My Script Engines Know What You Did In the Dark: Converting Engines into Script API Tracers

Toshinori Usui

NTT Secure Platform Laboratories
Institute of Industrial Science,
The University of Tokyo
toshinori.usui.rt@hco.ntt.co.jp

Yuto Otsuki

NTT Secure Platform Laboratories
yuuto.otsuki.uh@hco.ntt.co.jp

Yuhei Kawakoya

NTT Secure Platform Laboratories
yuhei.kawakoya.sy@hco.ntt.co.jp

Makoto Iwamura

NTT Secure Platform Laboratories
makoto.iwamura.sw@hco.ntt.co.jp

Jun Miyoshi

NTT Secure Platform Laboratories
jun.miyoshi.fu@hco.ntt.co.jp

Kanta Matsuura

Institute of Industrial Science,
The University of Tokyo
kanta@iis.u-tokyo.ac.jp

ABSTRACT

Malicious scripts have been crucial attack vectors in recent attacks such as malware spam (malspam) and fileless malware. Since malicious scripts are generally obfuscated, statically analyzing them is difficult due to reflections. Therefore, dynamic analysis, which is not affected by obfuscation, is used for malicious script analysis. However, despite its wide adoption, some problems remain unsolved. Current designs of script analysis tools do not fulfill the following three requirements important for malicious script analysis. (1) Universally applicable to various script languages, (2) capable of outputting analysis logs that can precisely recover the behavior of malicious scripts, and (3) applicable to proprietary script engines.

In this paper, we propose a method for automatically generating script API tracer by analyzing the target script engine binaries. The method mine the knowledge of script engine internals that are required to append behavior analysis capability. This enables the addition of analysis functionalities to arbitrary script engines and generation of script API tracers that can fulfill the above requirements. Experimental results showed that we can apply this method for building malicious script analysis tools.

CCS CONCEPTS

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

KEYWORDS

malicious script, dynamic analysis, reverse engineering, function enhancement

ACM Reference Format:

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 *2019 Annual Computer Security Applications Conference (ACSAC'19)*, December 9–13, 2019, San Juan, PR, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3359789.3359849>

1 INTRODUCTION

The diversity of script languages creates a blind spot for malicious scripts to hide from analyses and detections. Attackers can flexibly choose a script language to develop a module of their malicious scripts and change scripts for developing another module of it. 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 gives 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 detections.

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

Another approach for malicious script analysis is focusing on a specific language and embedding monitoring mechanisms into a runtime environment of the script. This approach resolves the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7628-0/19/12...\$15.00

<https://doi.org/10.1145/3359789.3359849>

semantic gap problem mentioned above but requires deep domain knowledge to develop a monitoring tool. For example, we have to know both the specifications of a script language and the internal architecture of the script engine to develop a dynamic analysis tool for the script. In addition, this approach supports only a target script language. That is, we need to develop an analysis tool for each script language separately.

In summary, we (security side) need an approach universally applicable for any script language and fine-grained enough for analyzing the detailed behavior of a malicious script. However, previous studies satisfied both only one of these requirements not both at the same time.

To mitigate the gap between attackers and defenders, we propose a method of generating the 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 of 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 [7][31] to clarify the local functions corresponding to the script APIs which are usually acquired from the deep 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 four steps: execution trace logging, hook point detection, tap point detection, and script API tracer generation. The most important parts of our method are 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 functionality provided by a script engine. For example, each built-in function and statement of Visual Basic for Applications (VBA) and VBScript, such as *CreateObject* and *Eval*, and commandlets (Cmdlets) of PowerShell, such as *Invoke-Expression*, are script APIs.

A challenge in this research was efficiently finding the local function that corresponds to the target script API from the large number of local functions of a script engine. 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 calls the target script API different times. To achieve this differentiation, we modified the Smith-Waterman algorithm [27] 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 first requirement, i.e., universal applicability. However, we believe that our method allows us to reduce the cost of developing an analysis tool for each script language. Therefore, we can lower the bar for preparing analysis tools for any script language.

