.security.Privilege manager .enable-privilege(). It recommends to enable privileges only when necessary and to disable them soon after use. Failure to do so introduces risks. This rule is used in [65, 68]. In our study, these defense rules are applied to the URL source, information, and permission of executed JavaScript code. By combining these rules in the teacher, we can test a trace. A violation of any defense rule indicates that the tested trace is malicious.

Note that defense rules are mainly used to check the trace that is tested by the JS* on the fly. Besides, using defense rules can only indicate whether a trace is benign or not [65], but fail to model the attack behaviors.

4.4.1.2 Data Dependency Analysis

Given the existing training traces, data dependency analysis is adopted to find equivalent permutations (EPs) of an action sequence. The assumption that the order of two mutually independent system calls (or actions) does not matter is reasonable [58][59]. Thus, we adopt this assumption and infer EPs via data dependency analysis. For example, given malicious sequence $\pi_1 = \langle a_1, a_2, a_3 \rangle$, where a_3 has data dependency[4] on a_1 and a_2, denoted as $\{a_3 \leftarrow (a_1, a_2)\}$, we can infer $\pi_2 = \langle a_2, a_1, a_3 \rangle$ is also malicious. So π_2 and π_1 are EPs.

Given a training set of malicious sequences S_π and an unknown system call sequence π_a, Algorithm 2 shows how to test if π_a is an EP according to the sequences in S_π. The basic idea is to get the data dependency closures among actions inside a sequence π_i by invoking method *getDependencyClosure*(π_i). In two steps, this method builds the dependency closure[5]. Given π_i, JS* gets all direct dependencies among system calls inside this trace, from which actions are abstracted. If two system calls have a direct data dependency (see Section 4.5.1), the two relevant abstracted actions have a direct data dependency. Second, this method propagates the direct data dependency relationship among actions into a closed transitive indirect data dependency. In the running example, this method returns two closures $\{b \leftarrow a\}$ and $\{(e,f) \leftarrow d\}$. As c is independent, actually, π_{m1} to π_{m6} (and other similar sequences only with different c positions) are all EPs.

Algorithm 2 isEqualPermutation

Input: $S_\pi = \{\pi_1, \pi_2, ...\pi_n\}$, a set of malicious action sequences
Input: π_a, a given action sequence
Output: $true$ or $false$, if π_a is an equivalent permutation

```
1: S_eπ ← ∅
2: for each sequence π_i ∈ S_π do
3:     S_eπ ← S_eπ ∪ getPermutations(π_i, S_dc)
4: end for
5: if π_a ∈ S_eπ then
6:     return true
7: end if
8: return false
```

[4] The data dependency is calculated based on the original system calls of the actions.

[5] A dependency closure is a transitive relationship among actions, and each in this closure is directly or indirectly involved in data dependencies on others.

By *getPermutations*, the mutually independent relation between actions (e.g., a_1 and a_2 in the above case of π_1, c and other actions in the case of π_{m1}) infers the equivalent permutations. All EPs should not equal to an existing benign sequence, and then be stored into a set for comparison with π_a. A right match indicates an EP. Practically, the calculation of $S_{e\pi}$ from lines 1 to 4 in Algorithm 2 is pre-built once as preprocessing. It is not necessary to calculate $S_{e\pi}$ every time. Due to the limited length of action sequence and the small size of training sequences, *getPermutations* is scalable in reality, e.g., for the partial DFA in Figure 4.12, there is the dependency closures: $\{n \leftarrow m \leftarrow l \leftarrow k \leftarrow j \leftarrow i \leftarrow (g,h) \leftarrow f \leftarrow (c,d,e) \leftarrow b \leftarrow a\}$. Thus, (g,h) and (c,d,e) are two sets of independent actions that lead to 12 EPs—multiplied by 2 for (g,h) and 6 for (c,d,e).

Data dependency analysis is used to check the benignity of traces on the fly in our teacher implementation. Based on dependency closures inferred from malicious sequences, JS* identifies benign sequences not holding these closures, and find malicious EPs. Sequences $\langle a.b.d.e.f \rangle$ (removing prefix c from π_{m1}) and $\langle c.a.b.d.f.e \rangle$ (partial reordering of π_{m1}), which satisfy dependency closures but are unknown for benignity, will be tested with replay mechanism as presented below.

JEIS statement	**Action**
1. nsIPrefBranch.getBoolPref("devtools.inspector.enabled");	$\Rightarrow c$
2. var nsIIOService2=Components.classes["@mozilla.org/network/ io-service;1"].getService(Components.interfaces.nsIIOService);	N.A.
3. var nsIURI=nsIIOService2.newURI("data:text/html;base64, PHNjcmlwd...",null,null);	$\Rightarrow a$
4. nsIURI.scheme();	$\Rightarrow b$
5. var nsILocalFile=Components.classes["@mozilla.org/ file/local;1"].createInstance(Components.interfaces.nsILocalFile);	N.A.
6. nsILocalFile.initWithPath("/usr/bin/gnome-calculator");	$\Rightarrow d$
7. var nsIProcess==Components.classes["@mozilla.org/ process/util;1"].createInstance(Components.interfaces.nsIProcess);	N.A.
8. nsIProcess.init(nsILocalFile);	$\Rightarrow e$
9. nsIProcess.run(true,['/usr/bin/gnome-calculator'],1);	$\Rightarrow f$

Figure 4.5: The JEIS Statements Reverse-Engineered from Action Sequence π_{m1}

4.4.1.3 Trace Replay

Replay mechanism is implemented to dynamically test maliciousness of an inquired trace during online learning. It is implemented using JavaScript's Equivalent Intermediate Script (JEIS), which is the intermediate code rather than the source code. The basic idea is to manually craft a JEIS with Chrome privileges, according to a given action sequence. We do not craft JEIS from scratch but create a JEIS by adding, deleting or reordering JEIS statements from the reverse-engineered JEIS of existing training traces.

We implement replay mechanism as instrumentation to FireFox to attain Chrome privilege. The artificial system call trace to be replayed is not conjectured up. Instead, the required order of system calls is guided by the L^* algorithm. As explained in Section 4.4, a new sequence usually has a prefix or suffix added to or deducted from a previous trace that has been tested in a membership query. For example, during the L^* learning, a membership query asks the maliciousness of an action sequence $\langle a.b.d.e.f \rangle$, which is similar to $\pi_{m1} : \langle c.a.b.d.e.f \rangle$ but without the prefix c.

We get the JEIS of π_{m1} that is shown in Figure 4.5, then remove the intermediary script relevant to c, and finally craft the expected JEIS. Executing this JEIS by the JavaScript engine realizes the functionality of replaying the action sequence $\langle a.b.d.e.f \rangle$. Actually, as executing this JEIS leads to the same result of executing the JEIS of π_{m1}, $\langle a.b.d.e.f \rangle$ is also malicious.

Given a system call sequence for opening an HTML file in the browser in Figure 4.6, we need to craft the corresponding *replayable* trace with Chrome privileges in Figure 4.7.

System calls sc_i**:** $(I \mid M \mid N_p \mid S_p \mid T_r \mid T)$
nsITextToSubURI | unEscapeNonAsciiURI | argc:2 | ISO-8859-1, https://www.google.com.sg | void | 19:42:34
nsIWebNavigation | document | argc:0 | void | 19:42:34 |

Figure 4.6: Sample Call Sequence Fragment

An interface in Mozilla is a definition of a set of functionality that could be implemented by components. Each component implements the functionality as described by interfaces. They can be referred to using a string, e.g., *'@mozilla.org/intl/texttosuburi;1'*.

```
var nsIIOService2=Components.classes["@mozilla.org/network/io-service;1"].
getService(Components.interfaces.nsIIOService);
var nsIURI=nsIIOService2.newURI("data:text/html;base64,PHNjcmlwd...", null, null);
nsIURI.scheme();
var nsILocalFile=Components.classes["@mozilla.org/file/local;1"].
createInstance(Components.interfaces.nsILocalFile);
nsILocalFile.initWithPath("/usr/bin/gnome-calculator");
var nsIProcess=Components.classes["@mozilla.org/process/util;1"].
createInstance(Components.interfaces.nsIProcess);
nsIProcess.init(nsILocalFile);
nsIProcess.run(true,['/usr/bin/gnome-calculator'],1);
```

Figure 4.7: The JEIS for the Malicious Action Sequence $\langle a.b.d.e.f \rangle$

This string is called a contract ID. To replay one single system call, we need to retrieve a component corresponding to its interface name. The contract ID of the component can be used to get the component, and we maintain a mapping table between the contract ID and interface name. It can also be attained from [69].

A challenge in replaying a trace is how to assign the proper arguments for system calls involved in the JEIS trace. Each system call instance must specify its interface name, method name, the number, and values of arguments and the type of return value, which are mandatory elements for a system call and also prerequisites of replay. As the elements of a system call is stored in our infrastructure, the arguments can be reused when this system call is replayed. The primitive argument types can be straightforwardly reused, but for the object type argument, our infrastructure has direct memory access to capture the address and the length of that object for reuse. Note that our replay mechanism generally supports the JEIS with arbitrary arguments.

The execution of JEIS outputs three types of outcomes. First, the execution of JEIS triggers some defence rules, or be against some heuristic rules that detect malware in the sandbox of JavaScript engine [65], e.g., a JEIS with System call nsIProcess.init whose origin is from an external website violates CAPS, as its interface nsIProcess cannot be executed without Chrome privilege. In such case, we consider the trace is malicious and causes potential risks. Second, the execution of JEIS may also cause some runtime exception or crash. Rather than considering the replayed sequence as malicious, the exception or crash actually indicates that the replayed sequence is infeasible, e.g., according to $\pi_{m3} : \langle a.b.c.d.e.f \rangle$, we want to craft the JEIS of $\langle a.b.c.d.f.e \rangle$ that has a

reversed order of e and f, shown in Figure 4.8. Inverting e and f causes an unpredictable error for the JavaScript engine, as nslProcess.run is executed before nslProcess.init. So $\langle a.b.c.d.f.e \rangle$ is infeasible, and this infeasible sequence means that the outcome is benign. Lastly, if the execution of the JEIS trace fails to lead to the same result as that of malicious traces and violates no rules. We consider it as benign.

```
var nsIIOService2=Components.classes["@mozilla.org/network/io-service;1"].
getService(Components.interfaces.nsIIOService);
var nsIURI=nsIIOService2.newURI("data:text/html;base64,PHNjcmlwd...", null, null);
nsIURI.scheme();
var nsIPrefBranch=XPCOMUtils.
getService("@mozilla.org/preferences-service;1",nsIPrefBranch.class);
nsIPrefBranch.getBoolPref("devtools.inspector.enabled");
nsIPrefBranch.getCharPref("preview.enable");
var nsILocalFile=Components.classes["@mozilla.org/file/local;1"].
createInstance(Components.interfaces.nsILocalFile);
nsILocalFile.initWithPath("/usr/bin/gnome-calculator");
var nsIProcess=Components.classes["@mozilla.org/process/util;1"].
createInstance(Components.interfaces.nsIProcess);
nsIProcess.run(true,['/usr/bin/gnome-calculator'],1);
nsIProcess.init(nsILocalFile);
```

Figure 4.8: The JEIS for the Benign Action Sequence $\langle a.b.c.d.f.e \rangle$

4.4.1.4 Membership Query Algorithm

Given malicious action sequences S_π and benign sequences S'_π, e.g., the sequences in Figure 4.4, Algorithm 3 describes the process in answering the membership query regarding the given action sequence π_a.

Algorithm 3 membershipQuery

Input: π_a, a given action sequence, which cannot be null
Output: $true$ or $false$, if π_a should be accepted by the expected DFA.

1: **if** $\exists \pi_i \in S_\pi \bullet \pi_i$ is a prefix of π_a **then**
2: **return** $true$
3: **end if**
4: **if** $\exists \pi'_i \in S'_\pi \bullet \pi_a == \pi'_i$ **then**
5: **return** $false$
6: **end if**
7: **if** $isEqualPermutation(\pi_a, S_\pi)$ **then**
8: **return** $true$
9: **end if**
10: **if** π_a violates any defense rule **then**
11: **return** $true$
12: **end if**
13: **return** $Replay(\pi_a)$

First, if any $\pi_i \in S_\pi$ is a prefix of π_a (Line 1), i.e., π_a equals to or starts with π_i, π_a must be malicious. If π_a equals any sequence in S'_π (Line 4), π_a must be benign. The assumption is that the same action sequence (or with the same prefix) leads to the same result even if the parameters for actions inside are different. Actually, two different sets of parameters for two same action sequences belong to the same input equivalent class. The actually different parameters will lead to different action sequences due to the change in control flow. Then *isEqualPermutation*(π_a, S_π) method at Line 7 defined in Algorithm 2 is called to check whether π_a is an EP, and a *true* answer means that π_a is also a malicious EP. The next step is to check if π_a violates the defense rules in Section 4.4.1.1 at Line 10. Finally, method $Replay(\pi_a)$ at Line 13 is called to concretely execute π_a using replay mechanism to verify its benignity.

In our running example, an example of satisfying the check at Line 1 is action sequence $\langle c.a.b.d.e.f.a \rangle$, which is considered as malicious as it starts with an existing malicious sequence π_{m1}. An example for *isEqualPermutation*(π_a, S_π) at Line 7 is to check action sequences $\langle c.a.b.e.f.d \rangle$ and $\langle a.b.c.c.d.e.f \rangle$. According to the collected traces like π_{m8} in Figure 4.2, we found that action b has a data dependency on a; c is independent; meanwhile e and f have a data dependency on d. Thus, $\langle c.a.b.e.f.d \rangle$ is not an equivalent permutation of existing malicious sequences like $\langle c.a.b.d.e.f \rangle$. But $\langle a.b.c.c.d.e.f \rangle$ is malicious as it satisfies the identified data dependency, similar to π_{m3} but with only one more independent c.

At Line 10, our example in Figure 4.2 does not violate any rule, as all the scripts are from the same origin. Lastly, $Replay(\pi_a)$ at Line 13 is effective in running the left uncertain sequences ranging from simple ones $\langle a.b \rangle$ to those complicated ones like $\langle b.a.f.a.b.f.e.d.a.e.d.d.f \rangle$ for membership querying. Actually, *Replay*(π_a) all returns *false* for these two queries, as the two replayed sequences produce no malicious results (no resource oriented activities and no permission/rule violations).

4.4.2 Candidate Query

During learning, an intermediate DFA $\mathscr{C}$ is inferred after multiple membership queries. To judge if $\mathscr{C}$ is equivalent to the expected DFA $\mathscr{D}$ that accurately models the attack, an efficient algorithm is required for the validity check in polynomial time. When a candidate query is evaluated, two types of counterexamples can be found on $\mathscr{C}$—false positives and false negatives. The former means that a sequence accepted by $\mathscr{C}$ should be rejected by $\mathscr{D}$, while the latter means that a sequence rejected by $\mathscr{C}$ should be accepted by $\mathscr{D}$.

Given S_π and S'_π in Figure 4.4, Algorithm 4 illustrates how the teacher answers the candidate query for the given $\mathscr{C}$, based on regression testing and random sampling testing. First, at Line 1, each sequence π_i from the known sequence sets S_π, S'_π and the previously tested sequence set $S_{o\pi}$ is input to *membershipQuery* and also $\mathscr{C}$*.isAccepted*(). Method *membershipQuery*(π_i) at Line 2 returns true if π_i is accepted by $\mathscr{D}$. If $\mathscr{C}$ and $\mathscr{D}$ show different acceptance results for π_i, a counterexample π_i is found and returned. Second, a random walk function *randomWalks*() is used to generate $S_{n\pi}$, a new set of random sequences that include both accepted and rejected ones on $\mathscr{C}$. The rationale of using random walk is that sequences on $\mathscr{C}$ cannot be enumerated due to potential loops. *randomWalks*() at line 6 has three parameters, where $\mathscr{C}$ is the candidate DFA; $\mathscr{C}$*.stateSize*times* denotes the number of generated sequences and $times$ is an input constant to multiply; $\mathscr{C}$*.stateSize+extraLenLmt* is the maximum allowed length of generated sequences and *extraLenLmt* is the extra length that can be larger than $\mathscr{C}$*.stateSize*. Here, $\mathscr{C}$*.stateSize* denotes the size of total states in $\mathscr{C}$. Then, each sequence π_i in $S_{n\pi}$ is also given to *membershipQuery*() and $\mathscr{C}$*.isAccepted*() for acceptance check—an inconsistency indicates a counterexample. Note that any tested π_i from $S_{n\pi}$ is added to $S_{o\pi}$, which is used for regression testing in answering the next candidate query. Finally, if no counterexample is found, *NULL* is returned and the learning process stops.

In our running example, candidate queries are asked twice. The first candidate DFA $\mathscr{C}_1$ is inferred when P in the observation table only contains $\{\lambda, a, b, c, d, e, f\}$ with one state 0. For $\mathscr{C}_1$, Algorithm 4 returns a malicious sequence $\pi_{m1} : \langle c.a.b.d.e.f \rangle$ as a counterexample for further learning. Afterward, a candidate DFA $\mathscr{C}_2$ in Figure 4.9 (b)

is learned. According to regression testing and random sampling testing in Algorithm 4, no counterexample for $\mathscr{C}_2$ is found—$\mathscr{C}_2$ is equivalent to the expected $\mathscr{D}$.

Algorithm 4 candidateQuery

Input:

$\mathscr{C} \neq NULL$, the learned candidate DFA

$S_{o\pi}$, the set of old sequences that have been tested in previous calls of candidateQuery, initially being $\varnothing$ before any candidateQuery is called

Output:

π_{ce}, a counterexample sequence found

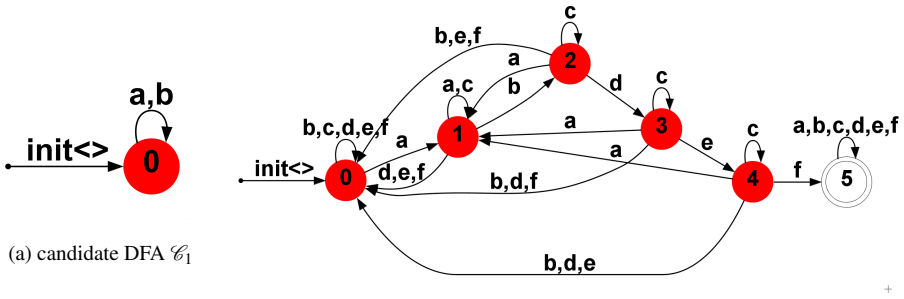
1: **for** each trace $\pi_i \in (S_\pi \cup S'_\pi \cup S_{o\pi})$ **do**
2: **if** $membershipQuery(\pi_i) \neq \mathscr{C}.isAccepted(\pi_i)$ **then**
3: **return** $\pi_{ce} \leftarrow \pi_i$
4: **end if**
5: **end for**
6: $S_{n\pi} \leftarrow randomWalks(\mathscr{C}, \mathscr{C}.stateSize * times, \mathscr{C}.stateSize + extraLengthLimit)$
7: **for** each trace $\pi_i \in S_{n\pi}$ **do**
8: $S_{o\pi} \leftarrow S_{o\pi} \cup \{\pi_i\}$
9: **if** $membershipQuery(\pi_i) \neq \mathscr{C}.isAccepted(\pi_i)$ **then**
10: **return** $\pi_{ce} \leftarrow \pi_i$
11: **end if**
12: **end for**
13: **return** $\pi_{ce} \leftarrow NULL$

4.4.3 The Learned DFA and Refinement

Given the 16 sequences in Figure 4.4, JS* undergoes 622 times of membership queries and 2 times of candidate queries (with *times=5* and *extraLenLmt=5* for method *randomWalks* in Algorithm 4), and infers a DFA $\mathscr{D}$ to model the attack—or equivalently a regular language $\mathscr{L} = (c)^* \cdot a \cdot (c)^* \cdot b \cdot (c)^* \cdot d \cdot (c)^* \cdot e \cdot (c)^* \cdot f \cdot (c)^*$ over $\Sigma = \{a, b, c, d, e, f\}$. To apply this DFA to detect malicious variants of this attack, a trace from a suspicious variant is collected and preprocessed to be converted into an action sequence π over Σ. An acceptance of π on $\mathscr{D}$ suggests that π is from a malicious trace.

The sequences in S_π in Figure 4.4 are all deterministic (being certainly malicious), and produced by the script with fixed arguments in Figure 4.2. In practice, the same sequence of actions might be sometimes malicious and sometimes benign, depending on the arguments of the calls. For instance, if the last argument of the statement `proc.run(true,[path],1)` in Figure 4.4 is changed from "1" to "0". Executing the new code

can produce the same sequences as those old ones in Figure 4.4, and all data dependencies inside these new sequences still hold. But the new sequence $\pi'_{m1} : \langle c.a.b.d.e.f \rangle$ is benign and creates no process since inside π'_{m1} the last argument of action f is "0", which makes nsIProcess.run include zero arguments from the argument list. One way to solve the problem is to model system calls with different arguments as different actions, for instance, we can represent the above as: $\pi_{m1} : \langle c.a.b.d.e.f(1) \rangle$ and $\pi'_{m1} : \langle c.a.b.d.e.f(0) \rangle$. To refine the DFA for such case, we derive new actions f_m and f_b from current action f. Here, f_m refers to the action $f(\{s_1, ...s_n\})$ with the argument set $\{s_1, ...s_n\}$ that produces malicious outcome like $f(1)$. Similarly, f_b refers to the action f with arguments that produce a benign outcome. Then with the new $\Sigma' = \{a, b, c, d, e, f_m, f_b\}$, JS* is applied again on the training sequences to learn a refined DFA. Thus, JS* supports refinement for nondeterministic sequences in a reactive way.



(a) candidate DFA $\mathscr{C}_1$

(b) candidate DFA $\mathscr{C}_2$

Figure 4.9: The Learned Candidate DFA $\mathscr{C}_1$ and $\mathscr{C}_2$

4.5 Implementation and Evaluation

In this section, we introduce the implementation and evaluation of JS*.

4.5.1 Capturing System Calls

Our implementation is instrumentation to the Firefox kernel. There are three layers in Firefox's JavaScript execution environment, i.e., the upper layer interpreter SpiderMonkey, the middle layer XPCOM and the lower layer libraries. At the upper layer,

Firefox JavaScript engine SpiderMonkey interprets input JavaScript code and invokes the related low layer libraries via XPConnect, which provides interaction between SpiderMonkey and XPCOM. Here, XPCOM (Cross Platform Component Object Model) is the middle layer and implemented by the low layer libraries, e.g., Gecko library for page rendering, Necko library for network access, etc.

As XPConnect delegates the low layer implementation and bridges the top and bottom layer, we hook calls to methods in XPConnect interfaces as JavaScript browser-layer system calls. The instrumented XPConnect layer intercepts the information of XPCOM system calls at runtime. Intercepted information includes interface name, method name, argument information (e.g., type, value and memory address), return value, call mode, file name, execution time, etc.

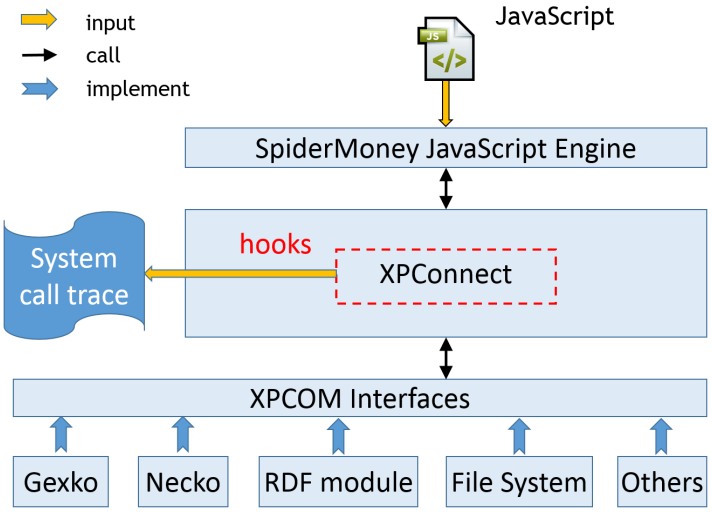


Figure 4.10: Firefox JavaScript Execution Environment

The intercepted information of system call objects, arguments and return value facilitates data dependency analysis among system calls. By comparing the arguments of each system call with the return value or the related object of another system call in terms of object type and memory address, it can be checked if there exists a direct data dependency. In the implementation, we also propagate data dependency check to calculate the transitive dependency relationship for ease of data dependency analysis in Section 4.4.1.2. If the type and the memory address of the last system call's return value equals that one of the next system call's arguments, we can conclude that they have a data dependency relationship. The frequency of calling interface is high. According to

one experiment, the frequency is average 879 times per minute. Therefore, interface calling information cannot be used as an elementary unit of analysis. What is more, individual interface calling cannot reveal any intact function.

4.5.2 Data Preparation and Setup

To evaluate JS* on the eight popular types of JavaScript attacks mentioned in Section 4.2, we collect 276 distinct malicious samples out of more than 1,000 real-world malware[6] that originate from various sources: 40 unique samples from VXHEAVEN [37]; 66 unique ones from OPENMALWARE [38]; and we also manually collected 170 samples from the most recent list of malicious websites reported by WEB INSPECTOR [70].

We manually inspect these 276 samples to verify their maliciousness. Besides, we collect 10,000 benign samples from the Alexa [71] top 100 websites, none of which is reported as malicious by the 56 tools provided by VIRUSTOTAL[7].

