IEICE TRANS. INF. & SYST., VOL.E103–D, NO.7 JULY 2020

PAPER Special Section on Information and Communication System Security

ROPminer: Learning-Based Static Detection of ROP Chain Considering Linkability of ROP Gadgets

Toshinori USUI[†,††a)], **Tomonori IKUSE**[†], **Yuto OTSUKI**[†*], ***Nonmembers***, **Yuhei KAWAKOYA**[†], **Makoto IWAMURA**[†], **Jun MIYOSHI**[†], ***Members***, ***and*** **Kanta MATSUURA**[††], ***Senior Member***

SUMMARY Return-oriented programming (ROP) has been crucial for attackers to evade the security mechanisms of recent operating systems. Although existing ROP detection approaches mainly focus on host-based intrusion detection systems (HIDSes), network-based intrusion detection systems (NIDSes) are also desired to protect various hosts including IoT devices on the network. However, existing approaches are not enough for network-level protection due to two problems: (1) Dynamic approaches take the time with second- or minute-order on average for inspection. For applying to NIDSes, millisecond-order is required to achieve near real time detection. (2) Static approaches generate false positives because they use heuristic patterns. For applying to NIDSes, false positives should be minimized to suppress false alarms. In this paper, we propose a method for statically detecting ROP chains in malicious data by learning the target libraries (i.e., the libraries that are used for ROP gadgets). Our method accelerates its inspection by exhaustively collecting feasible ROP gadgets in the target libraries and learning them separated from the inspection step. In addition, we reduce false positives inevitable for existing static inspection by statically verifying whether a suspicious byte sequence can link properly when they are executed as a ROP chain. Experimental results showed that our method has achieved millisecond-order ROP chain detection with high precision.

key words: *return-oriented programming, static detection, machine learning*

1. Introduction

Return-oriented programming (ROP) [1] is an attack technique used to bypass protection mechanisms of operating systems (OSes) such as no-execute bit (NX bit), which disables malicious code injected into a writable section to be run in the host. To detect the attacks using ROP (ROP attacks), there are mainly two approaches. One is a dynamic approach by running the code containing ROP in a real or virtualized host environment and monitoring the feasibility of its execution. The other is a static approach involved in detecting statistical features or specific patterns of the attack code for ROP (ROP chain), such as the frequency of appearances of specific byte sequences.

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

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

Our method is composed of two phases: offline learning and online detection. In the learning phase, our method learns the order of ROP components and the byte patterns of each component. ROP components are an element comprising a ROP payload and they are categorized into three types; pointer-type, constant value, and junk data. Our method learns the order of these components from real-world ROP chains and builds a model for detection with hidden Markov models (HMMs). In the online detection phase, we deploy the built model to an edge of the network for protection. To inspect arrival packets, we make a copy of network packets and pass them to the model to check whether the packets contain ROP payloads. If we detect a ROP payload in packets, we simply discard them. Otherwise, we pass them to the network to deliver.

A challenge in this paper is that we have to handle accidentally appeared benign byte sequences that have similar byte patterns to ROP chains. Since these byte sequences confuse static detectors and produce false positives, they should be clearly identified as benign. However, it is difficult to identify them as benign only with static byte patterns since their byte patterns are similar to those of ROP chains. To overcome this difficulty, we use a feature of dynamic linking between two ROP gadgets in addition to static byte patterns. Because verifying this dynamic feature of

Manuscript received August 31, 2019.
Manuscript revised January 8, 2020.
Manuscript publicized April 7, 2020.

[†]The authors are with NTT Secure Platform Laboratories, Musashino-shi, 180–8585 Japan.
[††]The authors are with Institute of Industrial Science, The University of Tokyo, Tokyo, 153–8505 Japan.
[*]Presently, the author is with NTT Security (Japan) KK, Tokyo, 101–0021 Japan.
a) E-mail: toshinori.usui.rt@hco.ntt.co.jp

DOI: 10.1587/transinf.2019ICP0016

Copyright © 2020 The Institute of Electronics, Information and Communication Engineers

ROP gadgets with a dynamic manner produces prohibitive runtime overhead, we managed to achieve this in a static manner at the online detection phase. Our method executes all ROP gadget candidates in the libraries that are used for ROP gadgets (*ROP target libraries*) and creates a dictionary that contains ROP gadget addresses and the corresponding offsets that indicates how much the stack pointer gains if the ROP gadget is executed (stack offset). This is done at the offline learning phase and enables to verify the dynamic linking feature in a static manner at the online detection phase.

We have implemented a prototype system which is based on our method called *ROPminer*. ROPminer is instantiated for detecting malicious Microsoft (MS) Office documents that are transferred on the network. This is because these malicious documents are one of the major attack vectors in the recent ROP-based exploits [2], [3]. We have tested ROPminer with real-world datasets of malicious and benign documents. The experimental results showed that ROPminer can detect ROP-based malicious documents with no false negatives and 3% false positives at 0.96 s/file on average.

We have achieved millisecond-order ROP chain detection with ROPminer. However, our method still has two limitations. The first is JIT-ROP, which dynamically crafts ROP chains in memory of the target host with the result of memory disclosure exploits to defeat address space layout randomization (ASLR). The second is encrypted communications. Note that these limitations are common among static detection methods and NIDSes. Even though there are these limitations, we argue that our method is still valuable for sharing among the security community. This is because we can handle these limitations by using our static detector combined with dynamic analysis like the existing methods [4], [5], and deployed with an SSL decryption gateway.

Our contributions are summarized as follows.

- We present a method for statically detecting ROP chains by learning the orders of ROP components and the byte patterns of the each component.
- We applied dynamic verification of the linkability of ROP gadgets to static detection by using the precalculation of stack offsets.
- We implemented ROPminer, a prototype system with our method, and evaluated its effectiveness on 1,067 malicious samples and 1,391 benign files used in the real world.