We implemented a prototype system that uses our method called *STAGER*, a script analyzer generator based on engine reversing, for evaluating the method. We conducted experiments using *STAGER* to detect hook and tap points on VBA, VBScript, and PowerShell. The experimental results showed that our method can precisely detect hook and tap points and generate a script API tracer that can output analysis logs containing script semantics. The hook and tap points are detected in about a few tens of seconds. Using the generated script API tracer, we analyzed real-world malicious scripts obtained from VirusTotal [1], a malware sharing service for research. The output logs showed that the generated script API tracer can 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 proposed a method that appends a script API tracing capability to a script engine by analyzing the script engine binaries.
- We confirmed that our method can accurately detect hook and tap points within realistic time through experiments. 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 effective for analysts by analyzing malicious scripts in the wild.

2 SCRIPT ANALYSIS TOOL

2.1 Requirements

We first clarify the three requirements that script analysis tools should fulfill from the viewpoint 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 by 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 by the proposed method outputs such logs.

2.2 Design and Problem

2.2.1 Script-level Monitoring. Design: Script-level monitoring directly inserts hooks into the target script itself. 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 a 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 construction method requires overriding of script APIs, it is only applicable to script languages that allow overriding of the built-in functions. Therefore, it does not fulfill the requirement of language independence mentioned in Section 2.1. This construction method is not sufficiently practical for malicious script analysis because few script languages support such a language feature.

Listing 1: Example of script-level monitoring

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

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

Problem: System-level monitoring causes 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. It occurs when one or more layers exist between an observation target and observation point. Ralf et al. [15] referred to the avalanche effect caused by the component object model (COM) layer, and we found that a script engine layer also causes the avalanche effect.

The main problem of semantics loss is that it decreases information effective for analysts. For example, Document.Cookie.Set, which has semantics of setting cookies in the script layer, loses the 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 preservability of script semantics mentioned in Section 2.1.

2.2.3 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

Table 1: Summary of requirements fulfillment with each construction method

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

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 script engine-level monitoring is its implementation difficulty. Although script engine-level monitoring may be easily achieved if a script engine provides interfaces for analysis such as Antimalware Scan Interface (AMSI) [22], this is just a limited example. In general, one has to discover appropriate hook and tap points for inserting hooks into the target program.

For open source script engines, we can insert hooks by analyzing the source code. However, it is limited to specific script languages whose source code is available. Even analyzing source code 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, script engine-level monitoring does not fulfill the requirement of binary applicability mentioned in Section 2.1.

2.3 Approach and Assumption

Table 1 summarizes how each design fulfills the requirements mentioned in Section 2.1. 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 improvement. The problem of the binary applicability of script engine-level monitoring will be solved if automatic reverse-engineering of script engines is enabled. Therefore, our approach automatically obtains 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 insider implementation of the target script engines. Therefore, no previous reverse-engineering of the target script engines is required.

3 METHOD

3.1 Overview

Figure 1 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 four steps: execution trace logging, hook point detection, tap point detection, 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 proposing 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. Using the obtained hook and tap points, the final step inserts hooks into the script engine and outputs it as the script API tracer.

Note that we define hook and tap points as follows.

- A hook point is the top 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.