Among the total 276 malicious samples, we select 120 samples (≈40%), i.e., 15 for each attack type, as the training set. Before JS* learning, each sample is executed 10 times to get 10 traces, most of which differ from each other due to different browser context at the time of execution. These traces are converted to similar action sequences after preprocessing. Among 8 different attack types, at least 37% (for *Type V*) to at most 60% (for *Type VII*) of traces are unique. We apply JS* separately for 8 attack types based on the traces executed from their training samples. To verify the inferred DFAs, we use the remaining 156 malicious samples, most of which are recently reported by WEB INSPECTOR, together with the 10,000 benign samples as the testing data set for prediction.

The experimental environment is a PC with Intel i7 2600 3.4GHz CPU and 8GB memory. The system environment is Ubuntu 12.04 and the Firefox that we instrumented in JS* is 17.0.

[6] There is a large number of duplicated and expired malware in our collected samples.

[7] To our best knowledge, JSAND is the only open service for JS malware detection, and VIRUSTOTAL is powered by updated versions of mainstream anti-virus products.

We conduct experiments on real-world JavaScript malware to study eight popular attack types to evaluate JS*. We aim at answering the following four questions in our evaluation:

RQ1. Do our learned DFAs *correctly* and *effectively* model common and abstract behaviors for each attack type?

RQ2. Does JS* perform *efficiently* in the learning process in terms of the running time?

RQ3. Is JS* *accurate* in malware detection, compared to other research prototypes and commercial anti-virus products?

RQ4. Are these learned DFAs *useful* in the detection of emerging malicious variants and new attacks by unknown malware?

RQ1 and RQ2 examine the effectiveness and efficiency of JS*. RQ3 and RQ4 are to investigate the usefulness of the inferred DFAs.

4.5.3 JS* Learning Evaluation

The statistics of the learning process as well as the inferred DFAs are listed in Table 4.2. $\mid S \mid$ denotes the size of the states inside the inferred DFA; $\mid \Sigma \mid$ denotes the size of the alphabet of the DFA (the size of common actions); **#M-Q** refers to the number of called membership queries; **#C-Q** refers to the number of called candidate queries; **Time(s)** denotes the core time (in seconds) of learning process; **Total Time(s)** denotes the total time (in seconds) of learning process, including trace generation and replay. Generally, the results in Table 4.2 shows the capability of the L* algorithm in learning a DFA to model malicious behaviors. With only dozens of malicious and benign traces, our approach leverages the merits of the L* algorithm in learning a DFA with a limited number of traces in polynomial time.

RQ1: Correctness. The soundness of these DFAs should be examined. Due to the large value of $\mid S \mid$, it is not easy to manually check all the states in S. We validate the correctness of these DFAs from two aspects: (1) identifying the high-level semantics

Table 4.2: The Learning Results of JS*

Attack Type	$\|S\|$	$\|\Sigma\|$	#M-Q	#C-Q	Time(s)	Total Time(s)
Type I	67	29	132,084	4	1.83	945
Type II	77	11	33,585	7	0.501	634
Type III	54	11	41,459	8	4.37	902
Type IV	74	28	158,120	4	1.95	1,068
Type V	121	29	468,124	6	8.33	2,768
Type VI	50	11	34,266	4	0.551	498
Type VII	466	15	309,277	20	22.425	4,597
Type VIII	74	11	28,480	12	0.563	993

by checking their alphabets, and (2) interpreting the accepted path of DFA with hints in the descriptions of the CVE used by the attack.

First, checking the comparatively small set of common actions Σ briefly tells whether common essential behaviors of the same attack type are captured and modeled. From our observation, the set of common actions (Σ) for each attack type is reasonable and relevant to the attack type. For example, *Type I* attack can be generally divided into three steps: first, putting the shellcode in a predictable memory location; then triggering an exploitable crash (modelled by common actions like nsIAppStartup.trackStartupCrashEnd and others in the alphabet of *Type I*); at last, the shellcode will be executed to perform the attack, invoking file operations system calls (modeled by common actions nsIFile.append, etc).

Second, we check accepted traces of *Type VI* attack DFA and identify five steps: downloading malicious pdf file, executing embedded JavaScript to scan vulnerability, exploiting the vulnerability, executing payload, and actual sabotage. The detailed alphabet of each learned DFA and the explanations can be found on our website [72].

We observe that the eight inferred DFAs share some common parts, e.g., payloads executing. Generally, payloads include executing the arbitrary command, binding shell or reversed shell using TCP, etc. In Figure 4.11, we illustrate the common behaviors

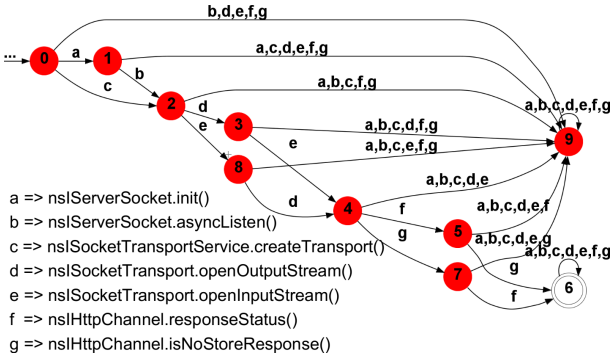


Figure 4.11: The DFA of Binding Shell Using TCP

of binding shell or reversed shell using TCP, after the exploit for each corresponding attack is done.

The unique parts of an inferred DFA model the essential attack behaviors of the corresponding attack type. In Figure 4.12, we show a fraction of *Type I* DFA. This fraction exploits the CVE-2013-1710 vulnerability. The crypto.generateCRMFRequest function in Mozilla Firefox before 23.0, Firefox ESR 17.x before 17.0.8, Thunderbird before 17.0.8, Thunderbird ESR 17.x before 17.0.8, and SeaMonkey before 2.20 allows remote attackers to execute arbitrary JavaScript code or conduct cross-site scripting (XSS) attacks via vectors related to Certificate Request Message Format (CRMF) request generation.

a	⇒ nsIDOMCrypto.generateCRMFRequest()	h	⇒ nsISocketTransport.openInputStream()
b	⇒ nsIHttpProtocolHandler.userAgent()	i	⇒ nsIInputStreamPump.init()
c	⇒ nsIDOMNavigator.userAgent()	j	⇒ nsIInputStreamPump.asyncRead()
d	⇒ nsIDOMNavigator.platform()	k	⇒ nsIDocShell.busyFlags()
e	⇒ nsIDOMNavigator.buildID()	l	⇒ nsIDocShell.currentDocumentChannel()
f	⇒ nsISocketTransportService.createTransport()	m	⇒ nsIHttpChannel.responseStatus()
g	⇒ nsISocketTransport.openOutputStream()	n	⇒ nsIHttpChannel.isNoStoreResponse()

Figure 4.12: The Partial DFA of *Type I* Attack Models the Exploit to CVE-2013-1710

Specifically, this attack invokes system call crypto.generateCRMFRequest (to enable remote attackers to execute arbitrary JavaScript code or conduct cross-site scripting (XSS) attacks via vectors related to Certificate Request Message Format (CRMF) request generation), nsIBoxObject.setProperty (To get an instance, use the boxObject property), nsIAppStartup.trackStartupCrashEnd (the last startup crashed then increment

a counter. enter safe mode if necessary), nsIScriptableUnicodeConverter. convertToByteArray (Convert a Unicode string to an array of bytes), nsIScriptableUnicodeConverter. convertToInputStream (Converts a Unicode string to an input stream. The bytes in the stream are encoded according to the charset attribute. The returned stream is non-blocking), nsICryptoHash.update (Adds an array of data to be hashed to the object), nsIFile.create (This method creates a new file or directory in the file system corresponding to the file path represented by this nsIFile. This method creates any path segments that do not already exist), nsIFile.isWritable (This method tests whether or not this nsIFile corresponds to a file or directory that may be modified by the user), nsIFile.append (nsIFile is the correct platform-agnostic way to specify a file). According to a technology report, attack targeting browser vulnerabilities can be generally divided into three steps: First, put shellcode in a predictable memory location; then trigger an exploitable crash, which will lead to controlling of EIP; at last, shellcode will be executed and download malicious executable into the user's computer, as shown in common actions, involving file operations system call.

The original trace contains 1,236 system calls and has 12 equivalent permutations with the same set of actions. However, with our JS* learning approach, the DFA in Figure 4.12 contains only 14 actions and 19 states. Thus, our inferred attack behavior models in form of DFA are concise yet accurate, without loss of the essence of attack.

A notable case is that the DFA of malicious redirecting attack *(Type VII)* has 466 states. A possible explanation is that samples of *Type VII* attack used for learning are less similar than samples of other attack types. Thus, 15 dissimilar samples infer a DFA with a small alphabetical size but a large size of states, which does not necessarily mean a bad modeling result. On the contrary, it suggests that there exist many traces from this DFA to be accepted—more possible variants of this attack type. Such explanation is backed up by the fact that malicious redirecting attacks are simple with a small set of common behaviors ($|\Sigma|$ =15), but flexible with possibly enormous variants: 986 out of totally 1,300 malicious JavaScript samples provided by VXHEAVEN are drive-by-download attacks that generally relate to *Type VII* attack.

Effectiveness. Besides manual observation, we empirically validate the usefulness of the DFAs by conducting prediction. On the testing set of 10,000 benign and 156 malicious samples, each sample is executed 10 times to get different traces. If any trace is accepted by one of the eight learned DFAs, the corresponding sample is detected as one variant of the attack that is modeled by the matched DFA. Totally, JS* correctly detects 149 out of 156 (95.51%) malicious samples and 9,957 out of 10,000 (99.57%) benign samples. Note that 43* of 1,000 refers to the distinct benign samples that are falsely reported. Actually, JS* conducts 80,000 times of acceptance check, considering 10 traces for each of 1,000 benign samples as well as 8 attack DFAs. Totally, JS* has 54 FP cases among 80,000 times of acceptance check.

The distributions of 7 FN cases (4.49%) and 54 FP cases involving 43 distinct benign samples (0.43%) are listed in column **JS* FN** and **JS* FP** of Table 4.3[8], respectively. Among 43 distinct benign samples, 11 (54-43) are falsely accepted by two DFAs, e.g., 2 benign samples are accepted by DFAs of both *Type VII* and *VIII*, which share commonality — toolkits based attacks utilize abnormal redirection iteratively to evade detection. Thus, JS* attains low overall FN ($\approx$5%) and FP ($\approx$0.5%) rate for all 8 attack types. Note that If we consider only one attack type, the FP rate becomes much lower, since only one DFA is checked against benign samples.

RQ2: Performance. The results reported in column **Time(s)** of Table 4.2 show that our approach is highly scalable in the core learning process. The required learning time is generally proportional to the state number and the alphabetical size of the learned DFA. Column **Total Time(s)** includes the time used for trace generation and replay, which is the major overheads for dynamic approaches. As reported in [57], CUJO takes averagely 500 ms to analyze a webpage in dynamic feature extraction. In JS*, it averagely takes 1 second to generate or replay one trace, for a given script snippet.

The smaller alphabetic usually leads to the less probability of an alphabetical sequence to be processed in learning. Owing to the step of preprocessing, a small alphabet can be built from the execution traces by filtering security-irrelevant system calls out and extracting common actions from traces. Alphabetical sizes ($|\Sigma|$) of the 8 learned DFAs

[8]The reason to put Total in the first row is that there is no type-based detection in JSand or the 56 tools on VirusTotal. We separately list the 8 type-specific rows to show how attack types affect detection results.

are all less than 30, and 5 out of 8 DFAs even have $|\Sigma|$ less than 16. Usually, a small value of $|\Sigma|$ leads to a small number of raised membership queries and candidate queries, e.g., DFAs with $|\Sigma| \leqslant 11$ have **#M-Q**$\leqslant 42K$ and **#C-Q**$\leqslant 12$. For these DFAs with $|\Sigma| \leqslant 11$, the core learning process can be accomplished in 5 seconds. For other types except for *Type VII*, it takes only less than 9 seconds. The most time-consuming one is for *Type VII*. Considering large values of **#M-Q** and **#C-Q**, it is fast to finish core learning in 22 seconds. It is also acceptable to finish all, including trace generation and replay, in 4,597 seconds.

Another observation is that JS* requires a large value of **#M-Q** but a small value of **#C-Q**. The explanation is that action sequences in the training set are quite different from each other in terms of length or substring. In contrast, sequences π_{m1} to π_{m6} and π_{b1} to π_{b6} in Figure 4.4 show high similarity in length or substring, with different positions of action c. These similar sequences in our running example quickly lead to a closed and consistent observation table, which makes **#M-Q**$= 622$ and **#C-Q**$= 2$. However, for these 8 real attack types, there are no such ideally similar sequences that lead to quick convergence to a closed and consistent observation table. Thus, it usually needs a large number of membership queries to reach a candidate DFA.

As the state number of each model is less than 100, the learning process generally terminates with less than or around 10,000 membership and 10 candidate queries. We also found replay is the most time-consuming part, which takes the similar time with SpiderMonkey interpreting one piece of JavaScript snippet, normally 4.5 ms.

RQ3: Tool comparison. We compared JS* with the JavaScript malware detection service JSAND 2.3.6 [73][56] and VIRUSTOTAL [55], an online malware detection service powered by 56 mainstream anti-virus products. The comparison mainly focuses on FN rate and average prediction time. We do not compare FP rate, as 1,000 benign samples are verified by the union of results from JSAND and 56 tools on VIRUSTOTAL. In another word, no FP case in benign samples for JSAND and 56 tools on VIRUSTOTAL.

Totally, JSAND[9] correctly detects 39.1% (61/156) malicious samples, and 95 FN cases

[9] As JSAND uses dynamic analysis, we submitted each sample ten times. If any submission reports that the sample is malicious or suspicious, we consider it malicious.

are not evenly distributed among the 8 types (see Table 4.3). Among the 8 attack types, JSAND yields the lowest FN rate (8.7%) for *Type VI*, and the highest FN rate (80%) for *Type I* and *Type II* attack. As JSAND is a dynamic detection tool, due to the limit of the used honeypot, it may miss samples from *Type I* and *Type II*, which are platform specific and not easy to trigger.

Table 4.3: The Prediction Results of JS*, JSAND, and VIRUSTOTAL

Type	**JS***			**JSAND**		**VIRUSTOTAL**
	FP	**FN**	**T(S)**	**FN**	**T(S)**	
Total	43* /10000	7 /156	1.36	95/156	4.8	21
Type I	4/10000	0/15	1.31	12/15	3.3	19
Type II	9/10000	1/15	1.34	12/15	5.9	21
Type III	3/10000	0/18	1.50	13/18	4.2	17
Type IV	7/10000	1/15	1.32	9/15	3.3	20
Type V	11/10000	2/20	1.33	12/20	3.4	21
Type VI	8/10000	0/23	1.32	2/23	8.3	34
Type VII	3/10000	1/23	1.26	18/23	3.4	15
Type VIII	9/10000	2/27	1.47	17/27	5.3	21

Column **VIRUSTOTAL**[10] in Table 4.3 denotes the average number of tools that detect each malicious sample, among 56 tools provided by VIRUSTOTAL. For each of 156 malicious samples, on average 21 (37.5%) of 56 tools can successfully detect it. We also observe that on average 34 (60.7%) of 56 tools can detect each of the *Type VI* samples. This observation indicates that 56 tools on VIRUSTOTAL can generally better detect *Type VI* attacks than others. This observation is consistent with the previous finding that JSAND has the lowest FN rate (8.7%) for *Type VI*. Among the 8 types, on average only 15 tools (26.8%) can detect each of *Type VII* attacks. Thus, *Type VII* attack is difficult to detect for state-of-the-art tools, due to its flexible attack behaviors and enormous variants. This point is supported by the complexity of the inferred DFA of *Type VII*.

[10] We also submitted each sample to VIRUSTOTAL five times in Dec. 2014, and the results reported by VIRUSTOTAL were consistent for different submissions.

To see the capabilities of state-of-the-art tools, in Table 4.4, we test the detection ratio of the open-source anti-virus tool CLAMAV [74] and 2014 best-reviewed anti-virus products [75]: AVG, AVAST!, BITDEFENDER, F-SECURE, GDATA, KASPERSKY, MCAFEE, PAN-DA, SYMANTEC, and TRENDMICRO. On the testing set of 156 malicious samples, the best tool AVAST! achieves a detection ratio of 81.41%. Other tools perform even worse. We manually inspect FN cases for JS* and other tools. One sample that belongs to *Type VII* is missed by both JS* and VIRUSTOTAL, as it targets at mobile platform and fails to launch the attack in our testing environment.

Table 4.4: Detection Ratio of JS* and Other Tools on 156 Malicious Samples

Tool	Detection %	Tool	Detection %
JS*	95.51%	MCAFEE	57.05%
AVAST!	81.41%	JSAND	39.10%
GDATA	73.72%	TREND	30.77%
AVG	73.08%	SYMANTEC	28.21%
BITDEFENDER	71.15%	CLAMAV	12.82%
F-SECURE	69.87%	PANDA	1.92%
KASPERSKY	67.95%		

According to the JSAND's report, we calculate the prediction time by deducting the analysis starting time from the report generation time. Averagely, it takes 4.7 seconds for JSAND to finish the execution and prediction of one sample (see column **JSAND T(s)** in Table 4.3). In contrast, FN of JS* (8.2%) is much lower than that of JSAND (76.7%), which is a significant improvement with 100 (69%) more cases detected. In contrast, JS* takes averagely 1.36 seconds to execute the tested sample and check the trace with 8 learned DFAs. Thus, the prediction time was reduced by 71% in JS*. For prediction time of the 56 tools, VIRUSTOTAL runs them in parallel and sets a timer (1 minute) to prevent no response. As results from different tools are dynamically added to the result page, according to our observations, most tools can finish the prediction in 5-10 seconds.

RQ4: Detecting variants. The prediction results show the capability of JS* in detecting malicious variants of the same attack type, as the 120 training samples used for learning

share quite low textual similarity with the 156 testing samples. We use code clone detection tool CloneMiner [76] and fail to detect sample pairs that have a file-level textual similarity above 30% (due to different exploits and obfuscation). Thus, owing to the nature of dynamic analysis of JS*, syntactic obfuscation poses no challenge to JS*.

4.6 Related Work

Inferring behavior model by L*. Both off-line learning and online learning can be applied to infer of behavior model. Passive off-line inference requires the availability of sufficient system call traces (or action sequences), as an abundance of known traces makes the online teacher optional. Existing off-line learning algorithm may adopt some heuristic best-effort inference algorithms in polynomial time [77]. Thus, using L* for off-line learning with some heuristic algorithms to substitute online teacher can work, but it may suffer from the incompleteness and impreciseness of the inferred model [78].

Several studies adopt the L* to infer behavior, but not attack behaviors. Chia *et al.* [78] proposed to infer botnet command and control protocols from the sequence of messages sent over the network by using the L* algorithm. Instead of inferring a default Moore machine in the classic L* algorithm, the authors model the attacks in botnet as a Mealy machine that has both input symbol and output symbol. Every Mealy machine has a corresponding counterpart of the Moore machine, but the Mealy machine is with fewer states. Chia *et al.* [79] also present an approach to infer an abstract model of the analyzed application in form of DFA and then apply symbolic execution for bounded state-space exploration by virtue of the guidance provided by the inferred DFA. Xiao *et al.* [8] apply L* on method call sequences in the randomly generated testing code to model behaviors as stateful typestate, which is suitable for data-rich programs, i.e., stack, piped stream, etc.

Answering membership queries depends on domain knowledge, e.g., in [8] for behavior learning, if an exception is thrown for the generated testing code, the corresponding method sequence is not accepted; in [79] for protocol inference, if a message exchange

trace violates the protocol, the given trace is not accepted. In our study, if malicious results are produced or defense rules are violated, the tested action sequence is accepted. Candidate queries can be answered in three ways. First, the expected DFA to be learned is available, e.g., the existing protocol in [79]. Second, random sampling is used to generate traces to test the equivalence between a candidate DFA and the expected DFA [8]. Last, a model checker is applied to verify the equivalence between a candidate DFA with the expected one based on some properties [6].

JavaScript malware detection. The existing studies on detection of JavaScript malware mainly can be categorized into the static analysis, dynamic analysis, or hybrid analysis of static and dynamic approaches. JSAND [56] extracted features from four aspects (redirection, de-obfuscation, environmental context, and exploitation) via dynamic analysis, and used Naïve Bayes to detect JavaScript malware. Canali *et al.* [5] propose to perform a large-scale analysis to identify the malicious web pages by applying a fast and reliable filter PROPHILER. PROPHILER adopts static analysis techniques that can quickly filter out benign pages and forward to the costly analysis tools only the suspicious pages that are highly probably malicious.

Curtsinger *et al.* [43] presented ZOZZLE, a tool that predicates the benignity or maliciousness of JavaScript code by using features associated with AST hierarchy information. REVOLVER [80] also heavily relies on static analysis to build the ASTs and to compute the similarity among ASTs. CUJO [57] uses hybrid analysis to on-the-fly extract dynamic and static features from program information and execution traces of JavaScript programs, respectively. All extracted features are processed by *q-grams* for SVM based classification.

Paper [81] makes the first attempt to protect the Android ecosystem by modeling and predicting the spread of Android malware between markets. To this end, they study the social behaviors that affect the spread of malware, model these spread behaviors with multiple epidemic models, and predict the infection time and order among markets for well-known malware families. Paper [82] proposes a unified framework that systematically integrates multiple views of apps for performing comprehensive malware detection and malicious code localisation. Based on the modularized attack features,

paper [83] audits the AMTs at runtime by applying the dynamic code generation and loading techniques to produce malware. Paper [84] systematically investigates numerous use cases of collaborative security by covering six types of security systems. Aspects of these systems are thoroughly studied, including their technologies, standards, frameworks, strengths and weaknesses. The authors then present a comprehensive study with respect to their analysis target, timeliness of analysis, architecture, network infrastructure, initiative, shared information and inter operability. Paper [85] first proposes a meta model for Android malware to capture the common attack features and evasion features in the malware. Based on this model, they develop a framework, MYSTIQUE, to automatically generate malware covering four attack features and two evasion features, by adopting the software product line engineering approach.

Paper [86] proposes a hardware assisted architecture to perform online malware detection with two phases. In the offline phase, they learn the attack model of malware in the form of Deterministic Finite Automaton (DFA). During the runtime phase, they implement a DFA-based detection approach in hardware to check whether a program's execution contains the malicious behavior specified in the DFA. Paper [87] proposes the frequency-centric model for feature construction using system call patterns of known malware and benign samples. They then develop a machine learning approach (using multilayer perceptron) in FPGA to train classifier using these features. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction.

These studies focus on the detection of general JavaScript malware. Other existing studies rely on dynamic analysis to detect specific attacks [88][89][90][91][92]. Compared with these tools, JS* can model attack behaviors. It does not require a large-size training set and can be generally applicable to attacks with explicit browser-level system calls.

4.7 Conclusion

In this work, we propose an approach, JS*, for detecting malicious JavaScript via attack behavior modeling. JS* is based on the L* algorithm and the learned model is in the

form of DFA. Our key contribution is to combine *data dependency analysis*, *defense rules* and *replay mechanism* to implement an online teacher for detection and classification of malicious JavaScript. We evaluate JS* using eight popular types of attacks. The experimental results demonstrate the scalability and effectiveness of our approach as well as the usefulness of the inferred DFAs.

5

Vulnerability Detection via Data-Driven Seed Generation

In the previous two chapters, we detect JavaScript malware which exploits vulnerabilities of browsers. The root cause of web-based attacks is the various vulnerabilities in browsers. Therefore, we also made our efforts in detecting vulnerabilities in browsers. The following two chapters first generate high-quality seeds for fuzzing and then advance fuzzing towards more intelligent.

5.1 Introduction

Fuzzing or fuzz testing is an automated software testing technique to feed a large amount of invalid or unexpected test inputs to a target program in the hope of triggering unintended program behaviors, e.g., assertion failures, crashes, or hangs. Since its introduction in the early 1990s [9], fuzzing has become one of the most effective techniques for finding bugs or vulnerabilities in real-world programs. It has been successfully applied to test various applications, ranging from rendering engines and image processors to compilers and interpreters.

A fuzzer can be classified as generation-based (e.g., [30–32, 93]) or mutation-based (e.g., [94–97]), depending on whether test inputs are generated by the knowledge of the input format or grammar or by modifying the well-formed test inputs. A fuzzer can also be classified as a whitebox (e.g., [28, 29]), greybox (e.g., [95, 97]) or blackbox (e.g., [9, 98]), depending on the degree of leveraging a target program's internal structure, which reflects the tradeoffs between effectiveness and efficiency. In this thesis, we focus on mutation-based greybox fuzzing.

In this work, we propose a novel data-driven seed generation approach, named Skyfire. It leverages the vast amount of samples (i.e., corpus) to automatically extract the knowledge of grammar and semantic rules, and utilizes such knowledge to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. In that sense, Skyfire is orthogonal to mutation-based fuzzing approaches, providing high-quality seed inputs for them and improving their efficiency and effectiveness for programs that process highly-structured inputs. Besides, Skyfire advances the existing generation-based fuzzing approaches, i.e., carrying the fuzzing exploration beyond the semantic checking stage to reach the application execution stage to find deep bugs without any manual specification tasks.

Basically, Skyfire takes as inputs a *corpus* and a *grammar*, and generates seed inputs in two steps, as shown in Figure 5.1. The first step parses the collected samples (i.e., the corpus) based on the grammar into ASTs, and learns a Probabilistic Context-Sensitive Grammar (PCSG for short), which specifies both syntax features and semantic rules.