The rest of this paper is organized as follows. The ROP attack mechanism and byte-level characteristics of ROP chains are explained in Sect. 2. Our method is described in detail in Sect. 3. The implementation details of ROPminer are presented in Sect. 4. The evaluation of ROPminer is shown in Sect. 5. The discussion of the method is presented in Sect. 6. Section 7 discusses related publications that are not discussed well in the previous sections. Finally, Sect. 8 concludes the paper.

2. Return-Oriented Programming

2.1 Mechanism

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

More detailed explanations of the ROP mechanism is available in the existing papers [1], [6].

2.2 Byte-Level Characteristics

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

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

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

Listing 1: Example of ROP chain seen in the wild

```
0x275de6ae                JUNK
JUNK                      0x275e0861
JUNK                      JUNK
0x27594a2c                JUNK
0x2758b042                JUNK
0x2761bdea                JUNK
0x275811c8                0x275ebac1
0x2760ea66                0x275e0327
0x275e0081                JUNK
0x40000000                0x40000000
0x00100000                0x275ceb04
0x00003000                JUNK
0x00000040                JUNK
0x00001000                JUNK
0x275fbcfc                JUNK
[To the upper right]      0x40000040
```

stack pointer indicates and has no significance in its values. Therefore, attackers can use arbitrary values for it, and it has no characteristics in its byte patterns.

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

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

Listing 1 shows an example of a real-world 32-bit ROP chain. This ROP chain uses *MSCOMCTL.OCX*, which is always loaded at the address of 0x27580000 and has the size of 0x90000, as a ROP gadget library. Therefore, the ROP gadget addresses in the ROP chain are in the range of 0x27580000-0x2761FFFF. The ROP chain has several constant values that are multiples of the page size (0x1000) that often indicates a memory address or size (e.g., 0x40000000, 0x00100000, and 0x00001000 in the Listing 1. In addition, the ROP chain has the symbolic constants of *MEM_COMMIT* | *MEM_RESERVE* (0x00003000) and *PAGE_EXECUTE_READWRITE* (0x00000040). Note that *JUNK* in Listing 1 means that attackers can put arbitrary four bytes here.

3. Method

3.1 Overview

We first provide assumptions of our method. Since our method uses byte-by-byte static analysis, we assume that ROP chains are visible in the byte sequence of the inspection target. This assumption is the same as existing static detectors [7]–[9]. Therefore, when handling compressed or encrypted data, our method requires a decompression or decryption process dependent on the format of the target file. In addition, some ROP attacks dynamically generate ROP chains only in memory using scripts. To detect these ROP attacks, we have to execute the scripts and analyze the memory region which contains the generated ROP chains. We call these procedures to expose the ROP chains as *preprocessing*. The details of the preprocessing are described in Sect. 4.

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

Figure 1 shows an overview of our method. The method consists of two phases: offline learning and online detection. The offline learning phase is composed of three steps: preprocessing, gadget exploration, and model learning. The online inspection phase is composed of four steps: preprocessing, model update, probability calculation, and likelihood ratio test. Note that the preprocessing step is commonly performed in both two phases. We input known malicious and benign byte sequences with ground-truth labels in the learning phase, as well as ROP gadget libraries used by embedded ROP chains. After the learning phase, our method outputs learned models. In the inspection phase, suspicious byte sequences will be input to our method with the learned models, and our method then outputs the result of the inspection as malicious or benign.

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

3.2 Preprocessing

To start the whole learning and detection procedure, our method preprocesses the input data. The purpose of this preprocessing is to extract the byte sequence of learning/detection targets from the input data. The preprocessing consists of three steps, protocol/file format identification, format-dependent parsing, and gadget library identification. Note that this preprocessing is performed before both learning and detection. The first step is to identify the protocol/file format of all the input data. This is achieved in a generic manner such as finding magic numbers and parsing headers. The second step is to conduct format dependent parsing. This is done only when the input data requires the step to be judged from its file format. For example, OOXML-formatted files are decompressed and RTF-formatted files are parsed for extracting embedded binary contents in which our method is interested, whereas OLE-formatted files and most image files do not require any processing because they are binaries. For traffic data, extracting payloads of TCP/IP streams and parsing application-layer

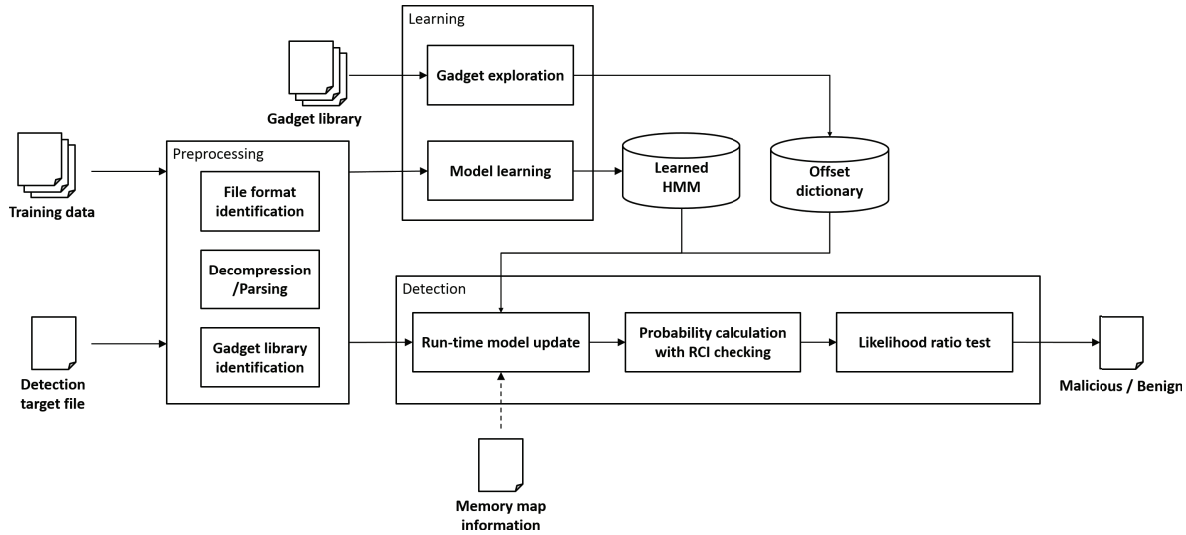


Fig. 1 Overview of our method

protocols dependent on the target applications. The third step is to identify the ROP gadget libraries. This step is described in detail in Sect. 3.4.3.