Figure 1: Overview of our method

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

Listing 2: Example of Test Script of *CreateObject* in VBScript

```

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

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. 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 2.3.

3.3 Execution Trace Logging

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 an executed branch instruction 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 2. With this step, it is assumed that an execution trace with one script API call differs from another with multiple calls only in the limited part of the trace regarding the called script API.

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

This step uses multiple test scripts, i.e., one that calls the target script API once and the other(s) that calls it multiple times, as described in Section 3.2. 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 leverages the Smith-Waterman algorithm borrowed from bioinformatics. The

Figure 2: Concept of hook point detection by differential execution analysis.

Smith-Waterman algorithm performs local sequence alignment, which extracts a subsequence with high similarity from two or more sequences. However, we have a problem that it does not take into account the number of common subsequences that appeared; therefore we modified it to take this 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.

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

where

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

$$d = -1 \quad (3)$$

Our modified algorithm is the same as the original one up to filling all cells of the DP table. We provide an example of a DP table in Figure 3 for further explanation. A sequence of A, B, and C in this figure indicates one of the gray boxes in Figure 2. 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 gray boxes that appear between the white boxes as margins.

Although, these elements actually consist of multiple lines of branch logs, they are compressed as A, B, etc. for simplification. The original Smith-Waterman algorithm only finds the common subsequence of the highest similarity (SABC with the dotted rectangle in Figure 3) by backtracking from the cell with the local maximum score (the cell with score 8 in Figure 3). 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 Smith-Waterman algorithm (i.e., the dashed rounded rectangle in Figure 3). This is done by recursively applying the original Smith-Waterman algorithm to the same rows of the rest columns of the DP table as described in Figure 3.

The modified Smith-Waterman algorithm repeats this procedure N times to extract N common subsequences (the three dotted rounded rectangles in Figure 3). The algorithm then calculates the final common subsequence of the N subsequences and detects the branches constructing it as hook point candidates.

Algorithm 1 Our Modified Smith-Waterman Algorithm

Input: $seq1, seq2, N$

Output: $result_seq$

if $\text{len}(seq2) > \text{len}(seq1)$ **then**

$rows \leftarrow seq1$

$columns \leftarrow seq2$

else

$rows \leftarrow seq2$

$columns \leftarrow seq1$

end if

$common_seqs \leftarrow []$

for $i = 0$ **to** N **do**

$dptbl \leftarrow \text{DPTable}(rows, columns).fillCell()$

$j = 1$

repeat

$max_cell \leftarrow dptbl.searchJthLocalMaxCell(j)$

$max_seq \leftarrow dptbl.backtrackFrom(max_cell)$

$rows \leftarrow dptbl.getSameRows(max_seq)$

$columns \leftarrow dptbl.getRestColumns(max_seq)$

$j = j + 1$

until $\text{len}(columns) = 0$

$common_seqs.append(max_seq)$

end for

$result_seq \leftarrow \text{calcCommonSubseq}(common_seqs)$

3.5 Tap Point Detection

The tap point detection step plays two important roles. The first is to select the final hook points from the hook point candidates obtained in the previous step. The second is to find the memory regions that should be dumped into logs. The memory regions that would be dumped have two patterns: arguments and a return value of script APIs.

3.5.1 Argument. Tap point detection adopts a value-based approach for finding the matching value in the test script and script engine memory. If an argument value of script APIs in test scripts also appear in a specific memory location of the script engine process, the memory location is identified as a tap point.

Tap points detection for arguments of script APIs is carried out by exploring arguments of the functions detected as hook point candidates. To do so, the proposed method acquires the execution trace again with hooks inserted into the hook point candidates obtained in the previous step. Arguments of hook point candidates

Figure 3: Modified Smith-Waterman algorithm.

are available by referring to the memory location on the basis of the calling convention. Since type information (e.g., integer, string, structure pointer) of each argument is not available, further exploration requires heuristics.

Figure 4 illustrates the exploration heuristics used with the proposed method. First, if an argument of a hook point candidate is not dereferenced as a pointer, we regard it as a value. Otherwise, we regard and dereference it as a pointer. When regarding it as a value, we regard the value as various known types including known structures. As a result of this exploration, if the arguments in the test script are observed as the arguments at the hook point candidate, the proposed method regards the candidate as legitimate and defines the point of the argument as a tap point.

The exploration is improved if the type information is available. Therefore, the proposed method may more precisely explore by applying research of reverse-engineering type information such as TIE [19] or those of type prediction such as Debin [12] and TypeMiner [21].