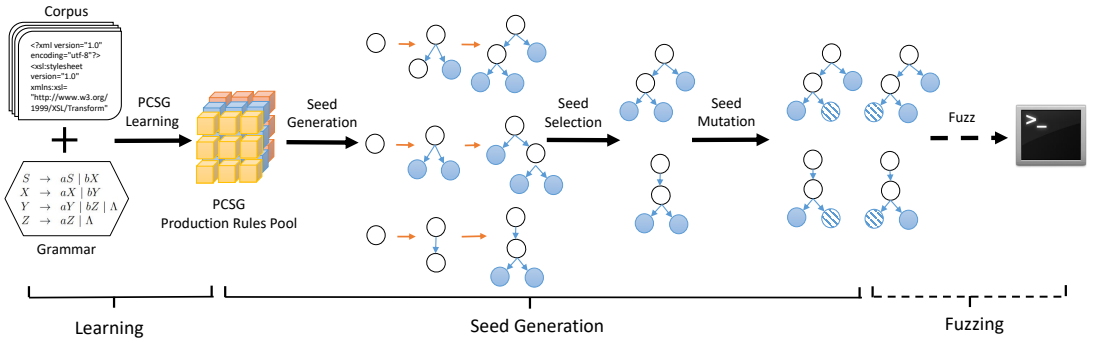


Figure 5.1: The Overview of Skyfire

Different from a context-free grammar that is widely used in generation-based fuzzing, each production rule in PCSG is associated with the *context* where the production rule can be applied as well a the *probability* that the production rule is applied under the context.

In the second step, Skyfire first generates seed inputs by iteratively selecting and applying a production rule, satisfying the context, on a non-terminal symbol until there is no non-terminal symbol in the resulting string. During this process, we prefer low-probability production rules to high-probability production rules for producing uncommon inputs with diverse grammar structures. Skyfire then conducts seed selection to filter out the seeds that have the same code coverage to reduce redundancy. Finally, Skyfire mutates the remaining seed inputs by randomly replacing a leaf-level node in the AST with the same type of nodes based on the production rules, which introduces semantic changes in a diverse way while maintaining grammar structures.

To evaluate the effectiveness and generality of our approach, we collected the samples of XSL and XML. Then, we fed the collected samples and the inputs generated by Skyfire as seeds to the AFL [18] to fuzz several open-source XSLT and XML engines (i.e., Sablotron, libxslt, and libxml2). Evaluation results have indicated that i) Skyfire can generate well-distributed inputs, and hence effectively improve the code coverage of fuzzers (i.e., 20% for line coverage and 15% for function coverage on average); and ii) Skyfire can significantly improve the capability of fuzzers to find bugs. We found 19 new memory corruption bugs (among which we discovered 16 new vulnerabilities and received 33.5k USD bug bounty rewards), and 32 new denials of service bugs (including stack exhaustion, NULL pointer dereference, and assertion failure).

5.2 Approach Overview

In this section, we will first introduce the target programs that to be fuzzed, which process highly-structured inputs, and the challenges they throw up to traditional fuzzing approach. Then follows with the overview of our approach.

Gathering an initial corpus of inputs is a desired step in a majority of cases. It makes it possible to reach some code paths and programs states immediately after starting the fuzzing. In most cases, the input may contain complex data structures which would be difficult or impossible to generate organically using just code coverage information, e.g. magic values, correct headers, compression trees etc. Even with the same inputs could be constructed during fuzzing with an empty seed, having them right at the beginning saves a lot of CPU time. Corpora containing files in specific formats may be frequently reused to fuzz various software projects which handle them. For example, XML sample corpora can be reused to fuzz various XML engines.

5.2.1 Target Programs

The compact and unstructured inputs, such as images, shell commands, are adequately fuzzed by coverage-based fuzzing approach. The mutation strategies such as bit flipping, byte flipping and so on are proved to be awfully successful for target programs like these. However, as the program becomes more complex and processes more structured inputs, mutation without knowing the structure of input is getting inefficient. For example, we tried to use AFL to fuzz XML and the result turns out to be frustrating. To meet the challenges structured input pose to traditional fuzzers, researchers normally tend to apply context-free grammar.

Definition 5.1. A context-free grammar (CFG) is a 4-tuple $G_{cf} = (N, \Sigma, R, s)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,

- R is a finite set of production rules of the form $\alpha \rightarrow \beta_1\beta_2...\beta_n$, where $\alpha \in N, n \geq 1$, and $\beta_i \in (N \cup \Sigma)$ for $i = 1...n$,
- and $s \in N$ is a distinguished start symbol.

Example 5.1. *Figure 5.2 shows the context-free grammar of the XSL language, which is a representative example of highly-structured input. As a running example, we choose a very simple structured input. Other languages are normally more complex than XSL. The XSL context-free grammar starts with a document symbol, and a document can be extended into a prolog plus mise and element. An element can have one or more attributes and each attribute has an attribute name and an attribute value. A complete CFG of XSL can be found at [99].*

N = { *document, prolog, content, element, reference, attribute, chardata, misc, entityRef, name, ...*}
Σ = { *comment, cdata, charRef, string, text, sea_ws, pi, ...*}
R = { *document* → *prolog? misc∗ element misc∗,*
prolog → **'<?xml'** *attribute∗* **'?>'**,
element → **'<'** *name attribute∗* **'/>'** | **'<'** *name attribute∗* **'>'** *content* **'</'** *name* **'>'**,
attribute → *name = string,*
content → *chardata?* ((*element* | *reference* | *cdata* | *pi* | *comment*) *chardata?*)∗,
reference → *entityRef* | *charRef,*
chardata → *text* | *sea_ws,*
misc → *comment* | *pi* | *sea_ws,*
... }
s = document

Figure 5.2: Part of the Context-Free Grammar of XSL

Moreover, structured inputs also have to follow semantic rules, which make an input limited and meaningful. The search space of input without any semantic restrictions are infinite. For example, the element name can be the combination of any length of any characters.

Example 5.2. *All XSL language engines will check whether an attribute can be applied by the certain element. If an attribute cannot be applied on the certain element is given to this element, the parser will prompt an "unexpected attribute name" notice. Even worse, some XSL engine will abort the further execution.*

5.2.2 Overview of Skyfire

By fuzzing exploration breadth, we refer to how much functionalities are reached, and by exploration depth, we mean how through a function is tested. To find more bugs, we pursue both breadth and depth. To reach deep bugs, our samples need to be valid or semi-valid, thus it can pass the syntax checking stage and reach the execution stage.

The test case also needs to be diverse. It is easy to achieve that several correct samples can reach separate functions of target programs. However, it is difficult to cover all or most of the functions of target programs by limited samples and limited execution time. Therefore, we need to make sure our samples are diverse.

The less one function is tested, the more likely it contains bugs undiscovered. Therefore, we need to pay special attention to those uncommon seeds. For example, some statements are rare and only occurs in a few samples. They may indicate some functions that are less reached and interesting to fuzz.

Directed by these three principles, we designed a fuzzing approach as shown in Figure 5.1. The inputs to our approach are grammar and a corpus of a given grammar. It first parses samples in corpus into ASTs and learns a PCSG, which later serve as the model to generate more new samples of a given grammar and can be used for fuzzing.

Mere grammar only contains syntax information. For example, it can tell an element includes an element name and several attributes, but it fails to tell what can be the value of an element name, attribute name and even attribute value. According to XML grammar, an element name can be an arbitrary combination of letters, which is a large value space. The meaningful element name is some specific strings, which cannot be expressed with grammar.

In this work, we use PCSG to model information of semantic of an input. When we parse existing samples, we can get a collection of element names in the corpus. We also can get a collection of production rules. Once we attach a context to each production rules, the semantic of a given input is better modeled. Further, attach a probability to production rules can help us to control the generation process to generate more uncommon samples or more valid samples. For example, when we choose production rules

with higher probability, we get more valid samples. While when we choose production rules with less probability, we can generate more uncommon samples.

The generation step is similar to the normal left-most derivation generation step using context-free grammar. A derivation of a string for grammar is a sequence of production rule applications that transform the start symbol into the string. The left-most derivation is the one in which we always expand the left-most non-terminal symbols. The right-most derivation is the way we always expand the right-most non-terminals. We here choose the left-most derivation since their results are the same. When choosing production rules from the pool of production rules, we have some heuristic rules to apply.

In the following two sections, we will first elaborate how to learn a PCSG to describe both syntax features and semantic rules (Section 5.3), and then we will explain how to generate seed inputs based on the learned PCSG (Section 5.4).

5.3 PCSG Learning

This section will introduce the probabilistic context-sensitive grammar (PCSG) and then how to learn a PCSG.

5.3.1 PCSG

Formal constraints not captured by the grammar are considered to be part of the semantics of the language. The difference between implementation and specification of semantics is complex and easy to introduce bugs.

Example 5.3. *Figure 5.3a is a sample of the XML format. Its concrete syntax tree is shown in Figure 5.3b.*

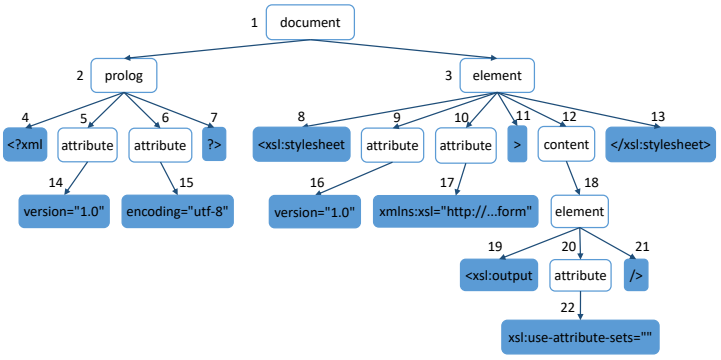
- *Nodes 5 and 6 in Figure 5.3b can be a* $version$ *or* $encoding$ *respectively. This is because* $prolog$ *only has these two kinds of* $attributes$*. If other* $attribute$

names are given to nodes 5 or 6, an "XML declaration not well-formed" will be prompted.

- *The node 3 can only be* $xsl : stylesheet$ *or* $xsl : transform$ *element, since according to the semantic rules, the "the root element that declares the document to be an XSL style sheet is* $xsl : stylesheet$ *or* $xsl : transform$*".*
- *The two* $attribute$ *nodes 9 and 10 can only be a* $version$ *or* $xmlns : xsl$ *since their parent is an* $element$ *node of type* $xsl : stylesheet$*. Otherwise, the error message "unexpected attribute {...}" will be reported.*
- *The* $attribute$ *node 10 can only be set to* $xmlns : xsl = "http://www.w3.org/1999/XSL/Transform"$ *once the* $attribute$ *node 9 is set to* $version = "1.0"$ *since its parent is an* $element$ *node of type* $xsl : stylesheet$ *and these two attributes are mandatory. Otherwise, the error message "required attribute {...} is missing" is reported.*

```
<?xml version="1.0"
    encoding="utf-8"?>

<xsl:stylesheet version
    ="1.0" xmlns:xsl=
    "http://www.w3.org
    /1999/XSL/Transform">
    <xsl:output xsl:use-
    attribute-sets=""/>
</xsl:stylesheet>
```



(a) A Sample XSL File (b) The AST of the Sample XSL File

Figure 5.3: A Running Example: A XSL File and Its Corresponding AST

As shown in Example 5.3, semantics rules are related to whether the certain value can be applied to a certain position. The position information can be expressed as the context of a node, its parent, sibling, grandparent and so on.

In Table 5.1, we listed some representative semantic rules of xsl input. We give the error messages of violating each semantic rules and the corresponding context of nodes. For

example, the *prolog* element is needed to be well defined, otherwise, the error message of "XML declaration not well-formed" will be thrown. This semantic rules can be modeled by parent context.

Table 5.1: Examples of Semantic Rules

#	Error Messages of Violating Semantic Rules	Context
1.	XML declaration not well-formed	parent
2.	The root element that declares the document to be an XSL style sheet is xsl:stylesheet or xsl:transform	parent and first sibling
3.	Unexpected attribute {...}	first sibling
4.	Unbound prefix	first sibling
5.	XSL element xsl:stylesheet can only contain XSL elements	great-grandparent
6.	Required attribute {...} is missing	first sibling and all mandatory attributes
7.	Duplicate attribute	all siblings

Instead of capturing all context information, we define the context in an efficient way that recording lest information and still can model most semantic rules. For example, the first five semantic rules in Table 5.1 can be expressed using a context includes a node's parent, grandparent, great-grandparent and first sibling. This context information is very easy to collect during learning and easy to check during generation. Though the last two semantic rules in Table 5.1 cannot be captured by the context we adopted, we do not need to fulfill all semantic rules. We only need to fulfill most semantic rules, particularly those semantic rules that checked in the beginning. To capture all of the semantic rules, we need to check too much information.

To manipulate the process of generation, we need to attach a probability to production rules, which also includes the probability of each value.

Definition 5.2. Similar to CFG, a context-sensitive grammar (CSG) is a 4-tuple $G_{cs} = (N, \Sigma, R, s)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,

- R is a finite set of context-aware production rules of the form $[c]\alpha \rightarrow \beta_1\beta_2...\beta_n$, where c is the context in the form of $\langle$type of α's great-grandparent, type of α's grandparent, type of α's parent, value of α's first sibling or type of α's first sibling if the value is null$\rangle$, $\alpha \in N, n \geq 1$, and $\beta_i \in (N \cup \Sigma)$ for $i = 1...n$,
- and $s \in N$ is a distinguished start symbol.

Definition 5.3. A PCSG is a tuple $G_p = (G_{cs}, q)$, where

- $G_{cs} = (N, \Sigma, R, s)$ is a CSG,
- and $q : R \rightarrow \mathbb{R}^+$ assigns each production rule with a probability so that $\forall\, \alpha \in N$:

$$\sum_{[c]\alpha \rightarrow \beta_1\beta_2...\beta_n \in R} q([c]\alpha \rightarrow \beta_1\beta_2...\beta_n) = 1.$$

Here $\mathbb{R}^+$ denotes the set of non-negative real numbers. Definition 5.3 ensures that for a given non-terminal symbol, the probability of the applicable production rules for all contexts sums to one. In summary, by integrating both context and probability of a production rule, a PCSG is more expressive than a CFG or a CSG, which summarizes the knowledge of the samples in a single data structure.

5.3.2 Learning PCSG

The input to the learning step is a corpus and grammar. The output is the PCSG. Then the PCSG serves as the input of generation stage. Notice that the corpus will not be used in the generation stage, all information needed is captured in the PCSG.

According to the grammar, we parse the given inputs into concrete syntax trees. Then we traverse the syntax trees and extract every parent-children pair, which corresponds to a production rule. During this procedure, we collect the context information and count the occurrence of production rules which later used to calculate the probability.

Example 5.4. *In Figure 5.3b, node 1 is the parent of nodes 2 and 3, which forms a parent-children pair. This pair corresponds to the application of the production rule*

$document \rightarrow prolog\ element$ under the context $\langle null, null, null, null\rangle$ as the $document$ is the root node. Nodes 4, 5, 6, and 7 are the children of node 2, which corresponds to the application of the production rule $prolog \rightarrow$ <?$xml\ attribute\ attribute$? >under the context $\langle null, null, document, null\rangle$. Besides, node 9 and node 16 correspond to the application of the production rule $attribute \rightarrow version =$ "1.0" under the context $\langle null, document, element,$ <$xsl{:}stylesheet\rangle$, where <$xsl{:}stylesheet$ is the value of node 9's the first sibling.

We calculate the probability through the maximum likelihood estimation,

$$q([c]\alpha \rightarrow \beta_1\beta_2...\beta_n) = \frac{count([c]\alpha \rightarrow \beta_1\beta_2...\beta_n)}{count(\alpha)}$$

where $count([c]\alpha \rightarrow \beta_1\beta_2...\beta_n)$ is the number of times that the rule $\alpha \rightarrow \beta_1\beta_2...\beta_n$ is seen in all ASTs under the context c, and $count(\alpha)$ is the number of times that α is seen in all ASTs.

Example 5.5. *Table 5.2 shows part of the learned production rules of XSL. We can see that, there are only two production rules (1 and 2) whose left-side is a $document$. Their context can only be $\langle null, null, null, null\rangle$, which is consistent with the fact that the $document$ is the root node; and their probability is respectively 0.82 and 0.18. Besides, production rules 13 and 14 capture the first semantic rule in Table 5.1, while production rules 6, 7, and 8 reflect the second semantic rule in Table 5.1. It is worth noting that the probability of the production rules for the $element$, $attribute$, and $content$ is very low because there are many different types of $elements$, $attributes$, and $contents$ in the corpus.*

We store learned production rules in a pool. In this work we use a relatively simple learning approach, later we may try out some more powerful learning approach, such as machine learning based.

Table 5.2: Part of the Learned Production Rules of XSL

ID	Context	Production Rule		Probability
1	<null, null, null, null>	document	→ prolog element	0.82
2			→ element	0.18
3	<null, null, document, null>	prolog	→ <?xml attribute attribute?>	0.646
4			→ <?xml attribute?>	0.347
5			→ ...	
6	<null, null, document, prolog>	element	→ <xsl:stylesheet attribute >content</xsl:stylesheet>	0.0034
7			→ <xsl:transform attribute>content</xsl:transform>	0.0001
8			→ ...	
9	<document, element, content, element>	element	→ <xsl:template attribute>content</xsl:template>	0.0282
10			→ <xsl:variable attribute>content</xsl:variable>	0.0035
11			→ <xsl:include attribute/>	0.0026
12			→ ...	
13	<null, document, prolog, <?xml >	attribute	→ version="1.0"	0.0056
14			→ encoding="utf-8"	0.0021
15			→ ...	
16	<null,document,element,<xsl:stylesheet>	attribute	→ xmlns:xsl="http://www.w3.org/1999/XSL/Transform"	0.0068
17			→ version="1.0"	0.0052
18			→ ...	
19	<element, content, element, <xsl:text>	content	→ chardata	0.0750
20			→ reference	0.0073
21			→ ...	

5.4 Seed Generation

In this section, we introduce how to generate the seeds using left-most derivation algorithm. A derivation of a string for grammar is a sequence of production rule applications that transform the start symbol into the string. The left-most derivation is the one in which we always expand the left-most non-terminal symbols. The right-most derivation is the way we always expand the right-most non-terminals. We here to choose the left-most derivation since their results are the same. When choosing production rules from the pool of production rules, we have some heuristic rules to apply.

Definition 5.4 gives how to generate a seed from a PCSG. Concretely, we first set t equals the start symbol of grammar; then we expand the left-most non-terminal symbol in t with a production rule whose context matches current generation context. Repeat this procedure until no non-terminal symbols left in t. The key to success of fuzing is how to select production rules.

Definition 5.4. Given a PCSG $G_p = ((N, \Sigma, R, s), q)$, a left-most derivation is a sequence of symbols $t_0, ..., t_n$, where

- $t_0 = s$, i.e., t_0 contains only the start symbol,
- $t_n \in \Sigma^*$, i.e., t_n is made up of terminal symbols, and Σ^* denotes the set of all possible strings made up of sequences of words taken from Σ,
- t_i for $i = 1, ..., n$ is derived from t_{i-1} by replacing the left-most non-terminal symbol α (with context c) in t_{i-1} with certain β where $[c]\alpha \rightarrow \beta$ is a production rule in R.

Without attached probability and context of production rules, the left-most derivation can only randomly select a production rule since there is no information to assist. However, there are several defects to this approach. First, if we randomly select each time, since each production rule can introduce several non-terminal symbols, the generation will be infinite. Second, the samples can be unnecessarily complex and large since long derivation sequence.

To avoid such problems, we use heuristic rules to control the generation. As shown in Algorithm 5, we adopt four heuristic rules into the left-most derivation.

Heuristic 1: Favor low-probability production rules. Given all production rules which match current context, we separate them into two groups according to probability. One group with higher probability, R_l^H, and one with lower probability R_l^L. This is implemented by sort production rules according to probability and then cut it into two parts from the middle. In (Line 10–13), we choose a production rule which has lower probability with a probability of 90%, and with a probability of 10%, we choose production rules from the higher probability group.

Heuristic 2: Favor low-frequency production rules, and restrict the application number of the same production rule. Filtered by heuristic 1, we still get a lot of production rules to choose. To generate diverse samples, we record the application times of each production rules. If a selected production rule has been applied many times before, we choose another time to find one which has been selected less before.

Heuristic 3: Favor low-complexity production rules. This rule is easy to understand. We measure the complexity of a production rule by its number of non-terminal symbols.

Algorithm 5 Generating Seed Inputs from a PCSG

Input: the PCSG $G_p = ((N, \Sigma, R, s), q)$
Output: the set of seeds generated T

```
1:  T := ∅
2:  repeat
3:      t := s      // set the seed input to the start symbol
4:      c := null   // set the context to null
5:      num := 0  // set the times of applying rules to 0
6:      repeat
7:          l := the left-most non-terminal symbol in t
8:          update c according to l
9:          R_l := the set of rules in R whose left-side is l given c
10:         if random() < 0.9 then
11:             heuristically choose a less-frequently applied and less-complexity rule r from low-probability rules in R_l
12:         else
13:             heuristically choose a less-frequently applied and less-complexity rule r from high-probability rules in R_l
14:         end if
15:         replace l in t with the right side of r
16:         num := num + 1
17:     until there is no non-terminal symbol in t, or num == 200
18:     T := T ∪ {t}
19: until time budget is reached, or enough seed inputs are generated
```

If one production rule contains five non-terminal symbols, we say its complexity is five. The more non-terminal symbols one production contains, larger is the finally generated sample. To restrict the size of the final sample, we favor low-complexity production rules.

Heuristic 4: Restrict the total number of rule applications. Under the certain scenario, the derivation procedure cannot stop even with the application of above-mentioned heuristic rules and the expansion times reaches a threshold. In such a case, we directly abort the test case and start to generate the next sample.

Example 5.6. *Figure 5.4 shows an example of our input generation by heuristically applying left-most derivation to XSL.* t_i *represents the result string after applying a production rule.* t_0 *is initialized with the start symbol* $document$*. The generation is conducted until no more non-terminal symbol is left, which is successfully completed in 10 rule applications.* s_1 *is derived from* s_0 *by applying production rule* $document->prologelement$*.* s_2 *is derived from* s_1 *by applying production rule* $prolog-><?xml\ attribute\ attribute?>$ *on the left-most non-terminal symbol* $prolog$*.* s_3 *is derived*

from s_2 *by applying production rule* $attribute- > version =' 1.0'$ *on the left-most non-terminal symbol* $attribute$. s_4 *is derived from* s_3 *by applying production rule* $attribute- > encoding =' utf -8'$ *on the left-most non-terminal symbol* $attribute$. s_5 *is derived from* s_4 *by applying the production rule* $element- >< xsl : stylesheet\ attribute\ attribute > content < /xsl : stylesheet >$ *on left-most non-terminal symbol* $attribute$. s_6 *is derived from* s_5 *by applying production rule* $attribute- > version =' 1.0'$ *on the left-most non-terminal symbol* $attribute$. s_7 *is derived from* s_6 *by applying production rule* $attribute- > xmlns : xsl = "http : //www.w3.org/1999/XSL/Transform"$ *on the left-most non-terminal symbol* $attribute$. s_8 *is derived from* s_7 *by applying production rule* $content- > element$ *on the left-most non-terminal symbol* $content$. s_9 *is derived from* s_8 *by applying production rule* $element- >< xsl : outputattribute/ >< /xsl : stylesheet >$ *on the left-most non-terminal symbol* $element$. s_10 *is derived from* s_9 *by applying production rule* $attribute- > xsl : use - attribute - sets = ""$ *on the left-most non-terminal symbol* $attribute$. *In fact, this simple input is the same one to Figure 5.3; and it crashed the XSLT engine Sablotron 1.0.3 [100]. Specifically, it triggers a buffer underflow that can be exploited to code execution.*

t_0 =document
t_1 =prolog element
t_2 =<?xml attribute attribute?> element
t_3 =<?xml version="1.0" attribute?> element
t_4 =<?xml version="1.0" encoding= "utf-8"?> element
t_5 =<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet attribute attribute>content</xsl:stylesheet>
t_6 =<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet version="1.0" attribute>content</xsl:stylesheet>
t_7 =<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">content</xsl:stylesheet>
t_8 =<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">element</xsl:stylesheet>
t_9 =<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:output attribute/></xsl:stylesheet>
t_{10}=<?xml version="1.0" encoding= "utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:output xsl:use-attribute-sets=""/></xsl:stylesheet>

Figure 5.4: An Example of the Left-Most Derivation Seed Generation

5.5 Implementation and Evaluation

Skyfire is implemented in Java and the source code is available on Github (https://github.com/zhunki/skyfire). In particular, Heritrix [39] is used to crawl samples of XML and XSL. ANTLR [101] is used to automatically generate the lexer, parser and visitor code for manipulating XML and XSL samples.

5.5.1 Evaluation Setup

The grammar of XSL and XML are available on ANTLR community [102]. XML and XSL share the same grammar but different semantics. Therefore, XSL samples can serve as XML samples while XML samples cannot be used as XSL samples. XSL (eXtensible Stylesheet Language) is a styling language for XML. XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library. There are over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, and more.

XML and XSL samples all can be found on the Internet. We crawled about 21m web pages and download XML and XSL files from those web pages. As shown in Table 5.3, we collected 18,686 XSL files and 671 of them are unique. Using Skyfire, we generate 5,017 distinct samples as seeds of fuzzing. Similarly, we crawled 19,324 XML files and 732 of them are unique. Later Skyfire generates 5,923 distinct samples to fuzz.

Table 5.3: Statistics of Samples Crawled and Generated

Language	XSL	XML
# of Unique Samples Crawled	18,686	19,324
# of Distinct Samples Crawled	671	732
# of Distinct Seeds Generated by Skyfire	5,017	5,923

Our target programs include two open-source XSL engines, Sablotron 1.0.3 [100] and libxslt 1.1.29 [103], and one XML engine libxml2 2.9.2, 2.9.3, and 2.9.4 [104]. They are all open-source. During fuzzing, we used AddressSanitizer [105] to find memory corruption bugs.