3.3 Learning Phase

3.3.1 Gadget Exploration

The gadget exploration step has two objectives:

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

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

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

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

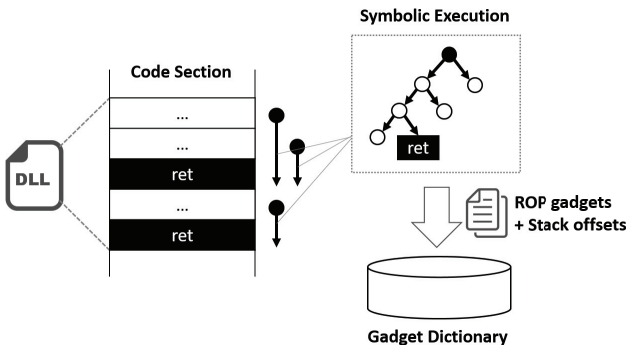


Fig. 2 Gadget exploration by symbolic execution

Figure 2 shows how our method exhaustively collects the gadgets from a library with symbolic execution. Our method repetitively conduct symbolic execution in which the entry points of these executions are all the addresses in the code section. That is, it first conducts symbolic execution by setting the top address of the code section to the instruction pointer register; then, it executes the code section by setting the top + 1 address to the instruction pointer register, and repeats the execution until it reaches the end of the code section. If the result of symbolic execution indicates that the gadget that starts from an entry point is valid, the method adds the entry point of the execution and the stack offset to the gadget dictionary.

We explain the setup of each execution. Our method first symbolize the values of the stack and registers except the instruction pointer. The instruction pointer register should point to the entry point of each execution target. We then begin the symbolic execution. When a symbolic value on the stack is moved to the instruction pointer, the execution stops as it has reached the end of the gadget.

3.3.2 Model Learning

For modeling byte sequences that include ROP chains, an HMM is designed to use the byte sequence of data as an observed sequence and the label sequence as a hidden state sequence. Therefore, the emission symbols of an HMM are the set of 0x00-0xFF. As argued in Sect. 2, ROP chains generally consist of three components: ROP gadget addresses (*addr*), constant values (*const*), and junk data (*junk*). In addition, the ROP chains are embedded in data such as documents and network payloads that generally have the same format as benign ones for being loaded on the memory properly. The bytes of this data have the label called *data*. Therefore, the state space consists of the set of addr[1-4], const[1-4], junk, and data labels, where the index number indicates the byte-wise position in the components, e.g., addr3 means the third byte of a ROP gadget address. Figure 3 depicts the state transition diagram of an HMM of 32-bit ROP chains designed for our detection method. In the diagram, D denotes data, A addr, C const, and J junk.

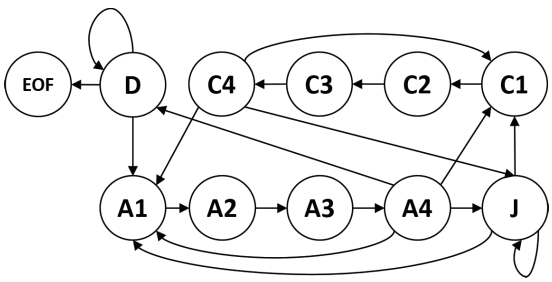


Fig. 3 State transition diagram for byte-wise HMM of 32-bit ROP chain embedded in data

Figure 4 and Fig. 5 shows an example model of ROP chains embedded in OLE2 files that uses the library of MSCOMCTL.OCX for their ROP gadgets. The former figure shows the transition probabilities of the model and the latter shows the emission probabilities. These figures can exhibit the differences in the probabilities among the labels. The transition probabilities are compliant to the state transition diagram in the Fig. 3. In addition, the emission probabilities are following the byte-level characteristics of ROP chains described in Sect. 2.2.

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

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

$$a_{i,j} = \frac{K_{i,j}}{\sum_{k \in Z} K_{i,k}},\ b_{i,o} = \frac{M_{i,o}}{\sum_{p \in V} M_{i,p}},\ \pi_i = \frac{N_i}{\sum_{j \in Z} N_j} \quad (1)$$