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

Figure 4: Concept of tap point detection.

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 Script API Tracer Generation

We use the hook and tap points obtained above for appending script API trace capability to the target script engines. Our method hooks the local functions that correspond to the hook points and inserts analysis code. Note that a hook point indicates the top 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.

4 IMPLEMENTATION

For evaluating our method, we implemented a prototype system that uses our method called *STAGER*, a script analyzer generator based on engine reversing. *STAGER* uses Intel Pin [20] to insert hooks into the target script engine for acquiring execution traces. *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. In addition, *STAGER* 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)`

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:** How much false positives (FPs) and false negatives (FNs) does the script API tracer generated by *STAGER* (*STAGER*-generated tracer) produce?
- **RQ4:** Is the *STAGER*-generated tracer applicable to malicious scripts in the wild?
- **RQ5:** How well does the *STAGER*-generated tracer work compared with existing analysis tools?
- **RQ6:** 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 virtual machine (VM). One virtual CPU was assigned to this VM.

Although *STAGER* is more beneficial for proprietary script engines, we applied it to both open source and proprietary script engines. Open source engines are used because we can easily confirm the correctness of the hook and tap points. Note that the source 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 engines, we used VBScript implemented in ReactOS project [25] and PowerShell Core [29] which is a open source version of 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 to the Windows of the experimental environment VM because Intel Pin used by *STAGER* does not work properly on ReactOS.

For proprietary engines, we used Microsoft VBScript and VBA implemented in Microsoft Office. These script engines are 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

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

and tap point detection. VBA and VBScript is designed to use COM objects for interacting with the OS, instead of directly interacting with it. Therefore, malicious scripts using VBA and VBScript use script APIs related to COM object handling. In addition, VBA has useful script APIs and VBScript has those of reflection such as `Eval` and `Execute`, used for obfuscation. PowerShell has script APIs called `Cmdlets` that provide various functionalities including OS interaction. We selected `Cmdlets` of object creation, file operation, process execution, internet access, reflection, etc., which are often used by malicious scripts.

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 point were obtained. The *Log Availability* column has ✓ if the obtained hook and tap point output the correct log corresponding to the known scripts.

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

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

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

As shown in Table 3, *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, we confirmed that it works even for script engines with an additional layer such as CLI.

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 5 shows the results. Note that the execution time in this figure does not include the time for preparing test scripts because it should be manually created before the execution.

Execution trace logging and tap point detection required about 10 seconds due to the overhead of execution and log output with Intel Pin. 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 number of all script APIs in VBScript is less than one hundred according to the language specifications, and the script APIs of interest for malicious script analysis is limited. Therefore, the proposed method could quickly analyze script engines and generate a script API tracer, which is sufficient for practical use.

Figure 5: Execution duration of our method.

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 the investigation showed that the streams were ransomware such as

Dridex. From these results, we confirmed that the script API tracers generated by *STAGER* are applicable to real-world malicious scripts. Note that we will upload all our analysis logs after the publication of this paper. We selected two samples and their analysis logs as case studies. The first is a VBScript downloader, and the second is a PowerShell fileless malware module.

5.4.1 Case Study 1: Downloader. Figure 6 shows the analysis log of a VBS downloader generated by a script API tracer. Although this malicious script has 1500+ lines of obfuscated code, the log consists of only 16 lines, which are responsible for the main behavior of downloading. Section (1) in the figure shows a part of the log in which the malicious script accessed a URL. Section (2) shows that the script saved the HTTP response to a specific file in the Temp folder. The saved buffer is also visible as a byte array of 0x3c 0x68 Section (3) shows that the saved file was executed through *cmd.exe*. As shown in this figure, the script API tracers generated by *STAGER* could successfully extract important indicators of compromise (IOCs) such as URLs, binaries, file paths and executed commands. Note that the log fulfills the requirement of the preservability of semantics mentioned in Section 2.1.