- Sablotron is an efficient, compact, and portable XSL toolkit, which implements XSLT 1.0, DOM Level 2, and XPath 1.0. It is an open-source project and subject to the Mozilla Public License or the General Public License and uses Expat by James Clark as the XML parser. It is adopted into Adobe PDF Reader and Adobe Acrobat.

- libxslt is the XSLT C library developed for the GNOME project, which is based on libxml2. libxslt is used in a variety of products such as Chrome browser, Safari browser, and PHP 5.

- libxml2 is the XML C parser and toolkit developed for the GNOME project, available open-source under the MIT License. It is also widely used outside GNOME due to its excellent portability and existence of various bindings. It is used in Linux, Apple iOS/OS X, and tvOS. The Google Patch Reward Program lists libxml2 as a core infrastructure data parser [106].

With seeds generated using Skyfire, we feed them to AFL for fuzzing. AFL is robust and easy to use. AFL is well-known for finding thousands of high-profile vulnerabilities. We have four configurations for evaluation. The first one is feeding crawled seeds directly to target programs; The second is to feed crawled samples to AFL and fuzz for a span of time; The third is to use seeds generated by Skyfire to test target programs; The fouth setup is to feed seeds generated by Skyfire to AFL and fuzz for a span of time. These four setups are named as *Crawl*, *Crawl+AFL*, *Skyfire*, and *Skyfire+AFL*, as shown in Table 5.4 and 5.5.

Section 5.5.2 shows the bug finding capability of Skyfire and Section 5.5.3 gives the coverage information of different fuzzing setups. Section 5.5.4 investigates the effectiveness of our proposed heuristic rules. Section 5.5.5 shows the performance of Skyfire.

We fuzzed XML and XSL engines for a time span of 15 months. Our evaluation environment is a workstation with 8 CPUs and 3GB main memory. The operating system is Ubuntu 14.04.

5.5.2 Vulnerabilities and Bugs Discovered

Table 5.4 shows the number of bugs we found in Sablotron, libxslt, and libxml2. Since the initial samples did not trigger any crashes, we omit the column *Crawl*. The first column listed the type of bugs we found in three open-source projects. The first type is previously undiscovered memory corruptions vulnerabilities, while the second type

is memory corruption vulnerabilities we found and during our report, they also found by other researchers before. The third type is the new denial of service bugs we found which are normally considered as hard to exploit.

Table 5.4: Unique Bugs Found in Different XSLT and XML Engines

Unique Bugs (#)	XSL						XML		
	Sablotron 1.0.3			libxslt 1.1.29			libxml2 2.9.2/2.9.3/2.9.4		
	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl+AFL	Skyfire	Skyfire+AFL
Memory Corruptions (New)	1	5	8	0	0	0	6	3	11[¶]
Memory Corruptions (Known)	0	1	2[†]	0	0	0	4	0	4[‡]
Denial of Service (New)	8	7	15	0	2	3	2	1	3[⊕]
Total	9	13	25	0	2	3	12	4	18

CVE-2016-6969, CVE-2016-6978, CVE-2017-2949, CVE-2017-2970, and one pending report.

[¶] CVE-2015-7115, CVE-2015-7116, CVE-2016-1835, CVE-2016-1836, CVE-2016-1837, CVE-2016-1762, and CVE-2016-4447; pending reports include GNOME Bugzilla 766956, 769185, 769186, and 769187. After Feb 9, 2017, it was communicated via GNOME Bugzilla by Apple libxml2 maintainer that the latter three reports will be addressed by a 'combined patch' to report 764615 and 765468.

[†] CVE-2012-1530, CVE-2012-1525.

[‡] CVE-2015-7497, CVE-2015-7941, CVE-2016-1839, and CVE-2016-2073.

[*] Some were reproducible in Adobe Reader but were not sent to the vendor as security bugs due to low severity.

[⊕] GNOME Bugzilla 759579, 759495, and 759675.

AFL with crawled samples as seeds finds 9, 0 and 12 bugs in total in Sablotron, libxslt, and libxml2 perspectively. On the other hand, Skyfire finds 13, 2 and 4 bugs in Sablotron, libxslt, and libxml2 perspectively. In Sablotron and libxslt, Skyfire finds more bugs and in libxml2, AFL finds more bugs than Skyfire. At last, Skyfire+AFL finds 25, 3 and 18 bugs in Sablotron, libxslt, and libxml2 perspectively. The result indicates that Skyfire + AFL can find more bugs than Skyfire and AFL alone.

Among these bugs, memory corruption vulnerabilities are more significant. Among 19 previously undiscovered vulnerabilities, 16 of them considered as exploitable. We list their CVE identifiers or bug identifiers in Table 5.5. Seven of them are Out-of-bound read vulnerabilities. Two of them are Out-of-bound write vulnerabilities. Seven of them are Use-after-free or double free kind of vulnerabilities.

Figure 5.5 shows how the number of bugs varies according to the time of fuzzing. The unit of time is days and as the fuzzing time increase, the growth rate is slowly falling. However, the number of bugs still gradually increase.

Based on the experimental results of Table 5.4, Table 5.5 and Figure 5.5, we can positively say that our tool generates high-quality seeds and thus increases the bug finding capability of fuzzing.

Table 5.5: New Vulnerabilities and Types

Vulnerability	Type
CVE-2016-6978	Out-of-bound read
CVE-2016-6969	Use-after-free
Pending advisory 1	Double-free / UAF
CVE-2017-2949	Out-of-bound write
CVE-2017-2970	Out-of-bound write
CVE-2015-7115	Out-of-bound read
CVE-2015-7116	Out-of-bound read
CVE-2016-1762	Out-of-bound read
CVE-2016-1835	Use-after-free
CVE-2016-1836	Use-after-free
CVE-2016-1837	Use-after-free
CVE-2016-4447	Out-of-bound read
Pending advisory 2	OOB read / UAF
Pending advisory 3	Out-of-bound read
Pending advisory 4	Use-after-free
Pending advisory 5	Out-of-bound read

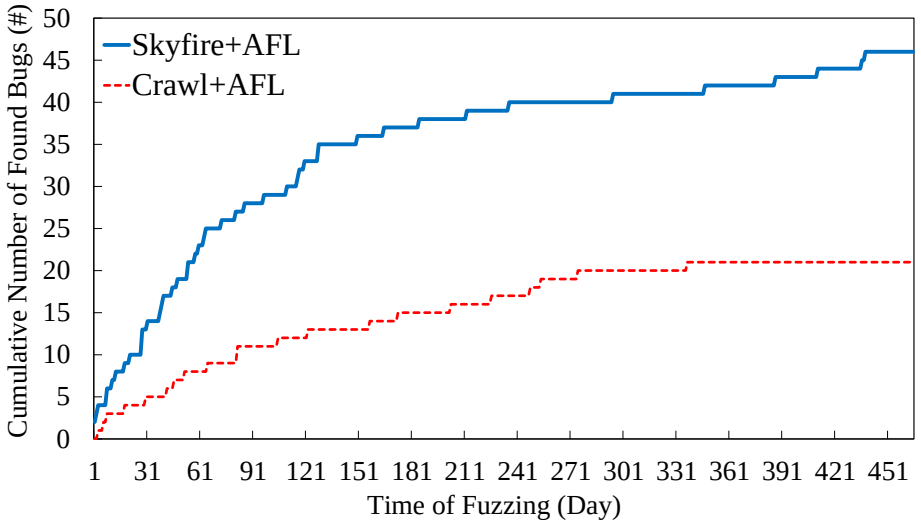


Figure 5.5: The Cumulative Number of Unique Bugs over Time

5.5.3 Code Coverage

In this section, we give the coverage of four configurations. The coverage is calculated using gcov. As shown in Table 5.6, the total number of lines of Sablotron, libxslt, and libxml2 is 10,561, 14,418 and 67,420 respectively. The numbers of functions are 2,230, 778 and 3,235.

Under column *Crawl*, we give the code coverage of the samples crawled; and under column *Crawl+AFL*, we give the code coverage achieved by AFL via taking the samples crawled as seed inputs. Similarly, under column *Skyfire*, we report the code coverage of the generated inputs by Skyfire; and under column *Skyfire+AFL*, we show the code coverage achieved by AFL via taking the generated inputs by Skyfire as seed inputs.

Table 5.6: Line and Function Coverage of Sablotron, libxslt, and libxml2

Program			Line Coverage (%)				Function Coverage (%)			
Name	Lines	Functions	Crawl	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl	Crawl+AFL	Skyfire	Skyfire+AFL
Sablotron 1.0.3	10,561	2,230	34.0	39.0	65.2	69.8	29.8	32.6	48.1	50.1
libxslt 1.1.29	14,418	778	29.6	38.1	57.4	62.5	30.0	34.2	51.9	53.1
libxml2 2.9.4	67,420	3,235	13.5	15.3	22.0	23.8	15.7	16.3	24.1	25.9

We listed the line coverage and function coverage. Crawled samples cover 34.0%, 29.6%, and 15.3% of lines of Sablotron, libxslt, and libxml2 respectively. Though 15 months fuzzing, AFL increases the line coverage to 39.0%, 38.1%, and 15.3%. AFL increase line coverage 5.1% on average. On the other hand, the seeds generated from Skyfire cover 65.2%, 57.4% and 22.0% of source code lines of Sablotron, libxslt and libxml2 respectively. AFL further advance them to 69.8%, 62.5%, and 23.8% respectively. It indicates that the seeds generated by Skyfire already outperform AFL itself and combining AFL and Skyfire reaches the best coverage.

For function coverage, crawled samples cover 29.8% of Sablotron functions, 30.0% of libxslt functions and 15.7% of libxml2 functions. Then AFL increases function coverage about 2.5% after 15 months of fuzzing. Seeds generated from Skyfire cover 48.1% of Sablotron functions, 51.9% of libxslt functions and 24.1% of libxml2 functions. AFL further increase function coverage about 1.7%.

The coverage indicates that our tool can generate well-distributed seeds that can reach more functions of target programs. We investigated the details of line and function coverage of three target programs, as shown in Table 5.7, 5.8, and 5.9. The first column is the name of source code. The second column reports the number of bugs in corresponding source code files.

Table 5.7: Detailed Code Coverage of Sablotron 1.0.3

File (.cpp)	#Bug	Line Coverage (%)				Function Coverage (%)			
	25	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
arena	0	86.8	92.1	92.1	92.1	85.7	85.7	85.7	85.7
base	0	45.7	45.7	92.6	92.6	61.5	61.5	84.6	84.6
context	**1**	**31.8**	**44.9**	**79.6**	**90.6**	**41.0**	**51.3**	**84.6**	**87.2**
datastr	**2**	**69.5**	**70.3**	**79.1**	**82.4**	**68.5**	**69.4**	**79.3**	**81.1**
decimal	**3**	**8.9**	**8.9**	**66.5**	**91.1**	**28.0**	**28.0**	**72.0**	**72.0**
domprovider	0	14.1	19.8	40.2	40.8	20.7	27.6	50.0	50.0
encoding	0	38.1	50.4	52.2	53.1	61.5	69.2	76.9	76.9
error	0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
expr	**1**	**31,4**	**52.6**	**87.0**	**90.3**	**38.1**	**61.9**	**79.4**	**80.4**
hash	0	60.5	60.5	83.2	83.2	61.5	61.5	69.2	69.2
key	0	4.3	4.3	78.4	79.9	12.0	12.0	72.0	72.0
numbering	0	0.0	0.0	65.2	93.5	0.0	0.0	84.6	100.0
output	**6**	**51.8**	**52.0**	**86.9**	**89.3**	**53.8**	**53.8**	**80.8**	**80.8**
parser	0	67.8	69.1	81.8	94.9	45.7	45.7	82.9	97.1
platform	0	68.4	89.5	100.0	100.0	50.0	83.3	100.0	100.0
proc	0	41.2	42.7	67.9	68.7	36.2	40.0	66.2	66.2
sablot	0	23.3	23.3	23.3	23.3	19.0	19.0	19.0	19.0
sdom	0	0.2	0.2	0.2	0.2	1.4	1.4	1.4	1.4
situa	0	59.5	60.7	64.6	65.4	45.0	45.0	52.5	55.0
sxpath	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
tree	**4**	**36.5**	**38.4**	**72.4**	**83.6**	**41.7**	**42.7**	**68.0**	**75.7**
uri	0	48.3	48.3	70.8	72.9	60.0	60.0	75.0	75.0
utf8	**4**	**34.3**	**34.3**	**62.7**	**64.9**	**38.9**	**38.9**	**66.7**	**66.7**
vars	**1**	**11.5**	**15.3**	**86.6**	**89.8**	**22.2**	**29.6**	**92.6**	**92.6**
verts	**3**	**30.9**	**31.7**	**72.0**	**76.5**	**38.3**	**39.5**	**57.5**	**62.3**

Among 25 source code files in Table 5.7, Skyfire + AFL increases 21 of them. Among 32 source code files in Table 5.8, Skyfire + AFL increases 26 of them. Among 38 source code files of libxml2 in Table 5.9, Skyfire + AFL increases 28 of them. The coverage of most of the source code files is increased. For those source code whose coverage is extremely low, we manually investigated the reasons for low coverage.

Some source code is for testing purpose and not executed from the fuzzing configuration. For examples, testThreads.c in libxslt, runsuite.c, runtest.c and runxmlconf.c in libxml2. Some functions are to be triggered by specific parameters to fuzzing command. For example, `xmlcatalog` and `c14n` are specific parameters to trigger certain

Table 5.8: Detailed Code Coverage of libxslt 1.1.29

File (.c)	#Bug	Line Coverage (%)				Function Coverage (%)			
	3	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
common	0	9.6	9.6	80.8	82.7	33.3	33.3	100.0	100.0
date	0	2.7	2.7	21.7	26.6	1.4	1.4	35.7	40.0
dynamic	0	3.1	3.1	3.1	3.1	33.3	33.3	33.3	33.3
exslt	0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
functions	**2**	**3.6**	**9.1**	**73.8**	**78.2**	**23.1**	**38.5**	**92.3**	**92.3**
math	0	5.4	5.4	30.8	33.5	2.8	2.8	44.4	44.4
saxon	0	9.3	9.3	38.4	53.5	12.5	12.5	62.5	75.0
sets	0	7.2	7.2	56.8	58.6	12.5	12.5	62.5	62.5
strings	0	2.7	2.7	43.0	49.2	9.1	9.1	63.6	63.6
attributes	0	3.2	3.2	83.1	85.7	13.3	13.3	86.7	86.7
attrvt	0	65.2	91.3	93.8	95.0	83.3	100.0	100.0	100.0
documents	0	48.4	71.1	75.0	77.3	66.7	77.8	77.8	77.8
extensions	0	46.5	56.2	64.3	64.3	56.5	64.5	71.0	71.0
extra	0	11.5	38.5	47.9	47.9	20.0	40.0	60.0	60.0
functions	0	12.9	30.8	67.8	74.0	33.3	50.0	91.7	91.7
imports	0	60.9	89.9	89.1	89.9	85.7	85.7	85.7	85.7
keys	0	24.0	38.3	86.8	88.3	35.7	57.1	92.9	92.9
namespaces	0	37.1	50.6	74.7	82.9	57.1	57.1	71.4	71.4
numbers	0	0.0	0.0	52.3	87.6	0.0	0.0	52.5	93.8
pattern	0	46.2	62.5	81.4	86.7	57.6	63.6	75.8	78.8
preproc	0	70.7	78.4	94.7	96.5	89.7	89.7	100.0	100.0
security	0	23.5	26.1	53.0	55.7	46.2	46.2	69.2	69.2
templates	0	22.7	43.0	72.2	75.9	18.2	45.5	72.7	72.7
transform	**1**	**41.1**	**55.7**	**72.2**	**77.8**	**49.3**	**62.7**	**74.6**	**76.1**
variables	0	51.4	63.3	68.6	70.8	66.7	69.4	77.8	77.8
xslt	0	68.5	77.9	86.1	87.2	78.4	78.4	89.2	89.2
xsltlocale	0	1.7	1.7	29.7	89.0	16.7	16.7	66.7	66.7
xsltutils	0	22.6	29.5	48.3	56.7	34.1	34.1	48.8	48.8
libxslt-py	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
libxslt	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
testThreads	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
xsltproc	0	28.4	30.0	30.5	30.5	42.9	42.9	42.9	42.9

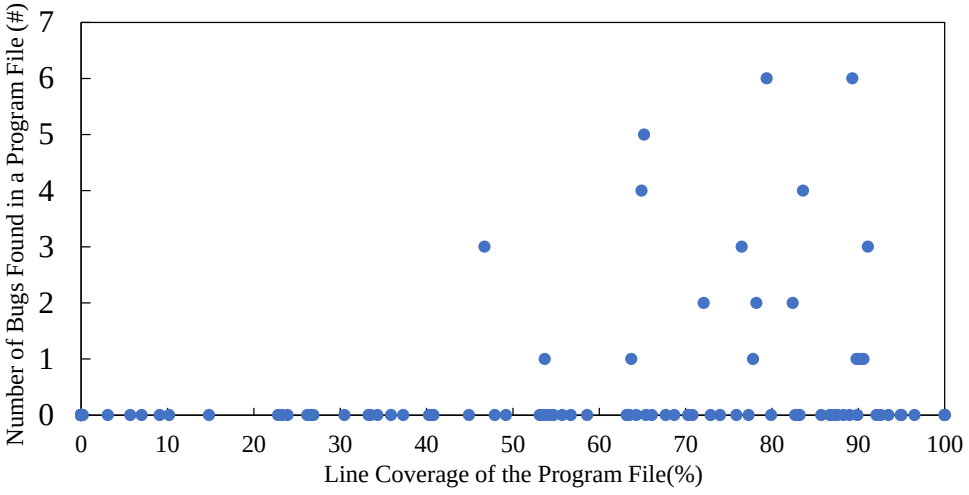
functions in libxml2. While some other source codes are deprecated in the latest version. For example, `sdom` feature in Sablotron and `SAX` in libxml2, which is replaced by `SAX2`.

We also analyzed the relationship between the coverage and the number of bugs in certain source code files, as shown in Figure 5.6. From the figure, we can see that all bugs are found in source code whose line coverage is above 45%. In the meantime, the source code there are bugs found in which have function coverage more than 55%. Although high coverage cannot ensure finding bugs but low coverage indicates the ineffectiveness of fuzzing.

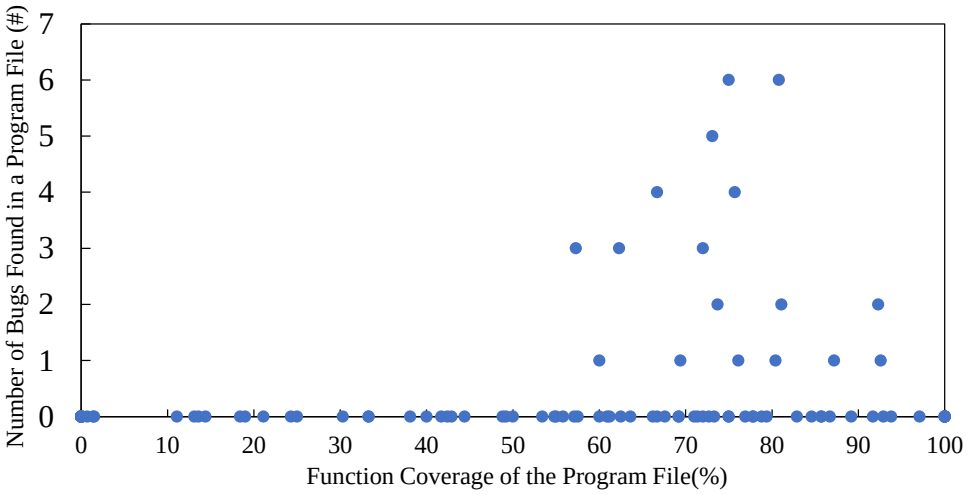
Table 5.9: Detailed Code Coverage of libxml2 2.9.4

File (.c)	#Bug	Line Coverage (%)				Function Coverage (%)			
	18	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
HTMLparser	**3**	**0.4**	**0.4**	**33.6**	**46.7**	**1.0**	**1.0**	**49.0**	**57.3**
HTMLtree	0	1.0	1.0	41.9	54.0	4.2	4.2	33.3	41.7
SAX	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SAX2	**1**	**38.1**	**39.7**	**53.3**	**53.7**	**55.1**	**57.1**	**69.4**	**69.4**
buf	0	48.4	55.9	63.0	63.5	59.5	62.2	67.6	67.6
c14n	0	0.0	0.0	37.5	40.3	0.0	0.0	55.0	57.5
catalog	0	0.4	0.4	23.6	23.9	2.8	2.8	33.3	33.3
chvalid	0	0.0	35.9	35.9	35.9	0.0	11.1	11.1	11.1
debugXML	0	0.0	0.0	0.0	14.8	0.0	0.0	0.0	21.1
dict	**6**	**65.4**	**78.9**	**79.4**	**79.4**	**70.8**	**75.0**	**75.0**	**75.0**
encoding	0	49.2	60.0	65.0	66.1	70.6	73.5	79.4	79.4
entities	0	35.0	38.9	52.1	54.5	43.5	43.5	60.9	60.9
error	**1**	**48.2**	**52.0**	**63.7**	**63.7**	**40.0**	**45.0**	**60.0**	**60.0**
globals	0	19.5	19.5	19.5	26.2	44.2	44.2	44.2	55.8
hash	0	62.1	69.0	70.3	70.3	70.0	70.0	73.3	73.3
legacy	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
list	0	24.2	24.2	34.3	34.3	21.2	21.2	30.3	30.3
parser	**5**	**48.3**	**55.9**	**64.4**	**65.2**	**60.4**	**62.6**	**72.0**	**73.1**
parserInternals	**2**	**56.6**	**66.4**	**72.0**	**72.1**	**57.9**	**60.5**	**73.7**	**73.7**
pattern	0	0.0	0.0	9.1	9.1	0.0	0.0	18.4	18.4
relaxng	0	0.1	0.1	0.1	0.1	0.7	0.7	0.7	0.7
runsuite	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
runtest	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
threads	0	33.3	33.3	33.3	33.3	57.1	57.1	57.1	57.1
tree	0	20.0	21.5	26.8	26.9	28.6	29.8	38.1	38.1
trionan	0	40.6	40.6	40.6	40.6	25.0	25.0	25.0	25.0
uri	0	39.6	59.2	62.6	63.2	71.4	80.0	82.9	82.9
valid	0	27.9	28.5	35.9	37.3	36.4	36.4	47.5	49.2
xinclude	0	0.0	0.0	0.0	10.2	0.0	0.0	0.0	24.3
xmlIO	0	44.5	45.6	55.6	63.2	47.5	48.8	56.2	61.2
xmlcatalog	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
xmllint	0	9.4	9.4	38.9	54.7	5.1	5.1	32.2	42.4
xmlmemory	0	7.0	7.0	7.0	7.0	13.6	13.6	13.6	13.6
xmlreader	0	0.0	0.0	22.8	22.8	0.0	0.0	14.4	14.4
xmlsave	0	57.1	59.9	62.6	67.7	46.6	46.6	48.3	53.4
xmlschemastypes	0	0.1	0.1	0.1	0.1	1.5	1.5	1.5	1.5
xmlstring	0	23.1	35.1	44.3	44.9	35.5	41.9	54.8	54.8
xpath	0	0.1	0.1	5.7	5.7	0.4	0.4	13.1	13.1

Based on the results of Table 5.6-5.9 and Figure 5.6, we can surely say that Skyfire can generate well-distributed seeds and thus cover more functions in target programs, which can better assist fuzzing.



(a) Line Coverage vs. Bugs



(b) Function Coverage vs. Bugs

Figure 5.6: The Relationship between Code Coverage of a File and the Number of Bugs Found in that File

5.5.4 The effectiveness of Context and Heuristics

To investigate the effect of context to PCSG, we designed an experiment and developed a CFG based seed generation approach. To compare the quality of generated seeds of two approaches, we generated 10,000 XML and the same number of XSL samples using PCSG and CFG. Then we feed these samples to XML and XSL engines and check their exit code, which indicates whether the samples pass the semantic checking.

The exit code of 10,000 XSL samples generated by CFG shows that none of them pass the semantic checking since they all violate the first two semantic rules in Table 5.1.

Similarly, 34% of XML samples generated from CFG pass the semantic checking. On the other hand, adding context to CFG increase the ratio of passing semantic checking of XSL to 85% and of XML to 63%. It indicates that considering context greatly improve the model ability of CFG.

We believe the experimental results in Section 5.5.2 and 5.5.3 can indicate the effectiveness of the Heuristics 1 and 2. Heuristics 3 and 4 are to restrict the complexity of seed generation and thus make sure most of the generation can finish in a limited time. Therefore, we investigate the termination ratio of generation using different Heuristics parameters. Figure 5.7a shows the trend of the termination ratio according to the total number of production rule applications.