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

When calculating the emission probabilities of addr[1-

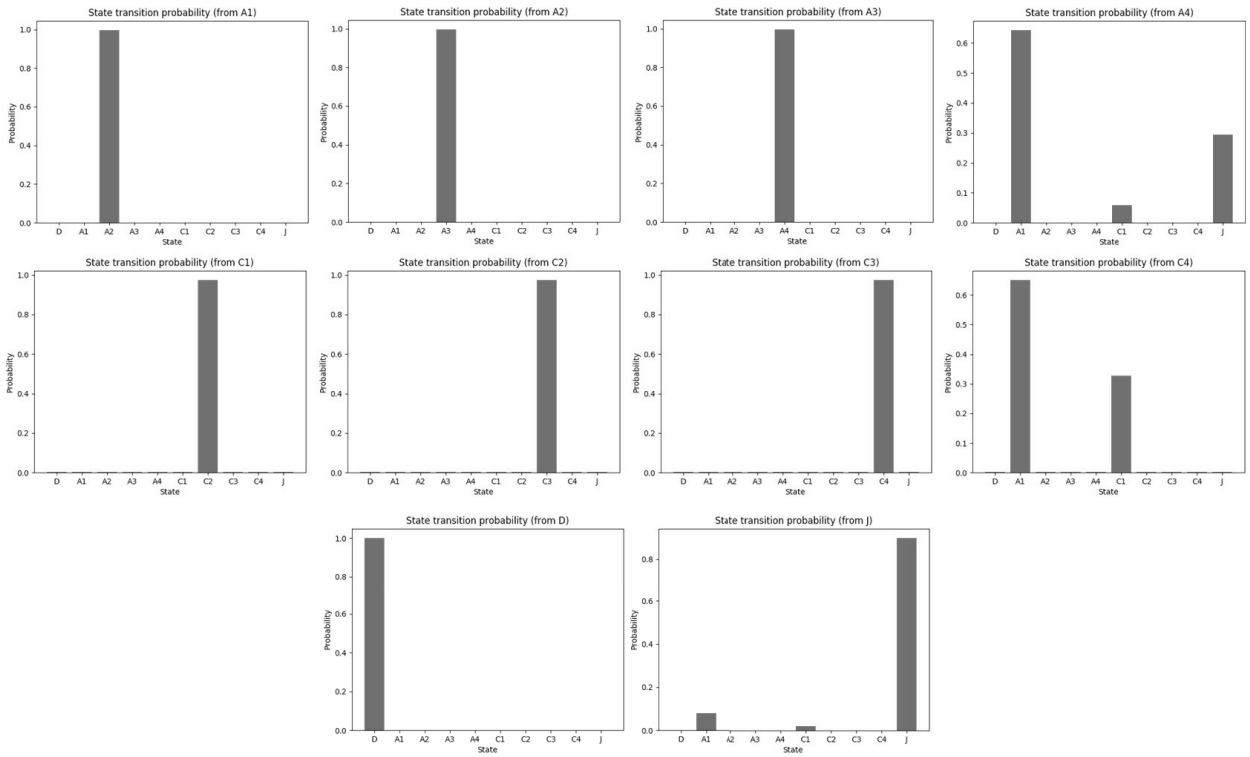


Fig. 4 Transition probabilities of an example model

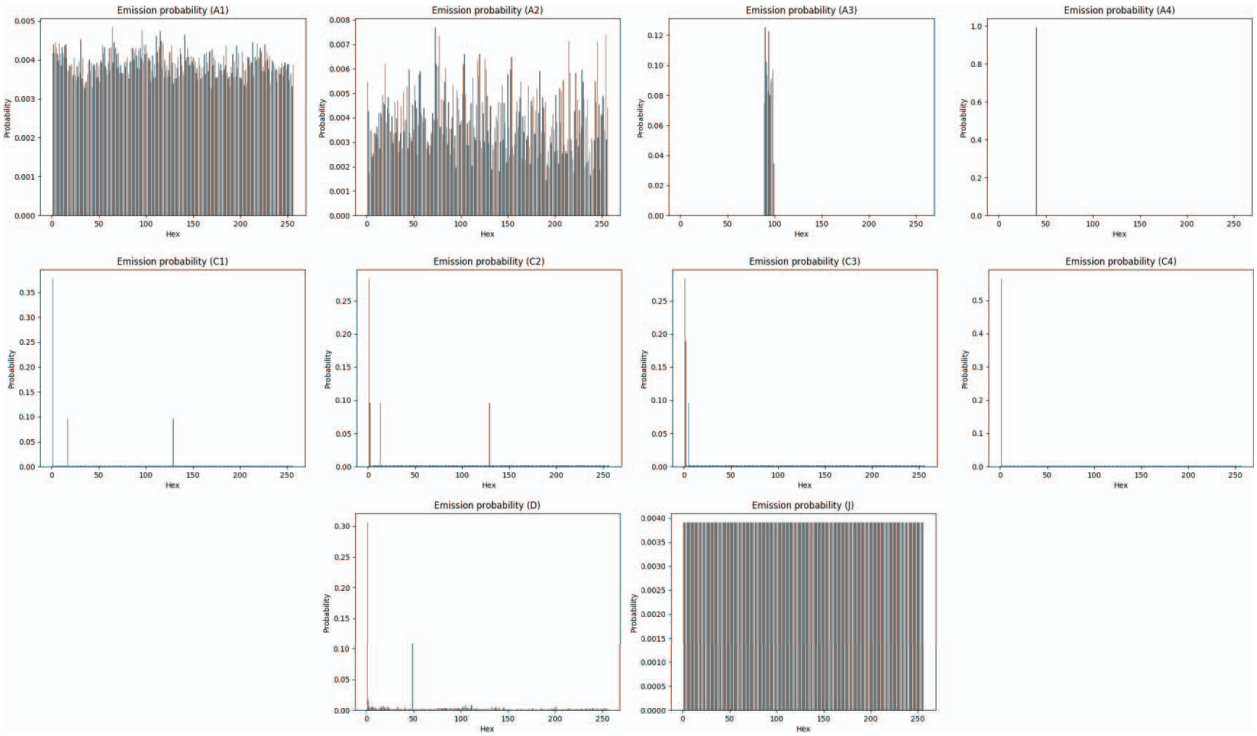


Fig. 5 Emission probabilities of an example model

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

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

3.4 Detection Phase

3.4.1 Run-Time Model Update

Several exploits in the wild employ JIT-ROP [10] for their ROP chains; therefore, our method targets ASLR-enabled libraries if memory map information of the target applications is available. Under the ASLR-enabled environment, a library is mapped at the various addresses. Therefore, probabilities of gadget addresses (addr[1-4]) of learned models

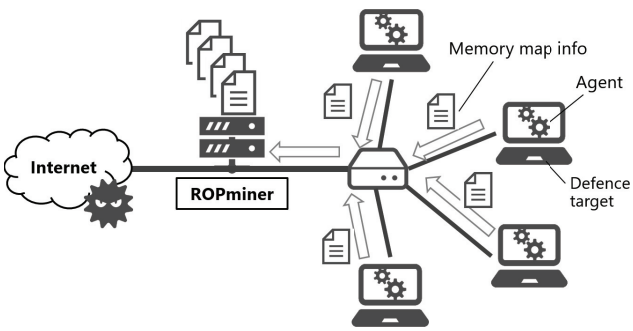


Fig. 6 Transfer mechanism of memory map information

dynamically change depending on the mapped address. This causes a problem that the model which only learned a static binary information of a library cannot work properly. Our method handles this problem by updating learned models on the basis of input memory map information.