```

CreateObject Microsoft.XMLHTTP
CreateObject Adodb.stream
CreateObject Wscript.shell
CreateObject Scripting.FileSystemObject
CreateObject WScript.Shell
Invoke Scripting.FileSystemObject.GetSpecialFolder(Param1=0x0002)
Invoke Microsoft.XMLHTTP.Open(Param1="GET", Param2="http://[anonymized by
author]/JHGcd476334?", Param3=False)
Invoke Microsoft.XMLHTTP.Send()
Invoke Microsoft.XMLHTTP.Status()
Invoke WScript.Shell.Type(Param1=Empty, Param2=0x0001)
Invoke WScript.Shell.Open()
Invoke MSXML.XMLHTTP.responseBody()
Invoke WScript.Shell.Write(Param1=<ARRAY:1*25>[3c 68 [snipped by author] 3e 0a])
Invoke WScript.Shell.SavetoFile(Param1="C:\Users\%[anonymized by
author]\AppData\Local\Temp\%GdiUvGoKq.exe", Param2=0x0002)
Invoke WScript.Shell.Run(Param1="cmd.exe /c call "C:\Users\%[anonymized by
author]\AppData\Local\Temp\%GdiUvGoKq.exe"")

```

Figure 6: Analysis log of VBS downloader with script API tracer

5.4.2 Case Study 2: Fileless Malware. Figure 7 shows an excerpt of the analysis log of a module used by PowerShell fileless malware. This module seems to retrieve additional PowerShell modules from the C&C server and execute it. Section (1) in this figure shows the spawn of a new PowerShell process with commands used for Web access. We can see the executed command with 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 to understand the behavior 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.

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

Figure 7: Analysis log of PowerShell fileless malware with script API tracer

We know this experiment can 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 by manual analysis. The result of the experiment showed that the hook and tap points produced neither FPs nor FNs.

5.6 Comparison with Existing Tracers

To answer RQ5, we compared a *STAGER*-generated script API tracer with two existing tracers: API Monitor [4] and ViperMonkey [18]. API Monitor is a system API tracer based on system-level monitoring. We enabled all system API hooks of API Monitor and made API Monitor observe the target script engine process. ViperMonkey is

a script API tracer for VBA based on script-level monitoring that is based on the emulator implemented for itself.

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

Table 4 shows the experimental results. Note that columns of observed behavior and log lines of ViperMonkey are calculated only with the samples that are 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 2.2.2 occurred.

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

Table 4: Comparison with existing tracers

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

Table 5: Lines of Code (LOC) of test scripts

Script languages	Average LOC
VBA	3.8
VBScript	2.75
PowerShell	2

5.7 Human Effort

To answer RQ6, we conducted an experiment to evaluate the amount of human effort required to prepare test scripts. We evaluated this from two viewpoints: lines of code (LOC) of test scripts and required time to create them.

We gathered ten people (eight graduate students, one technical staff member, and one visiting researcher) belonging to the computer science department as the participants of this experiment. We then explained the concept and requirements of the test scripts described in Section 3.2 to them. We asked them to write valid test scripts while measuring the required time. Many did not have experience on 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 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 8 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 showed that writing valid test scripts takes less time for programmers who have knowledge on the target script language. Therefore, the amount of human effort required for using *STAGER* is much less than manual reverse-engineering of script engines since manual reverse-engineering requires weeks of or months of analysis time.

6 DISCUSSION

6.1 Limitations

We discuss three cases in which the proposed 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, detection fails in principal if this matching is not available.

Figure 8: Required Time for Test Script Preparation

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, as far as we know.

6.2 Human-assisted Analysis

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

Note that the burden of manual analysis with our method is much less than complete manual analysis. This is because the number of functions that should be analyzed becomes much less by hook point detection, as described in Table 3. 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 [24] hooks JavaScript APIs by overriding built-in functions. It inserts hooks on open/send methods of XMLHttpRequest objects and methods regarding HTML-Element.prototype to obtain URLs accessed by Ajax communication. Practical script analysis tools such as Revelo [16], box-js [6], jsunpack-n [11]k and JSDetox [28] 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 2.1 because they deeply depend on the language specifications of JavaScript.

There are also script analysis tools based on script engine-level monitoring. Sulo [14][13] 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 [30] modifies an open source script engine of Flash for their hook. These are examples of script engine-level monitoring. ViperMonkey [18] 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 are available [17][23]. However, all these tools in this section cannot fulfill the requirements introduced in Section 2.1.

7.2 Script Engine Enhancement