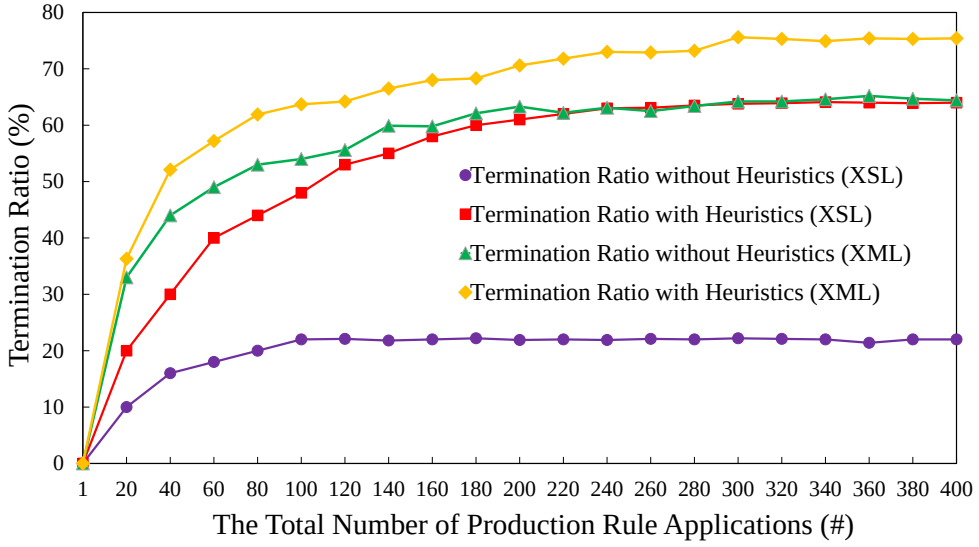
When setting the threshold of production rules application number as 100, without heuristics, only about 20% of XSL generation successfully terminates. While applying heuristics increase termination rate to about 50%. Similarly, the termination ratio of XML generation is about 50% without heuristics and it reaches about 65% with the application of heuristics. So we can positively say that adopting heuristics can increase the success ratio of generation. We practically set the maximum number of production rules as 200.

In Figure 5.7b, we investigate the complexity of generated seeds. We first generated 200,000 XML and XSL samples and counted the number of total production rule applications. We can see that most seeds can be generated within 45 times of production rule applications. That indicates our generated samples are relatively simple.

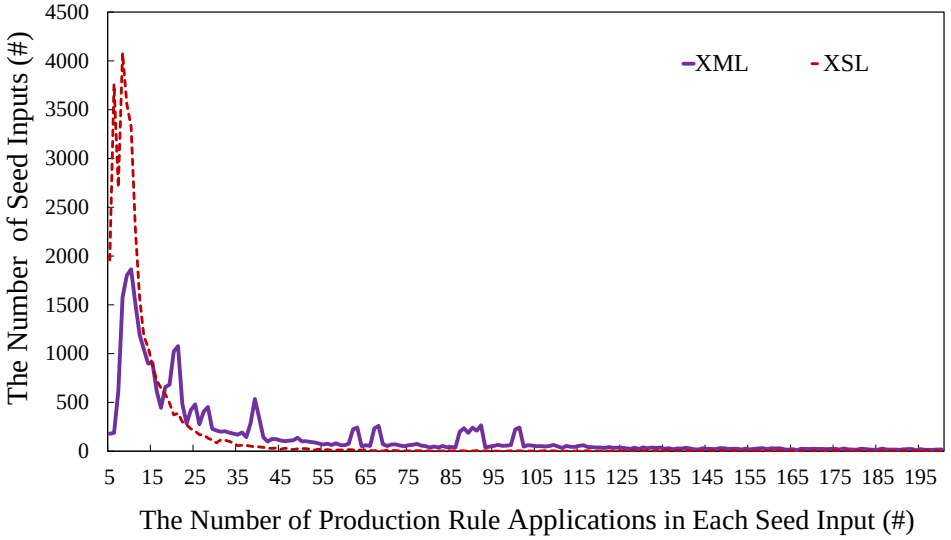
Through above experiments, we can clearly see that context and heuristics we adopted indeed increase the success rate of seed generation. Specially, applying context increase the ratio of passing semantic checking and heuristic rules help to generate well-distributed and relatively simple seeds.

5.5.5 Performance Overhead

In this Section, we measure the time needed to learn a PCSG from crawled 18,686 XSL samples and 19,324 XML samples. As shown in Table 5.10, the PCSG of XSL is



(a) Termination Ratio with and without Heuristics



(b) Distribution of the Number of Rule Applications

Figure 5.7: Evaluation Results for the Used Heuristics in our Seed Generation Approach

learned in about 1.6 hours and the similar is XML. The generation of XSL and XML seeds takes about 20 seconds.

Table 5.10: Performance of Learning and Generation

Time	XSL	XML
Learning (h)	1.5	1.6
Generation (s)	20.3	20.6

From Table 5.10, we can positively answer **RQ4** that Skyfire is scalable with respect to the learning and generation steps.

5.6 Related Work

We categorize work related to our into three groups: mutation approach and generation approach.

5.6.1 Mutation-Based Fuzzing

Mutation-based approaches produce new samples by modifying existing samples using operations such as bitflip or random havoc.

AFL [18] is a successful fuzzer that propose a novel coverage based feedback to guide fuzzing. Previous coverage feedback is on a basic block level. AFL advance it to transition between the basic block and which is considered better capturing coverage information. The mutation operations of AFL includes bit flip, byte flip, overwrite with interesting integers and so on.

Instead of blindly mutating inputs, BuzzFuzz [19] first uses taint analysis to locate the part of the input file which is interesting to fuzz. Then it focuses on mutating that part of samples to find bugs. Since it is smartly mutating part of the input, it can normally keep the structure valid.

Similarly, TaintScope [20] adopts taint analysis to locate the check-sum part of the input, which normally restricts the efficiency of fuzzer. It can also identify which bytes in the input are affecting the control flow. Then these bytes are heavily mutated with the hope of finding bugs.

SAGE [22, 23] combines symbolic execution with fuzzing. When fuzzing stuck, SAGE advance it by solving the path constraint. All the constraints in a path are negated one by one. Thus a single symbolic execution can generate thousands of new tests.

Babić et al. [21] propose an approach, which first runs target programs with part of inputs to generate a visibly pushdown automaton to model the control flow of target programs. Then the second stage they static analyze the pushdown automaton to find

potential vulnerabilities. At the last stage, directed fuzzing is conducted towards the potential vulnerabilities.

To find buffer overflow type of bugs, Dowser [24] first identifies the source code which accesses an array in a loop. Then authors apply taint analysis to locate which input bytes influence the array index and then mutate these bytes to discover bugs. BORG [25] firstly statically identify source code which accesses buffer and then guides symbolic execution towards this code.

Driller [94] is built on top of AFL. When AFL stops reporting any paths as 'favorite', Driller will selectively trace inputs generated by AFL and find the basic block that fails all of the inputs. Driller then applies angr to synthesize inputs that pass those basic block transitions and give it to AFL. Later AFL will mutate these synthesized inputs as normal to find further paths.

All of the above works are successful in mutating unstructured inputs, where some bytes greatly influence control flow. While when inputs are highly structured, such as JavaScript and XML, it is difficult to find a relationship between path constraints and specific bytes. On this occasion, our approach can advance fuzzing with well-distributed seeds.

5.6.2 Generation-Based Fuzzing

Generation-based approaches produce samples using a grammar or specification, similar to Skyfire.

Peach [26] is a smart-fuzzer which is capable of conducting both generation and mutation based fuzzing. Peach requires users to create PeachPit files that define the structure of data to be fuzzed. Peach has been under active development for several years. Spike [27] is developed by Immunity to fuzz binary protocols in less time.

CSmith [30] finds hundreds of bugs in compilers using differential testing. The test cases are C language and which are generated by the grammar. LangFuzz [31] mutates poc samples for previous bugs. It first parses the corpus of JavaScript samples into

syntax trees and collects sub-trees from them, so-called code fragments. Then collected code fragments are used to replace sub-trees of previous bug samples. Thus LangFuzz finds hundreds of severe bugs. Radamsa [107] learns an artificial context-free grammar from the corpus and then used to generate samples from it. mangleme [33] generates a basic set of broken HTML tags to fuzz browsers in its early stage.

Model-based approach normally can pass the syntax checking but has difficulty with passing semantic checking. There is another line of approaches using grammar to guide generation.

Jsfunfuzz [34] is most successful JavaScript fuzzer creates random JavaScript functions to fuzz JavaScript engines. Jsfunfuzz has a hard-coded JavaScript grammar inside. Dewey et al. [35] use constraint logic programming to generate samples. Users can write declarative predicates specifying programs features which are interesting to fuzz.

The disadvantage of these approaches is that specifying predicates or generation grammar is labor-intensive and error-prone.

5.7 Conclusion

In this work, we have proposed a novel data-driven seed generation approach, named Skyfire, to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. We use the large corpus of samples and their grammar to automatically extract the semantic rules and the frequency of production rules, and holistically incorporate them by learning a PCSG. The learned PCSG is then used to generate well-distributed seeds.

Using the seeds generated to fuzz several XSLT, XML, JavaScript and Rendering engines, we have empirically shown that Skyfire can generate well-distributed seeds and help to improve the code coverage and bug-finding capability of fuzzers. We discovered 19 new memory corruption bugs (among which we discovered 16 new vulnerabilities and received 33.5k USD bug bounty rewards) and 32 denial-of-service bugs.

6

Vulnerability Detection via Grammar-Aware Greybox Fuzzing

The previous chapter generates well-distributed seeds for fuzzing, however, the current fuzzing approaches are too dumb to fuzz browsers which accept high-structured inputs. In this chapter, we advance the fuzzing approach to become grammar-aware. Grammar-aware fuzzer can save much time by not generating lots of syntax invalid samples and exploring parsing errors.

6.1 Introduction

The current coverage-based greybox fuzzers can effectively fuzz programs that process compact and unstructured inputs (e.g., images). However, some challenges arise when they are used to target programs that process structured inputs (e.g., XML and JavaScript) that often follow specific grammars. Such programs often process the inputs in stages, i.e., syntax parsing, semantic checking, and application execution [93].

First, the trimming strategies (e.g., removal of chunks of data) adopted in AFL are grammar-blind, and hence can easily violate the grammar or destroy the input structure. As a result, most test inputs in the queue cannot be effectively trimmed to keep them syntax-valid. This is especially the case when the target program can process a part of a test input (triggering coverage) but errors out on the remaining part. This will greatly affect the efficiency of AFL because (i) it needs to spend more time on fuzzing the test inputs whose structures are destroyed, but only finds parsing errors and gets stuck at the syntax parsing stage, which heavily limits the capability of fuzzers in finding deep bugs; and (ii) the test inputs become larger and larger as fuzzing proceeds, which greatly increases the overhead of each mutation as well as the number of mutations applied to each test input.

Second, the mutation strategies (e.g., bit flipping) of AFL are grammar-blind, and hence most of the mutated test inputs fail to pass the syntax parsing and are rejected at an early stage of processing. As a result, it is difficult for AFL to achieve large-step mutations; e.g., it is difficult to obtain *Content-Length: -1* from mutating *Set-Cookie: FOO=BAR* via small-step bit flipping mutations. AFL also needs to spend a large amount of time struggling with syntax correctness, while only finding parsing errors. Therefore, the effectiveness of AFL to find deep bugs is heavily limited for programs that process structured inputs.

Third, programs that process structured inputs often have intrinsic indeterminate behaviors (e.g., caused by multi-threads or memory allocations), which leads to the low stability of AFL, i.e., different executions of the target program against the same test input have different coverage. Consider a multi-thread program where thread 1 executes

block A, thread 2 executes block B, and there is no causality between blocks A and B. Different runs of this program by AFL might first execute block A or block B, and thus have different branch coverage. Such low stability makes it difficult for AFL to decide whether new coverage is achieved and whether a mutated test input should be added to the queue, which greatly affects the effectiveness of AFL.

The Proposed Approach. To address the challenges, we propose a new grammar-aware coverage-based greybox fuzzing approach for programs that process structured inputs. We also implement the proposed approach as an extension to AFL, named Superion. Our approach takes as inputs a *target program* and a *grammar* of the test inputs that is often publicly available. Based on the grammar, we parse each test input into an AST. Using ASTs, we introduce a grammar-aware trimming strategy that can effectively trim test inputs while keeping the input structure valid. This is realized by iteratively removing each subtree in the AST of test input and observing coverage differences. Besides, we propose two grammar-aware mutation strategies that can quickly carry the fuzzing exploration beyond syntax parsing. In particular, we first enhance AFL's dictionary-based mutation strategy by inserting and overwriting tokens in a grammar-aware manner, and then propose a tree-based mutation strategy that replaces one subtree in the AST of a test input with the subtree from the test input itself or another test input in the queue. Furthermore, we leverage block coverage to mitigate the stability problem caused by multi-threads, and we also propose a selective instrumentation strategy to preclude the source code which is responsible for indeterminate behaviors for enhancing the fuzzing stability.

To evaluate the effectiveness of Superion, we conducted experiments on one XML engine libplist, and three JavaScript engines, WebKit, Jerryscript, and ChakraCore. Specifically, we compared our approach with AFL with respect to the code coverage and the bug-finding capability. The results have demonstrated that Superion can effectively improve the code coverage over AFL by 16.7% in line coverage and 8.8% in function coverage, and can significantly improve the bug-finding capability over AFL by finding 34 new bugs (among which 22 new vulnerabilities were discovered with CVEs assigned). Besides, we compared Superion with jsfunfuzz [34], which is a successful

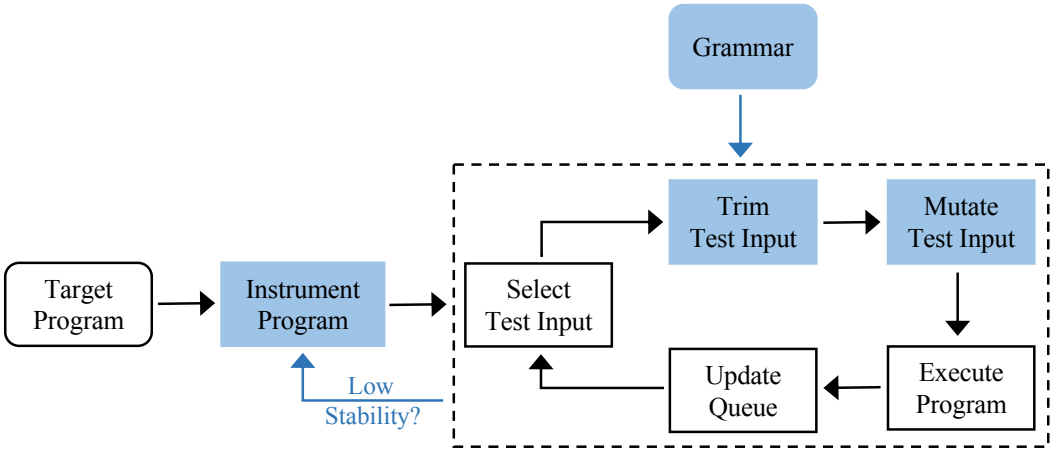


Figure 6.1: The General Workflow of Superion with the Highlighted Differences from AFL (see Figure 2.2)

fuzzer specifically designed for JavaScript. However, it failed to find any new bugs. Furthermore, we have also empirically indicated that our grammar-aware trimming strategy can effectively trim test inputs while keeping them syntax-valid; our grammar-aware mutation strategies can effectively generate new test inputs that can trigger new coverage and our selective instrumentation strategy can greatly improve fuzzing's stability.

6.2 Our Approach

To address the challenges of coverage-based greybox fuzzing, we propose a novel grammar-aware coverage-based greybox fuzzing approach, which targets programs that process structured inputs. We implement the approach as an extension to AFL [18], named Superion. Figure 6.1 introduces the workflow of Superion and highlights the differences from AFL (see Figure 2.2). In particular, a context-free grammar of the test inputs is needed, which is often publicly available (e.g., in ANTLR's community [102]). We introduce a grammar-aware trimming strategy (Section 6.2.1), two grammar-aware mutation strategies (Section 6.2.2), and a selective instrumentation strategy (Section 6.2.3) with the purpose of improving the effectiveness of AFL.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
        <key>Some ASCII string</key>
        <string></string>
        <data>
        </data>
</dict>
</plist>
```

Figure 6.2: An Example of AFL's Built-In Trimming

6.2.1 Grammar-Aware Trimming Strategy

The built-in trimming strategy in AFL is grammar-blind and treats a test input as chunks of data. Basically, it first divides the test input to be trimmed into chunks of len/n bytes where len is the length of the test inputs in bytes, and then tries to remove each chunk sequentially. If the coverage remains the same after the removal of a chunk, this chunk is trimmed. Note that n starts at 16 and increments by a power of two up to 1024. This strategy is very effective for unstructured inputs. However, it cannot effectively prune structured inputs while keeping them syntax-valid, possibly making AFL stuck in the fuzzing exploration of syntax parsing without finding deep bugs.

Example. Figure 6.2 gives an example of AFL's built-in trimming on an XML test input with respect to libplist (an XML engine), where "l versio" and "dict> </plis" are trimmed (highlighted by strikethrough). The trimmed test input is syntax-invalid but has the same coverage with the original test input due to the gap between the implementation of libplist and grammar specification. Hence, the trimmed test input is used for further fuzzing even though its grammar is destroyed by AFL's built-in trimming.

To ensure the syntax-validity of trimmed test inputs, we propose a grammar-aware trimming strategy, whose procedure is given in Algorithm 6. It first parses the test input to be trimmed in according to the grammar G into an AST $tree$ (Line 2). If any parsing errors occur (as in's structure may be destroyed by mutations), then it uses AFL's built-in trimming strategy rather than directly discarding it (Line 3–5); otherwise, it attempts to trim a subtree n from $tree$ (Line 6–7). If the coverage is different after n

Algorithm 6 Grammar-Aware Trimming

Input: the test input to be trimmed in, the grammar G
Output: the trimmed test input ret

```
 1: while true do
 2:     parse in according to G into an AST tree
 3:     if there are any parsing errors then
 4:         return built-in-trimming (in)
 5:     end if
 6:     for each subtree n in the tree do
 7:         ret = remove n from tree
 8:         run the target program against ret
 9:         if coverage remains the same then
10:             in = ret
11:             break
12:         else
13:             add n back to the tree
14:         end if
15:         if n is the last subtree in the tree then
16:             return ret
17:         end if
18:     end for
19: end while
```

```
...
try{eval("M:if(([15,16,17,18].some(this.unwatch(\"x\"),(([window if([[]])])[this.
    prototype])))) else{true;return null;}");} catch(ex){}
try{eval("M:while((null >=\"\")&&0){/a/gi}");} catch(ex){}
try{eval("\nbreak M;\n");} catch(ex){}
try{eval("L:if((window[(1.2e3.x::y)]).x) return null; else if((uneval(window))++.
    propertyIsEnumerable(\"x\")){CollectGarbage()}");} catch(ex){}
try{eval("/*for..in*/for(var x in ((({}).hasOwnProperty
    ){[,].hasOwnProperty(\"x\")))/*for..in*/ M:for(var
    [window, y] =(-1) in this) [1,2,3,4].slice");} catch
    (ex){}
try{eval("if(\"\"){}else if(x4) {null;}");} catch(ex){}
try{eval("{}");} catch(ex){}
try{eval("for(var x = x in x - /x/ ){}");} catch(ex){}
try{eval("if((uneval(x, x))) var x = false; else if((null\n.unwatch(\"x\"))) throw
    window; else {} return 3;");}catch(ex){}
...
```

Figure 6.3: An Example of Grammar-Aware Trimming

is trimmed, then n cannot be trimmed (Line 12–14), and it tries to trim next subtree; otherwise, n is trimmed, and it re-parses the remaining test input (Line 9–11) and then repeats the procedure until no subtree can be trimmed (Line 15–16). Thus, we resort to AFL's built-in trimming only when our tree-based trimming is not applicable since sometimes invalidity is also useful.

Example. Figure 6.3 shows an example of our trimming strategy on a JavaScript

test input, where a complete `try-catch` statement (highlighted by strikethrough) is trimmed without introducing any coverage differences. However, it is almost impossible for AFL's built-in trimming strategy to prune such a complete statement.

6.2.2 Grammar-Aware Mutation Strategies

The default mutation strategies (e.g., bit flipping or token insertion) in AFL are too fine-grained and grammar-blind to keep the input structure following the underlying grammar. Therefore, we propose two grammar-aware mutation strategies to improve the mutation effectiveness on triggering new program behaviors.

6.2.2.1 Enhanced Dictionary-Based Mutation

The instrumentation-guided nature of AFL enables to distinguish between combinations that are nonsensical and ones that actually follow the rules of the underlying grammar and therefore trigger new states in the instrumented binary. The dictionary feature proves its effectiveness by finding a couple of crashes in SQLite, which is well-tested.

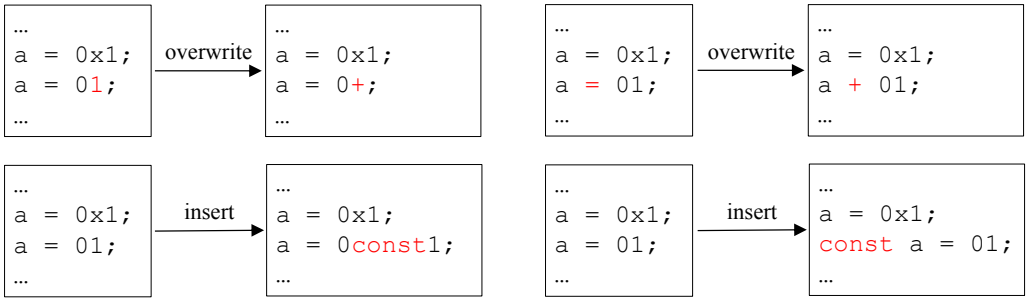
Dictionary-based mutation [108] was introduced to make up for the grammar-blind nature of AFL. The dictionary is referred to as a list of basic syntax tokens (e.g., reserved keywords) which can be provided by users or automatically identified by AFL. Every token is inserted between every two bytes of the test input to be mutated, or written over every byte sequence of the same length of the token. Such mutations can generate syntax-valid test inputs but are inefficient as most of the generated inputs have destroyed the structure.

Therefore, we propose the enhanced dictionary-based mutation as shown in Algorithm 7. This algorithm leverages the key fact that the tokens (e.g., variable names, function names, or reserved keywords) in a structured test input normally only consist of alphabets or digits. Hence, it first locates the token boundaries in a test input by iteratively checking whether the current and the next byte are both alphabet or digit (Line 3–10). Then it inserts each token in the dictionary to each located boundary, which avoids

Algorithm 7 Dictionary-Based Mutation

Input: the test input in, the dictionary D
Output: the set of mutated test inputs T

```
1:  T = ∅
2:  l = the length of in
3:  for i = 0; i < l; do
4:      j = i + 1
5:      curr = *(u8*)(in's address + i)  // current byte of in
6:      next = *(u8*)(in's address + j)  // next byte of in
7:      while j < l && curr and next are alphabet or digit do
8:          j = j + 1
9:          next = *(u8*)(in's address + j)
10:     end while
11:     for each token d in D do
12:         insert d at i of in / overwrite i to j of in with d
13:         T = T ∪ {in}
14:     end for
15:     i = j
16: end for
```



(a) Original (b) Enhanced

Figure 6.4: An Example of Dictionary-Based Mutation

the insertion between the consecutive sequence of alphabets and digits and thus greatly decreases the number of token insertions (Line 11–14). Similarly, it writes each token in the dictionary over the content between every two located boundaries, which also greatly decreases the number of token overwrites. Such token insertions and overwrites not only maintain the structure of mutated test inputs but also decreases the number of mutated test inputs, hence greatly improving the effectiveness and efficiency of dictionary-based mutation.

Example. Figure 6.4 illustrates the difference between the original and enhanced dictionary based mutation. In the original one, `01` is not treated as a whole, and thus `1` can be overwritten by `+` and `const` can be inserted between `0` and `1`, which destroys the

structure without introducing any new coverage. In the enhanced one, `01` is identified as a whole, and hence the mutated test inputs in Figure 6.4a will not be produced. Instead, it can generate the mutated test inputs in Figure 6.4b more efficiently, which are taken from our experiments and both lead to new coverage.

6.2.2.2 Tree-Based Mutation

Dictionary-based mutation is aware of the underlying grammar in an implicit way. To be explicitly aware of the grammar and thus producing syntax-valid test inputs, we utilize the grammar knowledge and design a tree-based mutation, which works at the level of ASTs. Different from the tokens used in dictionary-based mutation, AST actually models a test input as objects with named properties and is designed to represent all the information about a test input. Thus, ASTs provide a suitable granularity for a fuzzer to mutate test inputs.

The paper [109] believes that using an AST is a great strategy for fuzz testing. Previously, we had tried to use a regex-based approach. This involved stringifying the tests and finding specific tokens to replace or shuffle. But regexes are opaque and fragile. They are a nightmare to maintain, and it is very easy to introduce subtle mistakes that make the mutations less effective.

Algorithm 8 shows the procedure of our tree-based mutation. It takes as inputs a test input tar to be mutated, the grammar G, and a test input pro that is randomly chosen from the queue. It first parses tar according to G into an AST tar_tree; and if any parsing errors occur, tar is a syntax-invalid test input and we do not apply tree-based mutation to tar (Line 3–6). If no error occurs, it traverses tar_tree, and stores each subtree in a set S (Line 7–9). Then it parses pro into an AST pro_tree, and stores each subtree of pro_tree in S if there is no parsing error (Line 10–15). Here S serves as the content provider of mutation. Then, for each subtree n in tar_tree, it replaces n with each of the subtree s in S to generate a new mutated test input (Line 16–21). Finally, it returns the set of mutated test inputs.

Algorithm 8 Tree-Based Mutation

Input: the test input tar, the grammar G, the test input pro
Output: the set of mutated test inputs T