Figure 6 describes the transfer mechanism of memory map information. Our method installs light-weight userland agents to the defending target and collects memory map information via the agents. The information is generally collected through system utilities provided by OSes. For example, /proc/{PID}/maps can provide them on Linux and VMMap [11] on Windows. The agents use them to collect the information and send it to the NIDS implemented with our method.

Since general ASLR implementation randomizes only higher bytes of the mapped address, our method updates a model of higher bytes of addr[1-4] by shifting its prob-

ability histogram required times. For example, when a library whose base address of code region is 0x00100000 is mapped at 0x32200000, the probability histogram of addr[1] is shifted by 0x32 times and addr[2] by 0x20 times. Since this operation is done in $O(1)$, we can update models with little overhead.

3.4.2 Probability Calculation

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

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

$$L(\theta|X) = P(X, \bigcap_{(i,j)\in J_X} F_{i,j}|\theta) \tag{2}$$

$$= P(X|\theta)P(\bigcap_{(i,j)\in J_x} F_{i,j}|X,\theta) \tag{3}$$

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

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

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

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} P(i \neq A1 \cup j = A1|X,\theta) \tag{4}$$

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

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} 1 - P(i = A1 \cap j \neq A1|X,\theta) \tag{5}$$

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

$$L(\theta|X) \approx P(X|\theta) \prod_{(i,j)\in J_x} 1 - P(i = A1|X,\theta)P(j \neq A1|X,\theta) \tag{6}$$

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

Here, $P(X|\theta)$, $P(i = \cdot|X,\theta)$, and $P(j \neq \cdot|X,\theta)$ are quickly calculated using forward and forward-backward algorithms [12], respectively. Note that $\cdot$ here is a placeholder of the symbols. Therefore, we can also compute the entire likelihood $L(\theta|X)$ in a short time.

3.4.3 ROP Gadget Library Identification

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

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

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

For the second question, the method adopts models and gadget dictionaries of non-ASLR DLLs and major ASLR-enabled DLLs for detection. Since our method generates a model for each gadget library, the method repeatedly inspects the target data while changing it. Therefore, it is preferable to identify the gadget library that is actually used by a target malicious data if possible, for reducing the inspection times. Several applications tend to load DLLs dependent on the contents that they read. Attackers often leverage this mechanism to force an application to load non-ASLR DLLs. For example, MS Office applications load non-ASLR DLLs such as MSVCR71.DLL, MSVCRT.DLL, and MSCOMCTL.OCX on the basis of ProgID/CLSID specified in the file as Li et al. [13] investigated.

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

3.4.4 Likelihood Ratio Test

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

$$Z = \frac{P(X|H_{Mal})}{P(X|H_{Ben})} = \frac{L(\theta_{Mal}|X)}{P(X|\theta_{Ben})} \tag{7}$$

where H_{Mal} is the hypothesis that the inspected data is malicious (i.e., containing ROP chains), H_{Ben} is the hypothesis

that the inspected data is benign, θ_{Mal} is the HMM of malicious data with ROP chains, and θ_{Ben} is the HMM of benign data. Then, if $Z > t$ the data is detected as malicious; otherwise, it is benign, where t is a threshold.

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

4. Implementation

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

ROPminer supports three document formats: CDF, OOXML, and RTF. We therefore implemented modules that parse files of each format and extract binaries embedded in them. If the input file is CDF, ROPminer just treats it as the inspection target. This is because CDF is a binary format and ROP chains are directly embedded in it. If the input is OOXML, ROPminer unzips it and extracts contained binary files as the inspection target. These binary files are generally used to contain ROP chains in OOXML-based exploits. If the input is RTF, ROPminer first finds \objdata control words that contain binaries. Because \objdata contains binaries in hex strings, ROPminer then decodes them and regards the decoded binaries as the inspection target.

5. Evaluation

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

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

5.1 Experimental Setup

Table 1 lists the datasets used in the experiments. For malicious samples, we collected the RTF-formatted and OOXML-formatted malicious documents that are most commonly used for file-based exploitation in the wild. Note that these files contain the other formatted files (e.g., an RTF-formatted file may contain OOXML-formatted files and CDF-formatted files in it) because they sometimes have nested structure. ROPminer also detects the malicious files contained in this nested structure. We confirmed that the malicious samples include at least several different ROP chains by manual analysis. According to the reports by VirusTotal (VT), the malicious files in the dataset exploit the following vulnerabilities: CVE-2010-{1297, 2883, 3333}, CVE-2012-{0158, 1856, 2539}, CVE-2013-{3346, 3906}, CVE-2014-{0496, 1761}, and CVE-2015-{1641, 1770, 2545}. Note that these vulnerabilities are sometimes used in combination for one malicious file. To conduct better experiments on ROP chain detection, the two data cleansing operations below are performed on the data.

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

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

For benign samples, we adopted govdocs [14], which are the datasets collected for forensic research. Since govdocs have several file formats, we can obtain MS Office document files. In addition, we also collected benign files using Bing search API [15]. We removed the files that have no data objects because they have no inspection target for ROPminer. After the removal, 40 files from govdocs and 1,351 files from bing remained.