Chef [5][3] is a symbolic execution engine for script languages. It uses a real script engine for building a symbolic execution engine. It achieves symbolic execution of the target scripts by symbolically executing the script engine binaries with a specific path exploration strategy. The design is similar to that of *STAGER* in that it reuses the target script engine for building a script analysis tool by instrumentation. On the other hand, the approaches and goals with Chef are different from those of *STAGER*. Its approach was 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 Mitigation of Semantic Gap

Several methods were developed for mitigating the semantic gap between the guest and host OSes on the virtual machine monitor (VMM).

Virtuoso [10] automatically creates VM introspection tools that can produce the same results as a reference tool executed in a VM from out-of-VM. Virtuoso first acquires execution traces by executing the reference tool in the VM. 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.

Tappan Zee (North) Bridge [9], or TZB, discovers tap points effective for VM introspection. TZB 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 introspection.

7.4 Reverse Engineering of Virtual Machine

Sharif et al. [26] 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 analyzes VMs to identify the information about VMs and bytecode, whereas our method analyzes to detect the local functions and their arguments that relate to script APIs.

Coogan et al. [8] proposed an approach to identify the bytecode instructions responsible for invoking system calls. Since system calls are strongly relevant to malware behavior, their goal is to approximate the behavior by the set of the identified bytecode instructions regarding the invocation of the system calls. Their goal, focus, and approach are different from ours. First, their goal is 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 is only the bytecode instructions relevant to the invocation of system calls, whereas ours is all script APIs regardless of the existence of system calls. Last, their approach strongly relies on the invoked system calls and arguments, whereas ours only on the branch instructions logged with test scripts.

7.5 Differential Execution Analysis

Carmony et al. [7] 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 enables the extraction JavaScript as well as those that represent the termination and error of input file processing.

Zhu et al. [31] 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., an identification of the commonly appeared sequences).

8 CONCLUSION

In this paper, we focused on the problems of current script dynamic analysis tools and proposed a method for automatically building 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 building script API tracers. In addition, our case studies showed that the generated script API tracers can analyze malicious scripts in the wild. Appending more effective script behavior 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. This work was partially supported by JSPS KAKENHI Grant Number JP17KT0081.

REFERENCES

- [1] [n. d.]. VirusTotal. <https://www.virustotal.com/>. (accessed: 2017-03-09).
- [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] The Dependable Systems Lab at EPFL in Lausanne. [n. d.]. Chef. <https://github.com/S2E/s2e-old/tree/chef>. (accessed: 2018-01-01).
- [4] Rohitab Batra. [n. d.]. API Monitor. <http://www.rohitab.com/apimonitor>. (accessed: 2019-02-15).
- [5] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 239–254.
- [6] CapacitorSet. [n. d.]. box.js. <https://github.com/CapacitorSet/box-js>. (accessed: 2019-02-15).
- [7] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasishet 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Blake Hartstein. [n. d.]. jsunpack-n. <https://github.com/urule99/jsunpack-n>. (accessed: 2019-02-15).
- [12] 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.
- [13] Timo Hirvonen. [n. d.]. Sulo. <https://github.com/F-Secure/Sulo>. (accessed: 2019-02-15).
- [14] Timo Hirvonen. 2014. Dynamic Flash instrumentation for fun and profit. Blackhat USA briefings 2014, <https://www.blackhat.com/docs/us-14/materials/us-14-Hirvonen-Dynamic-Flash-Instrumentation-For-Fun-And-Profit.pdf>. (accessed: 2019-02-15).
- [15] Ralf Hund. 2016. The beast within - Evading dynamic malware analysis using Microsoft COM. Blackhat USA briefings 2016.
- [16] KahuSecurity. [n. d.]. Revelo Javascript Deobfuscator. http://www.kahusecurity.com/posts/revelo_javascript_deobfuscator.html. (accessed: 2019-02-15).
- [17] 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 '13)*. Springer, 123–143.
- [18] Philippe Lagadec. [n. d.]. ViperMonkey. <https://github.com/decalage2/ViperMonkey>. (accessed: 2019-09-20).
- [19] 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.
- [20] 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.
- [21] 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.
- [22] Microsoft. [n. d.]. Antimalware Scan Interface. <https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal>. (accessed: 2018-08-16).
- [23] 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 2015, IMECS 2015*. Newswood Limited.
- [24] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jak: 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.
- [25] ReactOS Project. [n. d.]. ReactOS. <https://www.reactos.org/>. (accessed: 2018-08-16).
- [26] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE, 94–109.
- [27] Temple F Smith, Michael S Waterman, et al. 1981. Identification of Common Molecular Subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.
- [28] T. Sven. [n. d.]. JSDetox. <http://relentless-coding.org/projects/jsdetox/>. (accessed: 2019-09-20).
- [29] PowerShell Team. [n. d.]. PowerShell. <https://github.com/powershell>. (accessed: 2018-08-16).
- [30] 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.
- [31] 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.