```
1:  T = ∅
2:  S = ∅  // the set of subtrees in tar and pro
3:  parse tar according to G into an AST tar_tree  // Heuristic 1
4:  if there are any parsing errors then
5:      return
6:  end if
7:  for each subtree n in tar_tree  // Heuristic 3 do
8:      S = S ∪ {n}
9:  end for
10: parse pro according to G into an AST pro_tree  // Heuristic 1
11: if there is no parsing error then
12:     for each subtree n in pro_tree  // Heuristic 3 do
13:         S = S ∪ {n}
14:     end for
15: end if
16: for each subtree n in tar_tree  // Heuristic 2  do
17:     for each subtree s in S do
18:         ret = replace n in tar_tree's copy with s
19:         T = T ∪ {ret}
20:     end for
21: end for
22: return  T
```

The size of this returned set can be the multiplication of the number of subtrees in tar_tree and the number of subtrees in tar_tree and pro_tree, which could be very large. As an example, our tree-based mutation on tar and pro whose number of subtrees is respectively 100 and 500 will generate $100 \times (100 + 500) = 60,000$ test inputs. This will add burden to the program execution step during fuzzing, making fuzzing less efficient. To relieve the burden, we design three heuristics to reduce the number of mutated test inputs. For clarity, we do not elaborate these heuristics in Algorithm 8, but only show where they are applied.

- **Heuristic 1: Restricting the size of test inputs.** We limit the size of test inputs (i.e., tar and pro in Algorithm 8) as 10,000 bytes long (Line 3 and 10). Hence we do not apply tree-based mutation to tar if tar is more than 10,000 bytes long and we do not use subtrees of pro as the content provider of mutation if pro is more than 10,000 bytes long. The reasons are that a larger test input usually needs a larger number

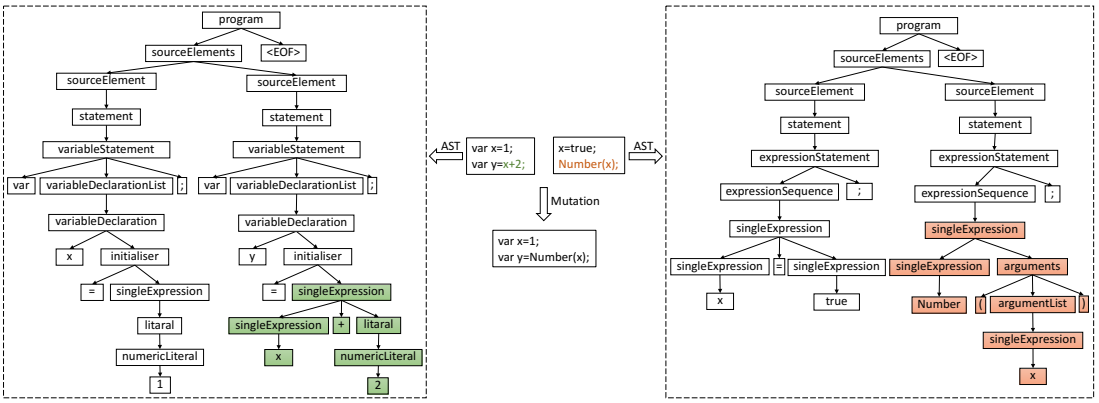


Figure 6.5: An Example of Tree-Based Mutation

of mutations; more memory is required to store the AST of a larger test input and a larger test input often has a slower execution speed.

- **Heuristic 2: Restricting the number of mutations.** If there are more than 10,000 subtrees in tar and pro, we randomly select 10,000 from all subtrees in S as the content provider of mutation (Line 16). As a consequence, we keep the number of mutations on each test input in the queue under 10,000 to make sure that each test input in the queue has the chance to get mutated.

- **Heuristic 3: Restricting the size of subtrees.** We limit the size of subtrees (i.e., each subtree in S in Algorithm 8) as 200 bytes long (Line 7 and 12). Thus we do not use the subtrees of tar and pro as the content provider of mutation if the subtree is more than 200 bytes long. Notice that 200 bytes are long enough to include complex statements.

The threshold values in these heuristics were empirically established as good ones. Although the grammatical crossover mutation is effective, it can be a time-consuming step. This is a heavy step to conduct than other mutation strategies. Therefore we investigate the effectiveness of replacing different subtrees.

As the fuzzer traverses the tree, it also picks up values that it thinks are interesting, which are usually primitive values like strings and numbers. These values are harvested and used to construct the final value of the placeholder nodes.

Grammar node type. We can replace each subtree with any other subtrees without caring about their grammar types, or we can only replace one subtree with another with same grammar type.

Example. Figure 6.5 shows an example of our tree-based mutation. The left-side is the AST of the test input to be mutated (i.e., tar in Algorithm 8), and the right-side is the AST of the test input that provides the content of mutation (i.e., pro in Algorithm 8). Here the subtree corresponding to the expression $x + 2$ in tar is replaced with the subtree corresponding to the expression $Number(x)$ in pro, resulting in a new test input.

6.2.3 Selective Instrumentation Strategy

Stability, i.e., whether a target program has a stable coverage (behaves stably) against test input, is very crucial to coverage-based greybox fuzzing. If the coverage becomes unstable across different executions of a test input, it is difficult for a fuzzer to determine whether a new coverage is found, and further whether a mutated test input should be added to the queue. As a result, most of the fuzzing efforts are wasted.

The stability of AFL is calculated as follows: $stab\_ratio = 100 - var\_byte\_count * 100/t\_bytes$, where t_bytes equals to the number of all non-255 bytes, var_byte_count equals to the number of var_bytes, if $trace\_bytes[i] != first\_trace[i]$.

Causes of low stability are the indeterminate behaviors in a target program (e.g., multi-threads and hashing) as well as in a fuzzer (e.g., incomplete clean-up or re-initialization of the state between subsequent program executions). To reduce the instability caused by multi-threads (that are quite common), we propose to use block coverage instead of the original branch coverage as the coverage feedback to AFL. Therefore, the coverage becomes stable no matter how the threads are scheduled, i.e., in what orders the basic blocks in the threads are executed.

Compared to the instability caused by multi-threads, the instability caused by other indeterminate behaviors is severer. To this end, we propose to first identify the source code that is responsible for indeterminate behaviors and then selectively instrument the

Algorithm 9 Indeterminate Code Identification

Input: the set of test inputs that cause indeterminate behaviors I
Output: the set of functions that behaves inconsistently F