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

Table 2 shows the environment used for conducting the experiments. All inspections were done on a single CPU.

Table 1 Datasets for evaluation

Category	Label	Samples	Source	Collection period
Training	Malicious	50	VirusTotal (VT) [16]	2016/12/26–2017/2/24
	Benign	278	govdocs [14], bing	-
Test	Malicious	1029	VT [16]	2016/12/26–2017/2/24
	Benign	1113	govdocs [14], bing	-

Table 2 Execution environments for evaluation

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

5.2 Detection Accuracy

For answering RQ1 and RQ2, we evaluated the false positives and false negatives in experiments. The result of the experiment suggest that ROPminer detected all malicious samples with no false negatives and that the average FPR was 0.03. The experiment is done by moving the

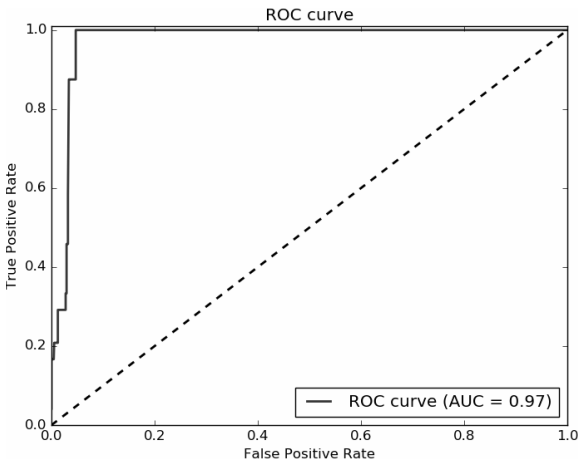


Fig. 7 ROC curve of inspection by ROPminer

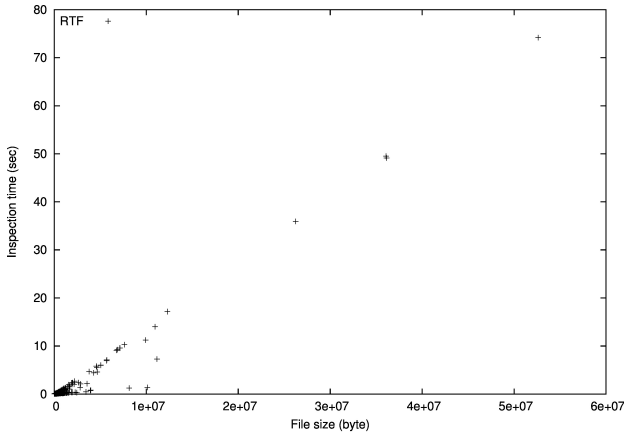


Fig. 8 Plot of relationship between file size and inspection time

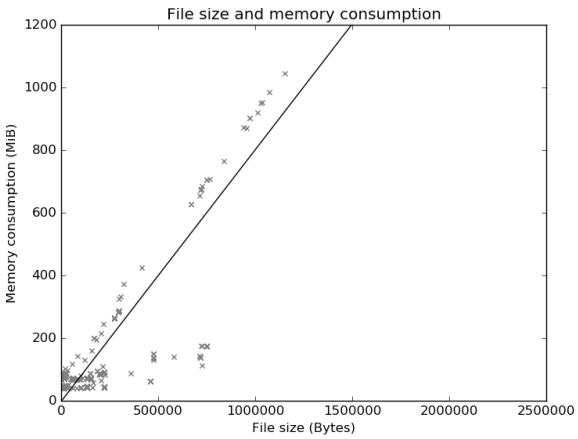


Fig. 9 Plot of relationship between file size and average memory consumption

parameter t for plotting the relationship between true positives and false positives as a receiver operating characteristic (ROC) curve. Figure 7 describes the ROC curve of the inspection by ROPminer. The area under curve (AUC) is 0.97, which in general indicates that it functions well as a recognition system.

In addition, we conducted an experiment with the cross-validation to reduce the sampling bias in the experiment. The experiment employs the 50 malicious samples with labels as well as the 50 benign samples that are undersampled from the benign category for avoiding the imbalanced data problem. Note that the ROP gadget libraries corresponding to the training data are used to build the models. We adopted 5-fold cross-validation and calculated the average FPR and FNR for each detection trial. The experimental result suggested that ROPminer detected all malicious samples with no false negatives and that the average FPR was 0.04. This result is almost identical to the previous experiment. Note that how ROPminer can detect future malicious documents by using the models generated with older samples is discussed in Sect. 6.6.

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

5.3 Performance

For answering RQ4, we also evaluated the performance of ROPminer while conducting the experiments. During the experiments, ROPminer inspected files at 0.96 s/file on average, and its throughput was around 0.83 Mbps/CPU. Figure 8 plots the relationship between file size and inspection time. The data were fitted by linear regression and the correlation coefficients were 0.96. Thus, the relation between the file size and the processing time is fairly correlated, and ROPminer can scale linearly in the processing speed. The theoretical analysis of the computational complexity is discussed in Sect. 6.3.