```
1:  F = ∅
2:  for each test input t in I do
3:      execute program against t to get coverage report rep_0
4:      for i = 1; i < N; i++ do
5:          execute program against t to get coverage report rep_i
6:          diff rep_i with rep_0
7:          get functions E and lines L that have variable coverage
8:          F = F ∪ E ∪ {f | ∀l ∈ L, l is located in function f}
9:      end for
10: end for
11: return F
```

target program to preclude those identified codes. Therefore, indeterminate behaviors will not reflect themselves in the coverage feedback to AFL, and thus AFL will not be misled.

Algorithm 9 shows the procedure to identify the indeterminate code. The input is a set of test inputs that trigger indeterminate behaviors. Note that this set is automatically collected by AFL. For each test input t in the set, the algorithm executes the target program against t for N times to get their code coverage report (Line 3–5). Here N was empirically set to 4. The report includes the number of times each function and line of code are executed, which is obtained through the coverage tool `gcov` [110]. Then, it analyzes the differences among the reports and identifies the set of functions E as well as the set of lines of code L whose execution times vary across program executions (Line 6–7), including the case that a function or line of code is executed in some runs but not in others. At last, it returns the functions E as well as the functions that contain the lines of code in L. These functions are considered to be responsible for indeterminate behaviors.

Once we identify the functions that can lead to indeterminate behaviors, we instrument the source code of the target program in a selective way to preclude those identified functions, which is achieved through the partial instrumentation extension for AFL [111]. Note that Algorithm 9 is triggered when the default stability metric of AFL is below 85%, after which the fuzzing efforts are considered to be meaningless [18]; then, we stop the fuzzer, apply selective instrumentation, and restart the fuzzer.

Table 6.1: Files that Have Indeterminate Behaviors in WebKit

Source File	# Functions	# Lines of Code
/Source/WTF/wtf/SmallPtrSet.h	1	12
/Source/WTF/wtf/HashTraits.h	1	3
/Source/WTF/wtf/text/WTFString.h	0	1
/Source/WTF/wtf/HashMap.h	13	3
/Source/WTF/wtf/RefPtr.h	1	9
/Source/WTF/wtf/HashFunctions.h	1	14
/Source/WTF/wtf/HashTable.h	23	62
/Source/WTF/wtf/FastMalloc.cpp	3	7
/Source/bmalloc/bmalloc/Heap.h	1	5
/Source/bmalloc/bmalloc/SmallLine.h	0	3
/Source/bmalloc/bmalloc/Algorithm.h	0	3
/Source/bmalloc/bmalloc/Allocator.cpp	2	12
/Source/bmalloc/bmalloc/Cache.h	2	7
/Source/bmalloc/bmalloc/PerHeapKind.h	0	1
/Source/bmalloc/bmalloc/Vector.h	0	1
/Source/bmalloc/bmalloc/PerThread.h	0	2
/Source/bmalloc/bmalloc/StaticMutex.h	1	3
/Source/bmalloc/bmalloc/FixedVector.h	2	6
/Source/bmalloc/bmalloc/BumpAllocator.h	0	5
/Source/bmalloc/bmalloc/Sizes.h	0	1
/Source/bmalloc/bmalloc/List.h	0	1
/Source/bmalloc/bmalloc/Map.h	0	3
/Source/bmalloc/bmalloc/Heap.cpp	5	35
/Source/bmalloc/bmalloc/bmalloc.h	0	2
/Source/bmalloc/bmalloc/SmallPage.h	0	6
/Source/bmalloc/bmalloc/Deallocator.h	1	7
/Source/bmalloc/bmalloc/Deallocator.cpp	2	9
/Source/bmalloc/bmalloc/Allocator.h	1	5
/Source/bmalloc/bmalloc/Chunk.h	0	12
/Source/JavaScriptCore/jit/JITThunks.h	2	4

Example. Table 6.1 shows a small part of the source files, the number of functions, and the number of lines of code that are identified by Algorithm 9 for the program WebKit. Its indeterminate behaviors are mostly caused by hashing (e.g., HashTable, HashMap and Hash Lookup), memory (de)allocations, heap, and map, which are related to the program itself.

6.3 Evaluation

We implemented our approach in 472 lines of C/C++ code by extending AFL [18]. Particularly, given the grammar of test inputs, we adopted ANTLR 4 [101] to automatically generate the lexer and parser, and used ANTLR 4 C++ runtime to parse test inputs and realize our trimming and mutation strategies. In that sense, our approach is general and easily adaptable for other structured test inputs. It is worth mentioning that we detected three memory leak issues in ANTLR 4 C++ runtime. Since we used one parser to parse each test input, the memory leak in the parser was accumulated gradually. Therefore, we detected such memory leak issues using Valgrind [112] and fixed them, which were also reported to and confirmed by developers of ANTLR 4.

6.3.1 Evaluation Setup

To evaluate the effectiveness and generality of our approach, we selected two target languages and three target programs, and compared our approach with AFL [18] with respect to the bug-finding capability and code coverage.

Target Languages. We chose XML and JavaScript as the target languages with different structure level. Their grammars are all publicly available in ANTLR's community [102]. In particular, XML is a widely-used markup language, which defines a set of rules for encoding documents. It has been widely used in a variety of applications. As shown in the second column of Table 6.2, the XML grammar we adopted in our evaluation only contains eight symbols, and thus XML can be considered to be weakly-structured. On the other hand, JavaScript is an interpreted programming language, which is employed by most websites and supported by all modern web browsers. The JavaScript grammar we adopted contains 98 symbols, and thus its structure level can be regarded as strong. It was invented by Brendan Eich in 1995 and became an ECMA standard in 1997 and ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

Table 6.2: Target Languages and Their Structure and Samples

Language	# Symbols	Structure Level	# Samples
XML	8	Weak	9,467 (534)
JavaScript	98	Strong	20,845 (2,569)

Open-source projects often include extensive sets of input data for testing, which can be freely reused as a fuzzing starting point. For example, there are lots of JavaScript samples under the directory of WebKit, Jerryscript, and ChakraCore, which would be otherwise very difficult to obtain in the wild. In fuzzing, it is important to get rid of most of the redundancy in the input corpus. This is because of both the base one and the living one evolving during fuzzing. In the context of a single test case, we should maximize the ratio of program states explored to input size, which strives for the highest byte-to-program-feature ratio. Each portion of a file should exercise a new functionality, instead of repeating constructs found elsewhere in the sample. Likewise, in the whole corpus, the ratio of program states explored to the sum of input samples should be maximized, which ensures that there are not too many samples which all exercise the same functionality. We should enforce program state diversity while keeping the corpus size relatively low.

As indicated by the last column of Table 6.2, we crawled 9,467 XML samples from the Internet, and 20,845 JavaScript samples from the test inputs of the two open-source JavaScript engines WebKit and Jerryscript. They were used as the initial test inputs (i.e., seeds) for fuzzing. As suggested by AFL, *afl-cmin* should be used to identify the set of functionally distinct seeds that exercise different code paths in the target program when a large number of seeds are available. Therefore, we used *afl-cmin* on the samples, and identified 534 and 2,569 distinct XML and JavaScript samples as the seeds for fuzzing, as shown in the parentheses in the last column of Table 6.2. Notice that, before fuzzing, we pre-processed the JavaScript samples by removing all the comments as comments, especially multi-line comments, account for a considerable percentage of waste of mutation.

Target Programs. We selected one open-source XML engine libplist and three open-source JavaScript engines WebKit, Jerryscript, and ChakraCore as the target programs

Table 6.3: Target Programs and Their Fuzzing Configuration

Program	Version	# Lines	# Func.	Coverage	Timespan
libplist	1.12	3,317	316	Edge	3 months
WebKit	602.3.12	151,807	60,340	Block	3 months
Jerryscript	1.0	19,963	1,100	Edge	3 months
ChakraCore	1.10.1	236,881	74,132	Block	3 months

for fuzzing. The first four columns of Table 6.3 list the details of these programs, including the version, the number of lines of code, and the number of functions. Particularly, libplist is a small portable C library to handle Apple Property List format files in binary or XML. It is widely used on iOS and Mac OS. WebKit is a cross-platform web browser engine. It powers Safari, iBooks and App Store, and various Mac OS, iOS and Linux applications. Jerryscript is a lightweight JavaScript engine for the Internet of Things, intended to run on very constrained devices (e.g., microcontrollers). Chakra JavaScript engine that powers Microsoft Edge. We chose these programs because they are security-critical and widely-fuzzed. Thus, finding bugs in them are significant.

As shown in the fifth column of Table 6.3, we adopted edge coverage for both libplist and Jerryscript during fuzzing because their stability was almost 100%, and block coverage for WebKit. Besides, we fuzzed libplist and Jerryscript for one month due to their relatively small size and completed more than 100 cycles of fuzzing. For WebKit, given its large size, we have fuzzed for about three months at the time of writing, and have not finished one cycle yet. Here a cycle means the fuzzer went over all the interesting test inputs (triggering new coverage) discovered so far, fuzzed them, and looped back to the very beginning.

Research Questions. Using the previous evaluation setup, we aim to answer the following six research questions.

- **RQ1:** How is the bug-finding capability of Superion?
- **RQ2:** How is the code coverage of Superion?
- **RQ3:** How effective is our grammar-aware trimming?
- **RQ4:** How effective is our grammar-aware mutation?

- **RQ5:** How effective is our selective instrumentation?
- **RQ6:** What is the performance overhead of Superion?

We conducted all the experiments on machines with 28 Intel Xeon CPU E5-2697v3 cores and 64GB memory, running 64-bit Ubuntu 16.10 as the operating system.

6.3.2 Discovered Bugs and Vulnerabilities (RQ1)

Table 6.4 lists unique bugs discovered in libplist, WebKit and Jerryscript by Superion. In libplist, we discovered eleven previously unknown bugs, from which we discovered ten vulnerabilities with CVE identifiers assigned. For anonymity, we omitted the last three digits of CVE and bug identifiers. In WebKit, ten new bugs were found, and three of them were vulnerabilities with CVE identifiers assigned. It is worth mentioning that these bugs obtained high appraisals, e.g., "*This bug is really interesting*", "*Thank you for the awesome test case*" and "*This bug has existed for a long time. A quick look through blame would say for 4-5 years or so*". In Jerryscript, we found three previously unknown bugs, from which we discovered one vulnerability with CVE identifiers assigned. Note that we received 3.2k USD bug bounty rewards.

With respect to the type of these bugs (see the third column of Table 6.4), nine of them are buffer overflow, two of them are integer overflow, four of them are memory corruption, two of them are arbitrary address access, two of them are null pointer dereference, and five of them are assertion failure. Moreover, among these bugs, AFL only discovered six of them (see the last column of Table 6.4) and did not discover any other new bugs. This demonstrates that our approach significantly improves the bug finding capability of coverage-based greybox fuzzers, owing to the grammar-awareness in Superion. All these bugs except for the two bugs in Jerryscript have been fixed.

Comparison to AFL. Among these 34 bugs, AFL only discovered six of them (as shown in the fourth column of Table 6.4) and did not discover any other new bugs. This demonstrates that our approach significantly improves the bug finding capability of coverage-based grey-box fuzzers, which owes to the grammar-awareness in Superion.

Table 6.4: Unique Bugs Discovered by Superion

Program	Bug	Type	AFL	jsfunfuzz
libplist	CVE-2017-5545	Buffer Overflow	✗	N/A
	CVE-2017-5834	Buffer Overflow	✓	N/A
	CVE-2017-5835	Memory Corruption	✓	N/A
	CVE-2017-6435	Memory Corruption	✗	N/A
	CVE-2017-6436	Memory Corruption	✗	N/A
	CVE-2017-6437	Buffer Overflow	✓	N/A
	CVE-2017-6438	Buffer Overflow	✓	N/A
	CVE-2017-6439	Buffer Overflow	✗	N/A
	CVE-2017-6440	Memory Corruption	✗	N/A
	Bug-90	Assertion Failure	✗	N/A
	CVE-2017-7440	Integer Overflow	✓	N/A
WebKit	CVE-2017-7095	Arbitrary Access	✗	✗
	CVE-2018-4378	Use-After-Free	✗	✗
	CVE-2018-4392	Buffer Overflow	✗	✗
	CVE-2017-7102	Arbitrary Access	✗	✗
	CVE-2017-7107	Integer Overflow	✗	✗
	Bug-191058	Assertion Failure	✗	✗
	Bug-192464	Uninitialized Memory Read	✗	✗
	Bug-185645	Null Pointer Deref	✗	✗
	Bug-188917	Assertion Failure	✗	✗
	Bug-170989	Assertion Failure	✗	✗
	Bug-170990	Assertion Failure	✗	✗
	Bug-172346	Null Pointer Deref	✗	✗
	Bug-172957	Null Pointer Deref	✗	✗
	Bug-172963	Buffer Overflow	✗	✗
	Bug-173305	Assertion Failure	✗	✗
	Bug-173819	Assertion Failure	✗	✗
Jerryscript	CVE-2017-18212	Buffer Overflow	✗	N/A
	CVE-2018-11418	Buffer Overflow	✓	N/A
	CVE-2018-11419	Buffer Overflow	✗	N/A
	Bug-2238	Buffer Overflow	✗	N/A
ChakraCore	MSRC Case 48387	Buffer Overflow	✗	✗
	Bug-5533	Null Pointer Deref	✗	✗
	Bug-5532	Null Pointer Deref	✗	✗

Specifically, for relatively weakly-structured inputs such as XML, AFL itself found 5 bugs, while Superion not only found all these 5 bugs, but also found 6 more bugs than

AFL. Differently, for highly-structured inputs such as JavaScript, AFL barely found bugs. Only one bug about utf-8 encoding problem was found by AFL in Jerryscript. All other bugs in JavaScript engines were actually found by Superion's tree-based mutation. This further demonstrates the significance of injecting grammar-awareness into coverage-based grey-box fuzzers.

Comparison to jsfunfuzz. We also compared Superion with jsfunfuzz [34], which is a successful grammar-aware fuzzer specifically designed for testing JavaScript engines. jsfunfuzz can be used to fuzz WebKit and ChakraCore; but it fails to fuzz Jerryscript because its generated JavaScript inputs have many JavaScript features that are not supported by Jerryscript. After three months of fuzzing, jsfunfuzz only found hundreds of out-of-memory crashes in WebKit and ChakraCore, but failed to find any bugs (as indicated by the last column of Table 6.4). This is because jsfunfuzz uses manually-specified rules to express the grammar rules the generated inputs should satisfy. However, it is daunting, or even impossible to manually express all the required rules. Instead, Superion directly uses the grammar automatically during trimming and mutation.

In summary, Superion can significantly improve the bug-finding capability of coverage-based greybox fuzzers (e.g., we found 34 new bugs, among which we discovered 22 new vulnerabilities with CVE identifiers assigned).

6.3.3 Code Coverage (RQ2)

As empirically studied that 1% increase in code coverage can increase the percentage of found bugs by 0.92% [113]. Hence, apart from the bug-finding capability, we measured the code coverage achieved by fuzzing. The results are shown in Table 6.5, including the line coverage and function coverage of the target programs we fuzzed. In particular, we list the coverage achieved by initial seeds, AFL and Superion. The coverage was calculated using *afl-cov* [114].

Basic blocks based coverage is a current state-of-the-art metric for measuring program states. Basic blocks provide the best granularity than function coverage and instruction coverage. Measuring just function loses lots of information on what goes on inside.

Table 6.5: Code Coverage of the Target Programs

Program	Line Coverage (%)			Function Coverage (%)		
	Seeds	AFL	Superion	Seeds	AFL	Superion
libplist	33.3	50.8	68.9	27.5	32.6	40.8
WebKit	52.4	56.0	78.0	35.1	37.0	49.5
Jerryscript	81.3	84.0	88.2	76.0	77.1	78.2
ChakraCore	46.7	54.5	76.9	40.7	49.8	63.2

Recording specific instructions is generally redundant since all of them are guaranteed to execute within the same basic block.

For line coverage, the initial seeds covered 33.3% lines of libplist, 52.4% lines of WebKit, 81.3% lines of Jerryscript and 46.7% lines of ChakraCore. By fuzzing, AFL respectively increased their line coverage to 50.8%, 56.0%, 84.0% and 54.5%. On average, AFL further covered 7.9% of the code. Superion improved the line coverage to 68.9%, 78.0%, 88.2% and 76.9%, respectively; and it further covered 24.6% of the code on average. Overall, Superion outperformed AFL by 16.7% in line coverage, because the grammar-awareness in Superion carries the fuzzing exploration towards the application execution stage.

On the other hand, for function coverage, the initial seeds covered 44.8% functions on average, and AFL and Superion increased the function coverage to 49.1% and 57.9%, respectively. Generally, Superion outperformed AFL by 8.8% in function coverage due to its grammar-awareness.

In summary, Superion can significantly improve the code coverage of coverage-based grey-box fuzzers (e.g., 16.7% in line coverage and 8.8% in function coverage).

6.3.4 The effectiveness of Grammar-Aware Trimming (RQ3)

Table 6.6 compares the trimming ratio (i.e., the ratio of bytes trimmed from test inputs) and the grammar validity ratio (i.e., the ratio of test inputs that are grammar-valid after trimming) using the built-in trimming in AFL and the tree-based trimming in Superion. Numerically, for libplist, the built-in trimming in AFL trimmed out 21.7% of bytes in

Table 6.6: Comparison Results of Trimming Strategies

Program	Trimming Ratio (%)		Grammar Validity Ratio (%)	
	Built-In	Tree-Based	Built-In	Tree-Based
libplist	21.7	11.7	74.1	100
WebKit	10.6	7.6	86.4	100
Jerryscript	5.1	4.7	89.3	100
ChakraCore	12.7	11.3	83.7	100

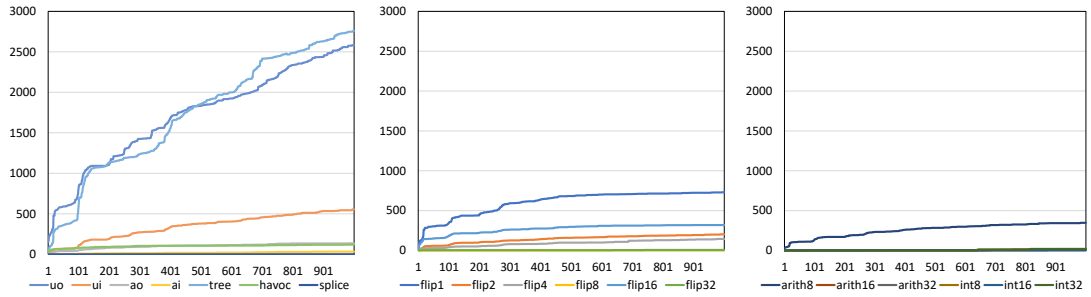


Figure 6.6: The Effectiveness of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage

XML test inputs on average, while our tree-based trimming trimmed out 11.7% on average. On the other hand, 74.1% of test inputs after the built-in trimming were grammar-valid, but 100% of test inputs after our tree-based trimming were grammar-valid and can be further used to conduct our grammar-aware mutation.

Similarly, the built-in trimming respectively trimmed out 10.6% and 5.1% of bytes in JavaScript test inputs for WebKit and Jerryscript, while our tree-based trimming respectively trimmed out 7.6% and 4.7% for WebKit and Jerryscript. However, our tree-based trim increased the grammar validity ratio for WebKit and Jerryscript from 86.4% and 89.3% to 100%, which can facilitate our grammar-aware mutation by improving the chance of applying grammar-aware mutation (which is more effective in generating test inputs that can trigger new coverage as will be discussed in Section 6.3.5).

In summary, although with a relatively low trimming ratio, our grammar-aware trimming strategy can significantly improve the grammar validity ratio for the test inputs after trimming, which facilitates our grammar-aware mutation.

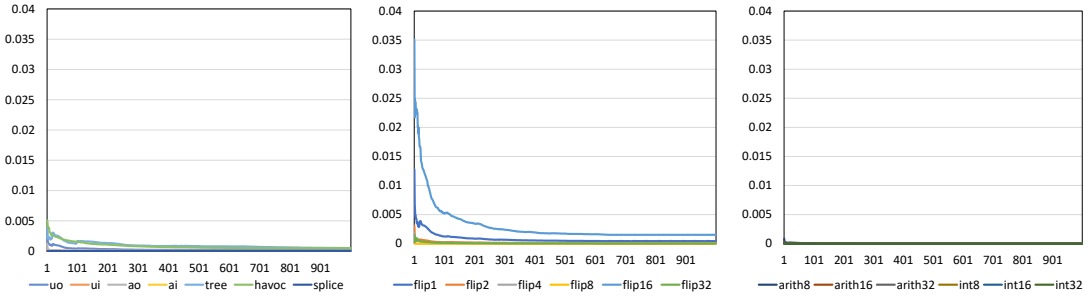


Figure 6.7: The Efficiency of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage

6.3.5 The effectiveness of Grammar-Aware Mutation (RQ4)

To evaluate the effectiveness of our grammar-aware mutation strategies, we compared them with those built-in mutation strategies of AFL [115], which include bit flips (*flip1* / *flip2* / *flip4* – one/two/four bit(s) flips), byte flips (*flip8*/*flip16*/*flip32* – one/two/four byte(s) flips), arithmetics (*arith8*/*arith16*/*arith32* – subtracting or adding small integers to 8-/16-/32-bit values), value overwrites (*interest8*/*interest16*/*interest32* – setting "interesting" 8-/16-/32-bit values to 8-/16-/32-bit values), *havoc* (random application of bit flips, byte flips, arithmetics, and value overwrite), and *splice* (splicing together two random test inputs from the queue, and then applying havoc). For the ease of presentation, our enhanced dictionary-based mutation strategy is referred as *ui* (insertion of user-supplied tokens), *uo* (overwrite with user-supplied tokens), *ai* (insertion of automatically extracted tokens), and *ao* (overwrite with automatically extracted tokens); and our tree-based mutation strategy is referred to as *tree*.

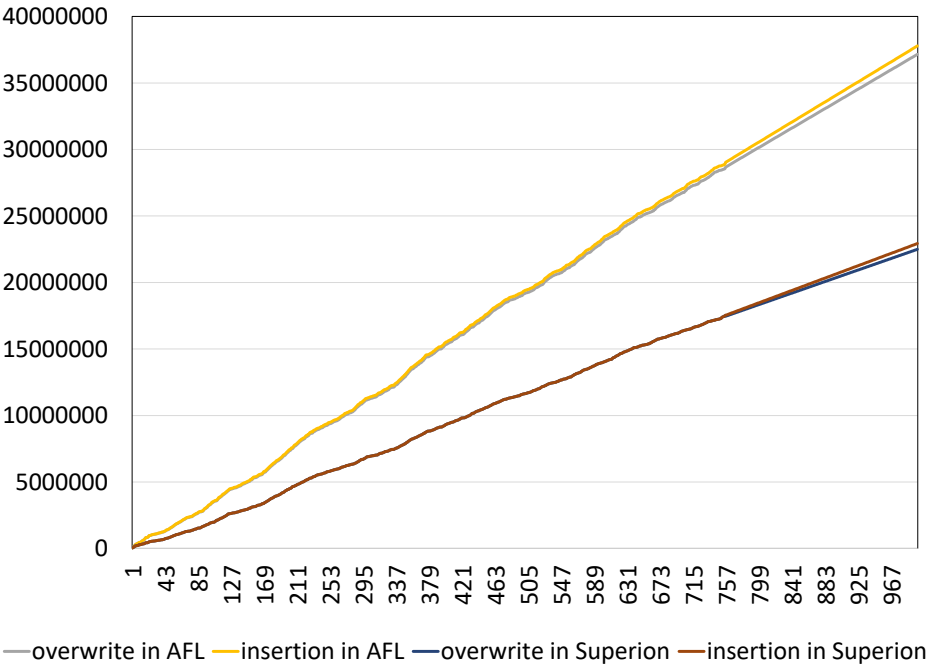
Figure 6.6 shows the number of interesting test inputs (i.e., triggering new coverage) discovered by different mutation strategies as we fuzzed WebKit. The x-axis denotes the number of test inputs that Superion sequentially took from the queue and processed, and the y-axis denotes the corresponding number of interesting test inputs produced by different mutation strategies. As the process of different test inputs often takes different time, we do not use the time to represent the x-axis. Besides, for clarity, Figure 6.6 omits the results when all the mutation strategies become ineffective in continuously producing interesting test inputs (i.e., when the curves in Figure 6.6 change gently).

The results vary across different seeds. Even with seeds fixed, the results may also vary across different runs due to the random nature of some mutation strategies (i.e., *havoc*, *splice*, and *tree*). However, the trend remains the same across runs, and we only discuss the trend which holds across runs. In the beginning, bit and byte flips take a leading position in producing interesting test inputs. The reasons are that i) bit and byte flips often destroy the input structure, and trigger previously unseen error handling paths; and ii) bit and byte flips are the first mutation strategy to be sequentially applied, thus having the opportunity to first trigger the new coverage that could also be triggered by other mutation strategies. Gradually, the number of interesting test inputs generated by our grammar-aware mutation strategies outperform other mutation strategies. Specifically, *tree* and *uo* significantly outperform other mutation strategies. These results indicate that grammar-aware mutation strategies are effective in producing interesting test inputs.

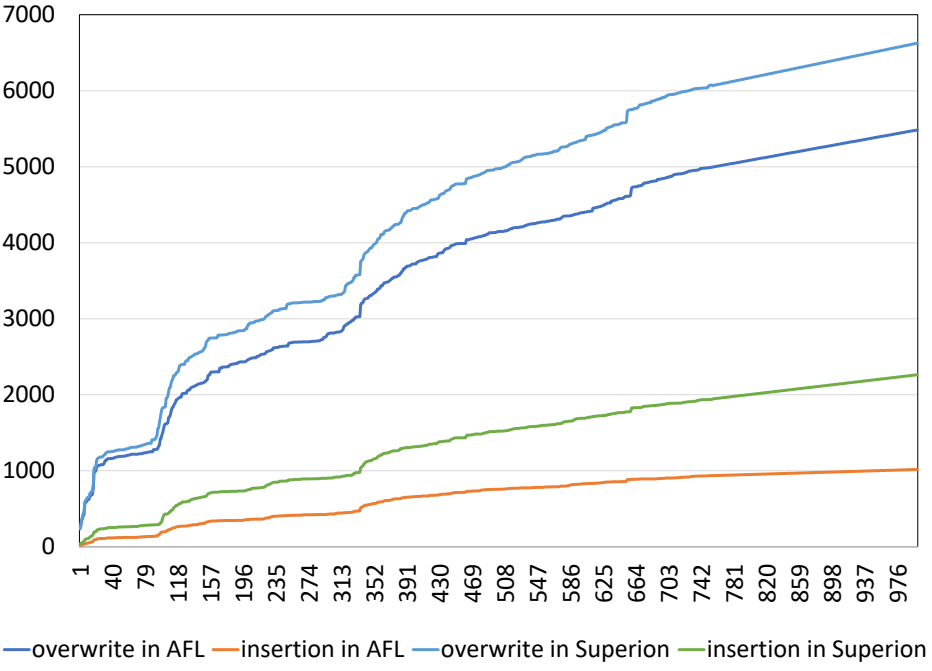
Besides, we also explore the efficiency of different mutation strategies in producing interesting test inputs. The results are shown in Figure 6.7, where the x-axis is the same to Figure 6.6 and the y-axis denotes the ratio of interesting test inputs to the total number of generated test inputs. Surprisingly, all the mutation strategies are very inefficient in producing interesting test inputs, i.e., only two of the 1,000 mutated test inputs can trigger new coverage. Thus, a huge amount of fuzzing efforts are wasted in mutating and executed test inputs. Therefore, adaptive mutation rather than exhaustive mutation should be designed to smartly apply mutation strategies.

Figure 6.9 and 6.11 show the number of interesting test inputs generated by different mutation strategies as we fuzzed Jerryscript and libplist, respectively. Figure 6.10 and 6.12 report the efficiency of different mutation strategies in producing interesting test inputs as we fuzzed Jerryscript and libplist, respectively. The results are similar to Figure 6.6 and 6.7, as discussed in Section 6.3.5.

Moreover, to evaluate our enhancement to dictionary-based mutation, we compared the dictionary overwrite and insertion in AFL with those in Superion. The results are reported in Figure 6.8, where the x-axis is the same to Figure 6.6, and the y-axis in Figure 6.8a and Figure 6.8b represent the number of times each mutation is applied



(a) The Number of Insertion and Overwrite Mutation Applications



(b) The Effectiveness in Producing Test Inputs Triggering New Coverage

Figure 6.8: Comparison Results of Dictionary-Based Mutations

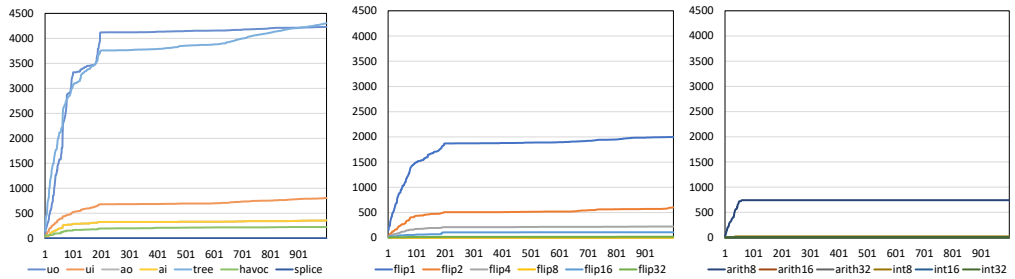


Figure 6.9: The Effectiveness of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage in Jerryscript

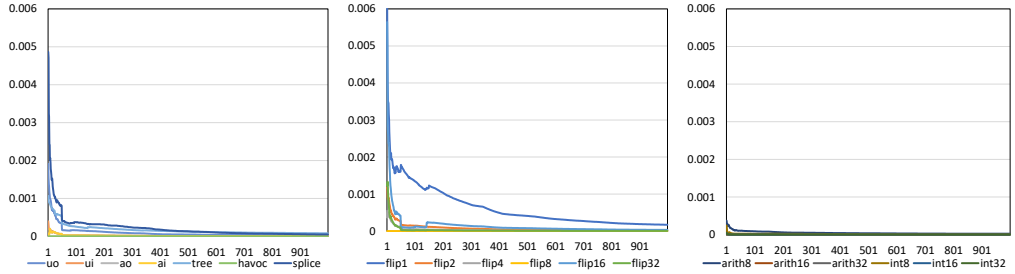


Figure 6.10: The Efficiency of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage in Jerryscript

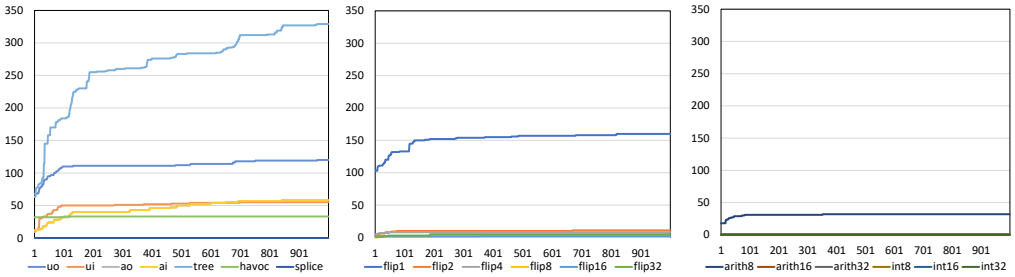


Figure 6.11: The Effectiveness of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage in libplist

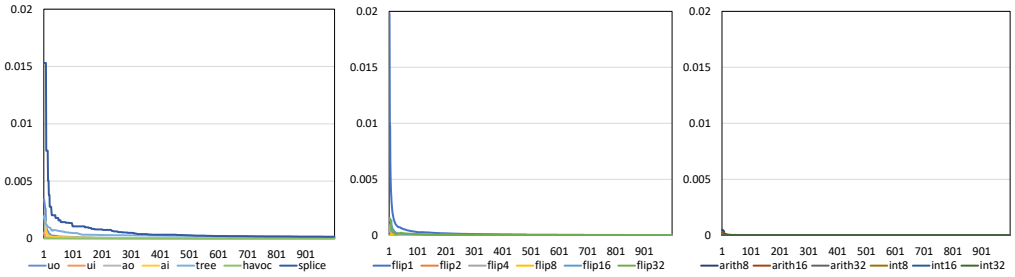


Figure 6.12: The Efficiency of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage in libplist

and the number of interesting test inputs generated. We can see that our enhanced dictionary-based mutation greatly decreases the number of mutation applications by

Table 6.7: Comparison Results of Fuzzing Stability

Program	w/o Selective Instrument.	Selective Instrument.
libplist	100%	100%
WebKit	43%	98%
Jerryscript	99.98%	99.98%

half, while still generating significantly more interesting test inputs.

In summary, our grammar-aware mutation strategies are effective in generating test inputs that can trigger new coverage, compared to the built-in mutation strategies in AFL. However, the efficiency of all mutation strategies need to be improved.

6.3.6 The effectiveness of Selective Instrumentation (RQ5)

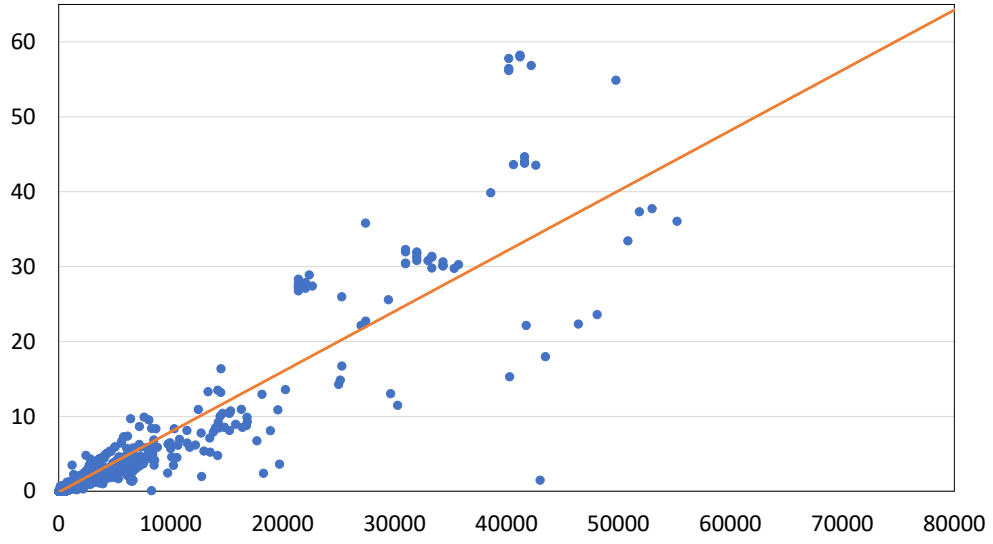
To evaluate the effectiveness of our selective instrumentation, we used the default stability metric in AFL as the indicator. The results are shown in Table 6.7. The first column lists the target programs, the second and third columns respectively report the stability in AFL (i.e., without selective instrumentation) and in Superion (i.e., with selective instrumentation).

The stability of libplist and Jerryscript in AFL were already 100% and 99.98%, and thus we did not need to apply selective instrumentation to them. For WebKit, the stability in AFL was quite low (i.e., 43%), which means that the fuzzing is almost a waste of time [18]. Instead, the stability in Superion increased to 98%. This indicates that we successfully identified most of the code that was responsible for low stability, and our selective instrumentation is effective in improving fuzzing stability.

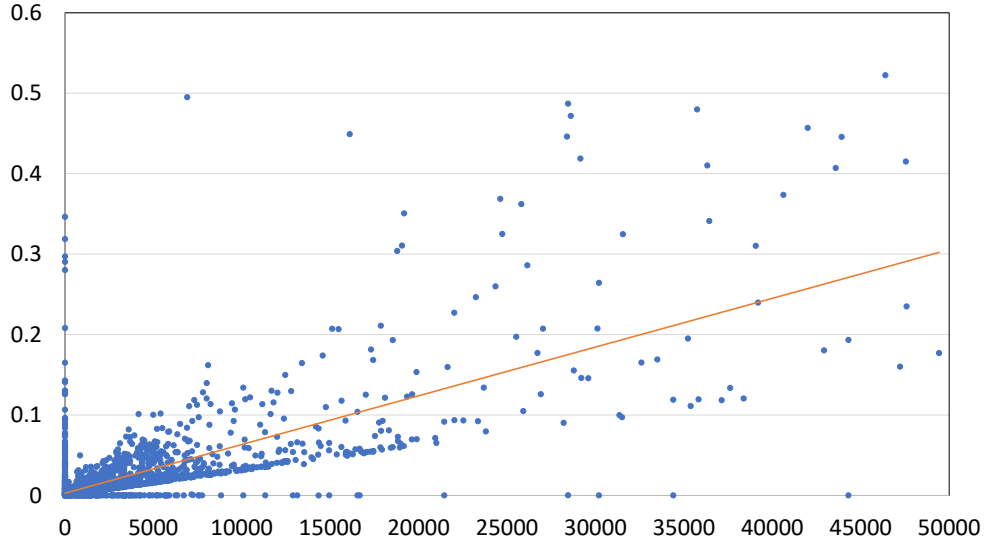
In summary, our selective instrumentation can greatly improve the fuzzing stability by excluding the source code that is responsible for low stability from instrumentation.

6.3.7 Performance Overhead (RQ6)

The fuzzing process of a test input includes three major steps: parsing, mutation, and execution. Among them, the parsing step is one-off for each test input, followed by



(a) JavaScript Test Inputs



(b) XML Test Inputs

Figure 6.13: The Time to Read, Parse and Traverse Test Inputs with Respect to Different Size

a large number of mutations and executions. In Figure 6.13a and 6.13b, we show the parsing time of JavaScript and XML test inputs in seconds (the y-axis) with respect to the size of test input files in bytes (the x-axis). Without loss of generality, we only report the results for the initial test inputs (see the last column in Table 6.2). In detail, the parsing time includes the time to read, parse and traverse a test input file. Generally, the parsing time is linearly correlated to the size of test input files. Most of the JavaScript test inputs' size is under 10 KB and their parsing time is under 10 seconds, and the parsing time of XML test inputs is under 0.045 seconds. Notice that the parser generated

Table 6.8: Performance Overhead on Target Programs

Program	Tree-Based Mutation (ms)	Execution (ms)
libplist	0.63	0.39
WebKit	5.65	12.50
Jerryscript	5.65	3.57
ChakraCore	5.65	20.00

using ANLTR is not optimized for the performance. We may reduce the execution time further by improving the parser's implementation.

Apart from the parsing time, the major performance overhead Superion imposes on mutation and execution is caused by our tree-based mutation. Table 6.8 reports the overhead of applying tree-based mutation (in the second column) as well as the corresponding overhead of executing the mutated test input (in the third column). For small projects like libplist, it is very fast to perform tree-based mutation and execution, i.e., the mutation took 0.63 ms and the execution took 0.39 ms on average. For large projects such as WebKit and Jerryscript, the execution took much more time; e.g., executing a JavaScript input on WebKit took 12.50 ms, while the mutation took 5.65 ms on average, and executing a JavaScript input on Jerryscript took 3.57 ms, and the mutations took 5.65 ms on average. Considering the improvements to bug-finding capability and code coverage, the performance overhead introduced by Superion is acceptable.

In summary, Superion introduces additional overhead due to our grammar-aware tree-based mutation strategy. However, such overhead is still acceptable considering the improved bug-finding capability and code coverage.

6.3.8 Case Study

The JavaScript code fragment in Figure 6.14 gives a representative test input that was generated by Superion and triggered an integer overflow vulnerability in WebKit, assigned CVE-2017-7107. In particular, this vulnerability is triggered because the method `setInput` in class `RegExpCachedResult` forgets to reify the `leftContext` and `rightContext` at the right time. As a result, when later WebKit attempts to reify them, it will end up using indices into an old input string to create a substring of a new

```
var str="ss";
var re=str.replace(/\b\w+\b/g);
RegExp.input="a";
RegExp.rightContext;
```

Figure 6.14: A Proof-of-Concept of CVE-2017-7107

```
JSString* RegExpCachedResult::rightContext(ExecState* exec, JSObject* owner)
{
  // Make sure we're reified.
  lastResult(exec, owner);
  if (!m_reifiedRightContext) {
   unsigned length = m_reifiedInput->length();
   m_reifiedRightContext.set(exec->vm(), owner, m_result.end != length ?
      jsSubstring(exec, m_reifiedInput.get(), m_result.end, length - m_result.end)
      : jsEmptyString(exec));
  }
  return m_reifiedRightContext.get();
}
```

Figure 6.15: The Vulnerable Code Fragment for CVE-2017-7107

```
...
var str = "ss"
var re=str.replace(/\b\w+\b/g);
...
```

Figure 6.16: Source Test Input 1 to Trigger CVE-2017-7107