In addition, for answering RQ5, we evaluated the memory consumption of ROPminer while conducting the experiments. The continuous memory consumption of the experiment is measured using top command, and the memory consumption per file is measured using a performance profiler. The average memory consumption throughout the experiments is 347.2 MiB/file. Figure 9 plots the relationship between file size and average memory consumption. In the figure, we can see that the relationship between file size and memory consumption is linear. The data were fitted by linear regression, and the correlation coefficients were 0.96. Thus, it is experimentally proved that the relationship between file size and memory consumption is linear in the inspection by ROPminer. Moreover, since the regression coefficient was 803, ROPminer consumes 803 times as much memory as the file size it inspects. However, we found that this is a problem of our current implementation. Some part of ROPminer is implemented in Python and does not explicitly free the memory; therefore it uses four times as much re-

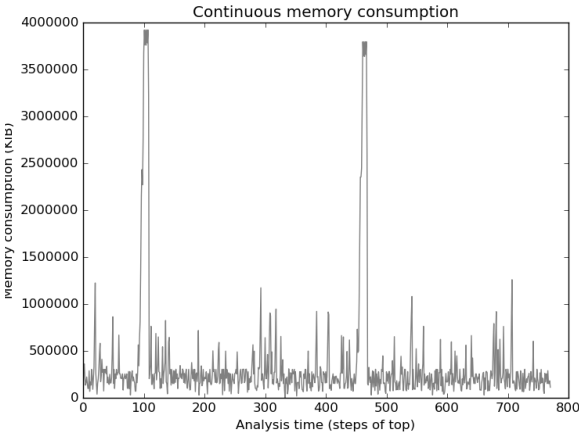


Fig. 10 Plot of continuous observation of memory consumption during experiments

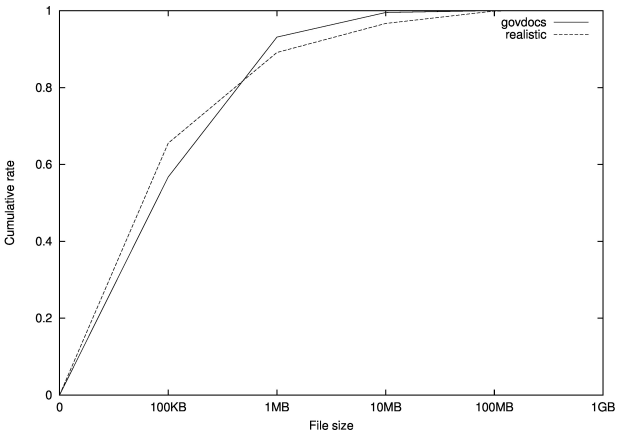


Fig. 11 Cumulative distribution function of size of files in datasets

dundant memory as an efficient implementation. The memory consumption is theoretically analyzed in Sect. 6.4. According to the analysis, the efficient implementation can inspect in memory about 200 times the size of the file ROPminer inspects.

Figure 10 plots the continuous observation of the memory consumption during the experiments. Although it has two spikes, the memory consumption during most of the experiments is limited to at most 1 GiB. Therefore, parallel execution is possible, considering the amount of equipped memory on recent machines. In addition, more concurrency will possible if the current implementation problem is fixed.

We investigated the size of files transferred on the network of an organization as realistic datasets. Figure 11 gives the cumulative distribution function (CDF) of the file size of the datasets. In the figure, "realistic" indicates the file sizes on the network of the organization. The function shows that there is little difference in the file size distribution between realistic and govdocs. In addition, about 90% of files in both data sets were not more than 1 MB. Since ROPminer can quickly inspect files that are below 1 MB in < 10 seconds, its detection is quick enough to deploy it in real networks.

Table 3 Duration of Gadget Exploration

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

Table 4 Duration of Model Learning

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

Lastly for answering RQ6, we measured the overhead of gadget exploration and model learning by ROPminer. Table 3 shows the duration of gadget exploration for two commonly used libraries. For each exploration, hours of execution duration is required. Since each of a symbolic execution begins from every byte of the code section of a library, the duration of gadget exploration mainly depends on the size of the code section. Table 4 depicts the duration of model learning. Each model takes only a few seconds to learn. Benign files are much faster to learn than malicious files since the benign files have just one type of label (i.e., data) and much simpler probabilistic calculation than malicious files. Overall, gadget exploration has a certain overhead whereas model learning requires only a few seconds. However, their overhead is not a major problem for ROPminer because both gadget exploration and model learning are performed separately from runtime detection.

6. Discussion

6.1 Comparison to Prior Work

We compare ROPminer with prior work which detects ROP attacks on the basis of static analysis. Table 5 shows the fundamental comparisons of the approaches between ROPminer and the methods proposed by existing research. Overall, most of the existing methods rely on predefined heuristic rules with the explicit or implicit assumptions of the form of ROP chains, whereas ROPminer adopts a rather systematic approach based on statistical machine learning without strong assumptions. The rest of this section (Sect. 6.1) discusses the differences between ROPminer and the other methods in detail.

6.1.1 Check My Profile

Check My Profile [5] quickly takes a memory snapshot (Minidump) of the target process by using a virtual machine and a shared memory driver. Then, it statically analyzes the snapshot and profiles the gadget candidate and ROP chain candidate based on predefined rules. Check My Profile is different from ROPminer in the assumption of the detection environment. Check My Profile requires the virtual machine and driver installation to take memory snapshots of the target process, whereas ROPminer needs no modification to end hosts. In addition, Check My Profile takes the time

Table 5 Comparison between ROPminer and prior work of ROP detection by static analysis

Method name	Approach	Fully static
ROPminer	Statistical learning	✓
Check My Profile [5]	Predefined rule	✗
eavesROP [9]	Pattern matching	✓
n-ROPdetector [7]	Pattern matching, Predefined rule	✓
STROP [8]	Predefined rule	✓

to open the target file for acquiring the memory snapshots, whereas ROPminer does not take much time. Because of these differences, Check My Profile can detect even client-side JIT-ROP, while ROPminer can be easily deployed and quickly detect static ROP.

6.1.2 eavesROP

eavesROP [9] detects a ROP chain based on matching the patterns of gadget addresses. It first collect the all possible gadget addresses from libraries, then, finds the gadget addresses in the target data by efficiently matching using fast Fourier transform (FFT). Unlike ROPminer, eavesROP only employs gadget addresses and does not utilize constant values and junk data for detection. The way to reduce false positives is also different from each other. eavesROP removes blocks that include UTF-8 strings which sometimes look like ROP chains, whereas ROPminer verifies RCI. Comparison between eavesROP and ROPminer in their detection accuracy may be difficult since only the simulation is done in the paper of eavesROP and experiments on real-world exploits are not conducted.