```
...
write('RegExp.input: ' + RegExp.input);
...
write('RegExp.rightContext: ' + RegExp.rightContext);
...
```

Figure 6.17: Source Test Input 2 to Trigger CVE-2017-7107

input string. For the test input in Figure 6.14, WebKit tried to get a substring through `jsSubstring`, whose length is 1 (i.e., length of "a") - 2 (i.e., `m_result.end` of "ss") = -1, as shown in Figure 6.15, which is a very large number when treated as positive. Thus, an integer overflow vulnerability is caused.

The test input in Figure 6.14 was actually simplified from a large test input for the ease of presentation. It was generated by applying our tree-based mutation on the two test inputs in Figure 6.16 and Figure 6.17. This proof-of-concept was not generated

through one mutation, but was generated after several times of mutations. The intermediate test inputs that triggered new coverage was kept and added to the queue for further mutations. Eventually, it evolved into the proof-of-concept. This vulnerability was not triggered by AFL. This demonstrates that the built-in mutation strategies in AFL are not effective in fuzzing programs that process structured inputs, where our tree-based mutation becomes very effective.

6.4 Related Work

Instead of listing all related work, we focus our discussion on the most relevant fuzzing work in five aspects: guided mutation, grammar-based mutation, block-based generation, grammar-based generation, and fuzzing boosting.

6.4.1 Guided Mutation

Mutation-based fuzzing was introduced to generate new test inputs by randomly mutating well-formed test inputs [9]. Then, a large body of work has been developed to design heuristics to guide the mutation. Specifically, AFL [18] applies a novel type of compile-time instrumentation and generic algorithm to detect interesting test inputs that can trigger new coverage. Guided by such coverage information, AFL can improve the code coverage, and has been used to find thousands of high-impact vulnerabilities. Steelix [97] further incorporates the coverage progress information to break through magic-byte comparisons. CollAFL [116] improves the coverage accuracy to mitigate path collisions in AFL and proposes three seed selection strategies to improve the efficiency of AFL. SlowFuzz [117] uses resource usage as the feedback to guide the evolutionary search to automatically find test inputs that maximize computational resource utilization for a system. SemFuzz [118] automatically extracts vulnerability-related knowledge from CVE reports or git logs, and uses such information to guide the systematic construction of test inputs through fuzzing to trigger a known or related unknown flaw.

BuzzFuzz [19], Vuzzer [96] and Angora [119] leverage taint analysis to locate the interesting bytes that are more profitable for mutation. BuzzFuzz [19] first relies on dynamic taint analysis to locate the regions of original seed inputs that influence values used at key program attack points (points where the program may contain an error) and then mutates those parts to generate new inputs. Vuzzer [96] uses control-flow features based on static analysis to prioritize deep (and thus interesting) paths and deprioritize frequent (and thus uninteresting) paths when mutating inputs, and uses data-flow features through dynamic taint analysis to accurately determine where and how to mutate such inputs. Angora [119] proposes a scalable byte-level taint tracking technique to track which input bytes flow into each path constraint. Then Angora relies on context-sensitive branch coverage, gradient descent-based searching, and input length exploration to improve the fuzzing effectiveness and efficiency.

Symbolic execution is also utilized to facilitate fuzzing. For example, SAGE [22, 23] implements a novel search algorithm to maximize the number of test inputs generated from every run of symbolic execution. Specifically, all the constraints in a path condition are systematically negated one by one, conjuncted with the prefix of the path condition leading to it, and then solved through a constraint solver. Therefore, a single run of symbolic execution can generate multiple test inputs. Babić et al. [21] propose a three-stage processing approach to generate test inputs that can reach potential vulnerabilities in a target program. First, it performs dynamic analysis with a small set of test inputs to resolve indirect jumps in the binary code and builds a visibly pushdown automaton to reflect the global program control-flow. Second, it conducts static analysis on the inferred automaton to find potential vulnerabilities. Finally, it uses the result of prior phases to assign weights to automaton edges and then uses symbolic execution to generate test inputs and direct its exploration to the target potential vulnerabilities.

Hawkeye [120] precisely collects the information such as the call graph, function and basic block level distances to the targets. During fuzzing, Hawkeye evaluates exercised seeds based on both static information and the execution traces to generate the dynamic metrics, which are then used for seed prioritization, power scheduling and adaptive mutating. These strategies help Hawkeye to achieve better directedness and gravitate towards the target sites. Paper [121] proposes an automated fuzz testing framework for

hunting potential defects of general-purpose DNNs. It performs metamorphic mutation to generate new semantically preserved tests and leverages multiple pluggable coverage criteria as feedback to guide the test generation from different perspectives. FOT [**?**] propose a fuzzing framework, namely Fuzzing Orchestration Toolkit (FOT). FOT is designed to be versatile, configurable and extensible. With FOT and its extensions, authors have found 111 new bugs from 11 projects. Among these bugs, 18 CVEs were assigned. Paper [122] identifies parts of the source codes which are more likely to contain bugs through program metrics. The metrics they used include code complexity, number of pointer arithmetics, the number of nested control structures and so on.

Following the advances in leveraging taint analysis and symbolic execution to guide fuzzing, several approaches have been proposed to combine taint analysis and symbolic execution to guide fuzzing. In particular, TaintScope [20] can accurately locate checksum-based integrity checks by branch profiling techniques, and bypass such checks via control flow alteration techniques. Then, TaintScope can identify which bytes in a well-formed input are used in security-sensitive operations based on fine-grained dynamic taint tracing, and generate test inputs that are more likely to trigger potential vulnerabilities using combined concrete and symbolic execution techniques. Dowser [24] first uses taint analysis to determine the set of input bytes that influence the array index and then executes the program symbolically by making this set of inputs symbolic. By constantly steering the symbolic execution along branch outcomes most likely to lead to overflows, Dowser targets buffer overflow and underflow vulnerabilities. Similarly, BORG [25] targets buffer over-read bugs. It first relies on taint analysis to select buffer accesses that could lead to an over-read bug and then guides symbolic execution towards those accesses along program paths that could actually lead to an over-read.

Recently, Pham et al. [28] leverage information about the format and data chunks of existing, valid test inputs to identify a format constraint, which can be used to rule out most invalid inputs during path exploration in symbolic execution. As a result, the validity of the generated test inputs is ensured, which can help to swiftly carry the exploration beyond the parser code and improve the effectiveness of fuzzing. Driller [94] combines fuzzing and concolic execution in a complementary way to find deep bugs.

Fuzzing is used to exercise compartments of a target system, while concolic execution is used to generate test inputs that satisfy those complex checks separating the compartments. Whenever fuzzing is saturated and fails to trigger new program behaviors, Driller switches to concolic execution to touch those hard-to-reach branches and generate test inputs to break through them, then switches back to fuzzing.

Kargén and Shahmehri [123] perform mutations on the machine code of the generating programs instead of directly on a well-formed test input. In this way, they can leverage information about the input format encoded in the generating programs to produce high-coverage test inputs.

In summary, these fuzzing techniques target programs that process compact or unstructured inputs, which are less effective for programs that process structured inputs; i.e., most mutated test inputs will be rejected at an early stage of program execution, heavily limiting the capability of fuzzers to find deep bugs. Complementary to them, Superion is grammar-aware and can effectively fuzz programs that process structured inputs.

It is also worth mentioning that application-specific fuzzing techniques have been attracting great interests, e.g., compiler fuzzing [124–128], kernel fuzzing [129–131], and IoT (Internet of Things) fuzzing [132]. For example, Orion [125] profiles a program's test executions, and stochastically prunes its unexecuted code to generate mutated test inputs to fuzz compilers; Athena [128] improves Orion by an advanced mutation strategy that can also insert code in the unexecuted regions; and Hermes [127] further improves Athena to support the mutation on both dead and live code. kAFL [129] is a coverage-guided kernel fuzzer that is OS-independent and hardware-assisted. It relies on a hypervisor and Intel's Processor Trace technology to obtain control flow information from running code, and uses such information to construct the feedback to guide the fuzzer. IoTFuzzer [132] dynamically identifies the content inside the IoT app that forms the messages to be delivered to the target IoT device, and automatically mutates such content to use the app's program logic to produce meaningful test inputs. It is interesting to investigate how to extend our general-purpose fuzzer (e.g., designing new mutation operators or feedback mechanisms) to be effective in fuzzing compilers, kernels, or IoT devices.

6.4.2 Grammar-Based Mutation

Instead of blindly mutating structured test inputs, several advances have been made to perform mutations based on the grammar. Specifically, MongoDB [109] converts a set of JavaScript test inputs to ASTs and then wreaks controlled havoc on the AST by selectively replacing nodes, shuffling them around and replacing their values. While our tree-based mutation strategy is similar to the mutations in MongoDB, Superion conducts the mutations in an incremental way by keeping those interesting intermediate test inputs for further fuzzing. Besides, MongoDB is carefully designed for JavaScript test inputs, and it may take some efforts to fuzz other structured inputs.

LangFuzz [31] uses a given grammar to separate each test input in a corpus (e.g., a suite of test inputs that previously failed a target program) to code fragments and save them into a fragment pool. Then, some code fragments of a test input are mutated by replacing them with the same type of code fragments from the pool to generate a new test input. Similarly, IFuzzer [32] uses a language's context-free grammar to extract code fragments from a set of test inputs. Then it recomposes the code fragments in a biological evolutionary way to generate new test inputs. Superion differs from LangFuzz and IFuzzer by combining grammar-aware and grammar-blind mutations that are both useful for fuzzing, and also designing techniques to improve the effectiveness of trimming and stability of fuzzing.

u4SQLi [133] applies a set of mutation operators on valid SQL statements to produce syntactically correct and executable SQL statements that can reveal SQL vulnerabilities. It is specifically designed for SQL statements, while Superion is designed for fuzzing programs that process structured inputs.

6.4.3 Block-Based Generation

As some bytes in a test input are used collectively as a single value in the program, they should be considered together during fuzzing. Following this observation, Test-Miner [134] first extracts literals from a corpus of existing test inputs and then queries

the mined data for values suitable for a given method under test. These predicted values are then used as test inputs during test generation.

Spike [27] and Peach [26] use the input models, specifying the format of data chunks and integrity constraints, and combine mutation to generate test inputs. Specifically, Spike [27] applies a novel technique to represent and fuzz network protocols. Protocol data structures are broken down and represented as blocks. Such a block-based protocol representation allows for abstracted construction of various protocol layers. Peach [26] is a cross-platform fuzzing framework. It exposes a number of basic components for constructing new fuzzers, including generators, transformers, and protocols. An XML document needs to be created to describe the protocol or file format of test inputs. Based on such protocols, data ranging from simple strings to complex messages are generated and can be chained together to produce complex data types. Besides, data can be modified in a specific way (e.g., a base64 encoder, gzip and HTML encoding).

While being very effective in fuzzing programs that process simply-structured inputs (e.g., images and protocols), these approaches may become less effective for highly-structured inputs (e.g., XML and JavaScript). Complementary to them, Superion is specifically designed for highly-structured inputs.

6.4.4 Grammar-Based Generation

Another line of work is to use the grammar to directly generate test inputs. mangleme [33] is an automated broken HTML generator and browser fuzzer. It is originally used to find dozens of security and reliability problems in all major Web browsers. Jsfunfuzz [34] is one of the most popular fuzzing tools, finding more than 1,000 bugs in the Mozilla JavaScript engine. It uses specific knowledge about past and common vulnerabilities and hard-coded rules to generate new test inputs. Dewey et al. [35] propose to use constraint logic programming (CLP) for program generation. Through CLP, users can manually write declarative predicates to specify interesting program features, including syntactic features and semantic behaviors. Valotta [135] takes advantage of

his domain knowledge to manually build a fuzzer to test browsers for some newly-introduced features. While being effective in finding vulnerabilities, these approaches all rely on some hard-coded or manually-specified rules to express semantic rules in order to generate interesting test inputs. Such manual efforts hinder their applications to a wider audience.

Godefroid et al. [29] first leverage symbolic execution to generate grammar-based constraints, and then use grammar-based constraint solver to check their satisfiability and generate new test inputs. CSmith [30] generates C programs that cover a large subset of C and avoid those undefined and unspecified behaviors that could destroy the capability of fuzzers to find bugs. CSmith iteratively and randomly selects one production rule from all the allowable production rules in the grammar to generate C programs. Domato [136] generates a test input from scratch given a set of grammars that describe HTML/CSS structures as well as various JavaScript objects, properties, and functions. It gives each browser approximately 100,000,000 iterations with the fuzzer and records the crashes.

Radamsa [107] automatically builds a CFG to describe the structure of given training test inputs. Then the CFG is used to generate similar test inputs for robustness testing. Radamsa strikes a practical balance between completely random and manual test design and is proved to be very effective by finding hundreds of bugs. Skyfire [93] generates test inputs by first learning a PCSG from a large corpus of test inputs to capture both the context and probability of the production rules in the grammar and then favoring low-probability production rules to produce test inputs. Thus Skyfire can generate test inputs most of which can pass the syntax and semantic checking stage and focus on exploring the execution stage of target programs. Similar to Skyfire, TreeFuzz [137] also learns a probabilistic, generative model to generate new test inputs. These approaches are generation-based, while Superion is grammar-aware mutation-based, which utilizes the interesting behaviors embedded in previous interesting test inputs.

6.4.5 Fuzzing Boosting

Another thread of work focuses on improving the efficiency of current fuzzing approaches. Rebert et al. [17] empirically study how to pick seed test inputs to maximize the total number of bugs found during fuzzing. In total, they evaluate six different algorithms, and demonstrate that the choice of the algorithm has a great impact on the number of discovered bugs.

Householder and Foote [138] propose a machine learning-based algorithm to select two fuzzing parameters (i.e., the seed input and the proportion of the seed input for mutation). The goal is also to maximize the number of unique application errors discovered during a fuzzing campaign. Their algorithm greatly improves the efficiency of discovering unique application errors over basic parameter selection techniques.

Woo et al. [98] empirically investigate how to schedule the fuzzing of a given set of program-seed pairs. Their goal is also to maximize the number of unique bugs found within a limited time budget. Specifically, they build a mathematical model for blackbox mutational fuzzing and use the model to evaluate 26 existing and new randomized online scheduling algorithms. The results show that one of their new scheduling algorithms outperforms the state-of-the-art algorithm.

Cha et al. [139] propose a novel algorithm to dynamically tune the mutation ratio for maximizing the number of bugs found for blackbox mutational fuzzing. Given a program and a seed test input, they leverage symbolic analysis on the execution trace of a program-seed pair to first detect dependencies among the bit positions of test input and then use such a dependency relation to compute an optimal mutation ratio for this program-seed pair. The experimental results indicate an average of 38.6% more bugs than three previous fuzzers over eight programs within the same amount of fuzzing time budget.

AFLFast [95] boosts AFL by designing several strategies that can help to focus most of the fuzzing effort on low-frequency paths in order to explore more paths with the same amount of fuzzing time budget. Specifically, during fuzzing, it chooses the seed that

exercises lower frequency paths and has been chosen less often. Such strategies allow the fuzzer to fuzz the best seeds as early as possible.

Theisen et al. [140] propose an automated technique to approximate attack surfaces by analyzing stack traces. It hypothesizes that stack traces from user-initiated crashes have several desirable attributes for measuring attack surfaces. Such attack surfaces can be used to prioritize the fuzzing efforts on more profitable program locations. In this direction, AFLGo [141] is proposed to generate test inputs with the objective of reaching a given set of target program locations efficiently. Specifically, AFLGo applies a simulated annealing-based power schedule that can gradually assign more energy to test inputs that are closer to the target locations while reducing energy for test inputs that are further away.

Xu et al. [142] design three new operating primitives specialized for fuzzing with the goal of shortening the execution time of each iteration in fuzzing and achieving scalable performance on multi-core machines. Experimental results demonstrate great performance improvement in AFL.

Herfert et al. [143] combine tree transformations with delta debugging and a greedy backtracking algorithm to reduce tree-structured inputs. However, their goal of reduction is to reduce the test input as small as possible to facilitate crash analysis or delta debugging, while our trimming strategy is to retain the program behavior based on coverage information.

Bounimova et al. [144] propose SAGAN to monitor information from every fuzzing run in SAGE for further analysis, making it easy to drill down into the progress of a run. Besides, they develop JobCenter to control deployment of their whitebox fuzzing infrastructure (i.e., SAGE) across commodity virtual machines. These two techniques are developed to tackle the challenges in large-scale whitebox fuzzing in production.

These boosting techniques focus on different aspects of fuzzing and are orthogonal to Superion. We will investigate how to combine them with our approach to further improve the efficiency of Superion.

6.5 Conclusion

In this work, we propose a grammar-aware coverage-based greybox fuzzing approach, Superion, for programs that process structured inputs. In details, we propose a grammar-aware trimming strategy and two grammar-aware mutation strategies to effectively trim and mutate test inputs while keeping the input structure valid, quickly carrying the fuzzing exploration into width and depth. Our experimental study on several XML and JavaScript engines has demonstrated that Superion improved code coverage and bug-finding capability over AFL. Moreover, Superion found 34 new bugs, among which 22 new vulnerabilities were discovered.

7

Conclusions and Future Work

In this chapter, we will summarize the research work completed in this thesis and discuss the future research direction.

7.1 Summary of Completed Work

In this thesis, we completed four separate works. The first two are on JavaScript malware detection. The first one detects JavaScript malware mainly using machine learning approaches. Since we used HtmlUnit to expose the true JavaScript code behind various obfuscation, we reached a considerably good accuracy. On the other hand, due to the lightweightness of HtmlUnit, the overhead of JSDC is also acceptable. Different from this one, the second work we instrument true browser to actually execute the

JavaScript code and analyze their behaviors. Through execution, we learn models of eight commonly-seed JavaScript malware. The learning algorithm is adapted from famous L^* algorithm. The learned models can be used to detect the type of new JavaScript malware.

In detecting vulnerabilities work, we choose to improve fuzzing, which is considered one of the most successful testing techniques. Our Improvements are from two aspects. The first work we generate well-distributed seeds for fuzzing using knowledge hide in large-scale samples. The second work we improved fuzzing itself by proposing a novel grammar-aware mutation operation.

7.2 Future Work

Many improvements, experiments, and new ideas have been left for future work due to lack of time. The following ideas could be tested:

Improvement of Fuzzer. The current fuzzer has many inefficiencies on fuzzing complex applications. Although we identified and improved some of them, including stability, ineffectiveness on structured inputs, there are still lots of mutations and execution could be canceled. For example, not all replacing of subtrees are of equal effectiveness.

Just measuring code coverage is not a silver bullet by itself. There are still many code constructs which are impossible to cross with a dumb mutation-based fuzzing. For example, one instruction comparisons of types larger than a byte, especially with magic values, or many-byte comparisons performed in loops. Constant values and strings being compared against may be hard in a completely context-free fuzzing scenario but are easy to defeat when some program/format specific knowledge is considered. For example, both AFL and libFuzzer support dictionaries.

Combing Fuzzing and Program Analysis. The more we are familiar with the target programs, the better the fuzzing can be done. Since the target program is large and evolving, we need to turn to program analysis for help. Recently, symbolic execution and taint analysis techniques are getting maturer, which can improve fuzzing in a new

way. Although there are some tools which combine fuzzing and program analysis approaches, such as Driller, they can only run in specific platforms or target at relatively small programs. The good point is they have already proved their effectiveness in finding security vulnerabilities. This will be the new direction of Web-based security.

Extending the Fuzzing to Support More Grammars. We have proved our approach works very well with JavaScript, XML grammar. Therefore, we will extend the approach to support more grammars, such as PDF, PHP, even C/C++ in the future. It allows us to fuzz more programs such as PHP engines, C/C++ runtime and so on.

Bibliography

[1] Ali Ikinci, Thorsten Holz, and Felix C Freiling. Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Sicherheit*, volume 8, pages 407–421, 2008.

[2] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS*, 2006.

[3] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[4] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *the 20th USENIX conference on Security*, Berkeley, CA, USA, 2011.

[5] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *WWW*, pages 197–206, 2011.

[6] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[7] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[8] Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. Tzuyu: Learning stateful typestates. In *ASE*, pages 432–442, 2013.

[9] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[10] Sdl process: Verification. URL `https://www.microsoft.com/en-us/sdl/process/verification.aspx`.

[11] Chris Evans, Matt Moore, and Tavis Ormandy. Google online security blog – fuzzing at scale, 2011. URL `https://security.googleblog.com/2011/08/fuzzing-at-scale.html`.

[12] Brad Arkin. Adobe reader and acrobat security initiative, 2009. URL `http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html`.

[13] Kaspersky security bulletin 2013. `http://media.kaspersky.com/pdf/KSB_2013_EN.pdf`, 2013.

[14] Internet security threat report. `http://www.symantec.com/security_response/publications/threatreport.jsp`.

[15] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *CODASPY*, pages 117–128, 2013.

[16] Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators, Baltimore, Maryland, 2007.

[17] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security*, pages 861–875, 2014.

[18] American fuzzy lop. URL `http://lcamtuf.coredump.cx/afl/`.

[19] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, pages 474–484, 2009.

[20] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *SP*, pages 497–512, 2010.

[21] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22, 2011.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[24] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security*, pages 49–64, 2013.

[25] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *CODASPY*, pages 87–97, 2015.

[26] Peach fuzzer platform. URL `http://www.peachfuzzer.com/products/peach-platform/`.

[27] Spike fuzzer platform. URL `http://www.immunitysec.com/`.

[28] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *ASE*, pages 543–553, 2016.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294, 2011.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security*, pages 445–458, 2012.

[32] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *ESORICS*, pages 581–601, 2016.

[33] mangleme. URL http://freecode.com/projects/mangleme/.

[34] Jesse Ruderman. Introducing jsfunfuzz, 2007. URL http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz.

[35] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *ASE*, pages 725–730, 2014.

[36] Htmlunit. http://htmlunit.sourceforge.net/.

[37] Vxheaven. http://vxheavens.com/.

[38] Openmalware. http://http://oc.gtisc.gatech.edu:8080/.

[39] Gordon Mohr, Michael Stack, Igor Rnitovic, Dan Avery, and Michele Kimpton. Introduction to heritrix. In *4th International Web Archiving Workshop*, 2004.

[40] Web inspector: Website security with malware scan and pci compliance. http://www.webinspector.com/.

[41] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security*, pages 637–652, 2013.

[42] Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Mining interprocedural, data-oriented usage patterns in javascript web applications. In *ICSE*, pages 791–802, 2014.

[43] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.

[44] Malware with your mocha? obfuscation and anti-emulation tricks in malicious javascript. http://www.sophos.com/medialibrary/PDFs/technical-papers/malware_with_your_mocha.pdf, 2010.

[45] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and CheolWon Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *International Conference on Future Generation Information Technology*, pages 160–172, Berlin, Heidelberg, Germany, 2009.

[46] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *MALWARE*, pages 47–54, Montreal, QC, October 2009.

[47] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. " nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research*, 2011.

[48] J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Update Time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[49] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[50] MDN:XPCOM. `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM`, 2014.

[51] The heritrix web crawler project. `http://crawler.archive.org/`, 2007.

[52] Rhino. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino`, 2005.

[53] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[54] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10, New York, NY, USA, 2012. ACM.

[55] VirusTotal. `https://www.virustotal.com/`, 2014.

[56] Marco Cova, Christopher Krügel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW*, pages 281–290, 2010.

[57] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC*, pages 31–39, 2010.

[58] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA*, pages 122–132, 2012.

[59] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-Yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

[60] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6, 2012.

[61] The vulnerability of coolpreviews. `http://www.security-assessment.com/files/advisories/CoolPreviews_Firefox_Extension_Security_Advisory.pdf`, 2008.

[62] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 0-521-58519-8.

[63] Thierry Lavoie and Ettore Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. In *IWSC*, pages 1–7, 2012.

[64] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW*, pages 921–930, 2010.

[65] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *ICECCS*, pages 85–94, 2005.

[66] Martin Johns and Justus Winter. Protecting the intranet against javascript malware and related attacks. In *DIMVA*, pages 40–59, 2007.

[67] Mozilla Configurable Security Policies. `http://www-archive.mozilla.org/projects/security/components/ConfigPolicy.html`, 2009.

[68] Ping Zeng, Jianhua Sun, and Hao Chen. Insecure javascript detection and analysis with browser-enforced embedded rules. In *PDCAT*, pages 393–398, 2010.

[69] Mdn: Mozilla: Xulrunner. `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface`, 2014.

[70] Web Inspector. `http://www.webinspector.com/`, 2014.

[71] Alexa Top Sites. `http://www.alexa.com/topsites`.

[72] JS*: Malicious JavaScript Detection via Attack Behavior Modelling. `http://pat.sce.ntu.edu.sg/jsstar`, 2014.

[73] JSand on-line service. `https://wepawet.iseclab.org/index.php`, 2012.

[74] Clamav. `http://www.clamav.net/`, Jan, 2015.

[75] 2014 best antivirus software review. `http://anti-virus-software-review.toptenreviews.com/`, Sep, 2014.

[76] Hamid Abdul Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Trans. Software Eng.*, 35(4):497–514, 2009. doi: 10.1109/TSE.2009.16. URL `http://doi.ieeecomputersociety.org/10.1109/TSE.2009.16`.

[77] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *ICNP*, pages 114–123, 2008.

[78] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security*, pages 426–439, 2010.

[79] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, 2011.

[80] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 637–652, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL `http://dl.acm.org/citation.cfm?id=2534766.2534821`.

[81] Guozhu Meng, Matthew Patrick, Yinxing Xue, Yang Liu, and Jie Zhang. *Securing Android App Markets via Modelling and Predicting Malware Spread between Markets*, 2019.

[82] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *Empirical Software Engineering*, pages 1–53, 2017.

[83] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Trans. Information Forensics and Security*, 12 (7):1529–1544, 2017.

[84] Guozhu Meng, Yang Liu, Jie Zhang, Alexander Pokluda, and Raouf Boutaba. Collaborative security: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, 48(1):1, 2015.

[85] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 365–376. ACM, 2016.

[86] Sanjeev Das, Hao Xiao, Yang Liu, and Wei Zhang. Online malware defense using attack behavior model. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 1322–1325. IEEE, 2016.

[87] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*, 11(2):289–302, 2016.

[88] Yinzhi Cao, Vinod Yegneswaran, Phillip A. Porras, and Yan Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS*, 2012.

[89] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA*, pages 88–106, 2009.

[90] V. Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of javascript worms. In *USENIX Annual Technical Conference*, pages 335–348, 2008.

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

[92] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *EUROSEC*, page 4, 2011.

[93] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *SP*, pages 579–594, 2017.

[94] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[95] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *CCS*, pages 1032–1043, 2016.

[96] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[97] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *ESEC/FSE*, pages 627–637, 2017.

[98] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *CCS*, pages 511–522, 2013.

[99] Xml grammar. URL `https://github.com/antlr/grammars-v4/tree/master/xml`.

[100] Ginger Alliance. Sablotron, 2006. URL `http://freecode.com/projects/sablotron`.

[101] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[102] Antlr's grammar list for different languages. URL `https://github.com/antlr/grammars-v4`.

[103] Daniel Veillard. Libxslt – the xslt c library for gnome, 2003. URL http://xmlsoft.org/libxslt/.

[104] libxml2. URL http://www.xmlsoft.org/.

[105] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[106] Google patch reward program rules. URL https://www.google.com.au/intl/iw/about/appsecurity/patch-rewards/index.html.

[107] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with model inference assisted fuzzing. In *USENIX Security*, 2008.

[108] Michal Zalewski. afl-fuzz: making up grammar with a dictionary in hand. URL https://lcamtuf.blogspot.sg/2015/01/afl-fuzz-making-up-grammar-with.html.

[109] Robert Guo. Mongodb's javascript fuzzer. *Commun. ACM*, 60(5):43–47, 2017.

[110] gcov. URL https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[111] Christian Holler. Using aflfuzz with partial instrumentation, 2016. URL https://github.com/choller/afl/blob/master/docs/mozilla/partial_instrumentation.txt.

[112] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[113] Miller Charlie. Fuzz by number. In *CanSecWest Conference*, 2008.

[114] Michael Rash. afl-cov - afl fuzzing code coverage. URL https://github.com/mrash/afl-cov.

[115] Michal Zalewski. Mutation strategies in american fuzzy lop, 2013. URL http://lcamtuf.coredump.cx/afl/status_screen.txt.

[116] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *SP*, 2018.

[117] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS*, pages 2155–2168, 2017.

[118] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *CCS*, pages 2139–2154, 2017.

[119] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *SP*, 2018.

[120] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS*, 2018.

[121] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266*, 2018.

[122] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulenable code through program metrics. In *ICSE*, 2019.

[123] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *FSE*, pages 782–792, 2015.

[124] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *PLDI*, pages 197–208, 2013.

[125] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, pages 216–226, 2014.

[126] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76, 2015.

[127] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *OOPSLA*, pages 849–863, 2016.

[128] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, pages 386–399, 2015.

[129] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security*, pages 167–182, 2017.

[130] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *CCS*, pages 2345–2358, 2017.

[131] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *CCS*, pages 2123–2138, 2017.

[132] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[133] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: an input mutation approach. In *ISSTA*, pages 259–269, 2014.

[134] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying'hi!'is not enough: Mining inputs for effective test generation. In *ASE*, 2017.

[135] Rosario Valotta. Taking browsers fuzzing to the next (dom) level. In *DeepSec*, 2012.

[136] Ivan Fratric. The great dom fuzz-off of 2017, 2017. URL `https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html`.

[137] Jibesh Patra and Michael Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Technical Report TUD-CS-2016-14664, TU Darmstadt, 2016.

[138] Allen Householder and Jonathan Foote. Probability-based parameter selection for black-box fuzz testing. Technical Report CMU/SEI-2012-TN-019, Software Engineering Institute, Carnegie Mellon University, 2012.

[139] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *SP*, pages 725–741, 2015.

[140] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *ICSE*, volume 2, pages 199–208, 2015.

[141] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[142] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *CCS*, pages 2313–2328, 2017.

[143] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *ASE*, 2017.

[144] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ISSTA*, pages 122–131, 2013.