6.1.3 n-ROPdetector

n-ROPdetector [7] is a detection method which uses pattern matching based rules. The method consists of two parts, one is the pattern matching of known ROP gadget addresses; the other is the predefined rule-based verification. Since the method begins with finding the gadget addresses of API calls related to the memory permission, its focus is a stager ROP. Unlike ROPminer, n-ROPdetector uses just gadget addresses that appear in the exploits of Metasploit Framework, and does not consider the constant values and junk data. Also, n-ROPdetector differs in detection characteristics from ROPminer. That is, n-ROPdetector reports 16% of undetected ROP chains in their experiments, whereas ROPminer does not so far. Note that the FPR of n-ROPdetector is not evaluated in the paper, so we could not compare with that of ROPminer.

6.1.4 STROP

STROP [8] uses several predefined rules and parameters for detecting ROP chains. There are two major differences between STROP and ROPminer. First, STROP uses several assumptions in the form of ROP chains and requires seven heuristic parameters. In contrast, ROPminer does not require them. Second, STROP detects ROP payloads at low FPR (about 0.013) and comparatively high FNR (about 0.25) whereas ROPminer has higher FPR (about 0.03) and lower FNR (0.00) than STROP.

6.2 Limitation

The first limitation is that since ROPminer is based on the likelihood of ROP chains, ROP attacks that use quite a few gadgets, such as the return-into-libc attack [17], are difficult to detect. However, this is not a serious problem for ROPminer when considering the situation of recent attacks. The least amount of behavior that the attackers have to achieve through ROP attacks is as follows.

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

These steps are difficult to do by return-into-libc or a ROP chain with a few gadgets since they require a certain number of instructions and several API calls. Using return-into-libc or a short ROP chain to directly execute a shell instead of executing shellcode may be another option. This is done by invoking an API such as WinExec with the argument of a pointer to the command line string which attackers intend to execute. However, this is also difficult because the pointer to the command line string is ambiguous for attackers in the recent ASLR-enabled environments, unless the attacker exploits a memory disclosure vulnerability beforehand. To achieve this attack without exploiting a memory disclosure vulnerability, two ways are considered. One is using strings in the non-ASLR data regions of the target process memory as a command line string. In this case, whether the attacker can achieve the intention or not depends on the content of the data region, so it decreases the reliability of the exploit. The other is applying a pusha instruction as Stancill et al. [5] described; however, they also argued that a ROP chain that has at least five gadgets is required to accomplish it. Thus, we do not have to consider return-into-libc and short ROP chains, so the limitation is not a significant problem for ROPminer.

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

6.3 Computational Complexity

Here, we theoretically analyze the computational complexity of inspection by ROPminer. The main computation of

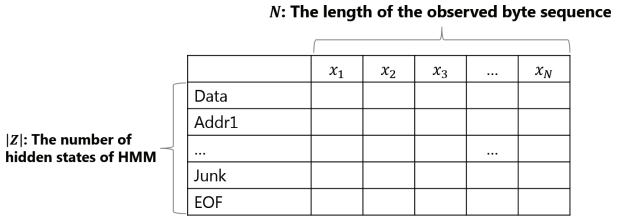


Fig. 12 Probability tables

the inspection involves calculating the likelihood ratio Z. Since ROPminer inspects data on the basis of the forward-backward algorithm of an HMM, it needs the table of forward probabilities and backward probabilities for calculation. Figure 12 describes the construction of probability tables. Both the forward probability table and backward probability table are in the same form as this. The probability table consists of $|Z|$ horizontal rows (the number of hidden states) and $N = |X|$ vertical columns (the length of the observed sequence in a data), where each cell contains the forward or backward probabilities. Because the HMMs of ROPminer have a constant number of hidden states, i.e., 11 states as shown in Sect. 3.3.2, the order of computation is $O(N)$, Thus, the inspection of ROPminer scales linearly with the size of inspected data.

6.4 Memory Consumption

We also theoretically estimated the memory consumption during ROPminer inspection. The main consumption of memory by ROPminer is the table of the forward probabilities and backward probabilities. As described in Fig. 12, the table that ROPminer uses consists of cells that each have a double variable. Assuming that a double variable is 8 bytes, the number of hidden states is 11 (as argued in Sect. 3.3.2), and the data inspected by ROPminer is N bytes, the memory consumption of a table is calculated as $88N$ bytes ($8*11*N$). This table is required for both forward probability and backward probability, so two tables are used for the likelihood calculation. Hence, $176N$ bytes ($88N*2$) are required to inspect data of N bytes for calculating the likelihood of one HMM. ROPminer uses two HMMs for inspection, a malicious one and a benign one. Therefore, if it is naively implemented, its memory consumption will be $352N$ bytes ($176N*2$) for an inspection. However, if the likelihood is calculated in a sequential manner (e.g., first it calculates the malicious likelihood with the table of the malicious HMM, freeing its memory, then calculates the benign one), the memory consumption for inspection will be suppressed to $176N$.

6.5 Number of Required Models

The number of models required for ROPminer is an important concern because it affects the inspection times. For learning, ROPminer has to generate one model for each library and each file format. Because some libraries have multiple versions, a library sometimes requires multiple models. Therefore, the number of models to be generated is calculated by the product of the number of libraries, the number of versions for each library, and the number of handled protocol/file formats. For detection, since ROPminer identifies the protocol/file format of the target data, it can determine which model to use from the viewpoint of data format. Also, ROPminer can identify a gadget library when it inspects some data formats as shown in Sect. 3.4.3. It can also determine from the viewpoint of the gadget library. Thus, only the number of versions affects the inspection times.

Table 6 Number of Versions and Mean Update Interval of DLL

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

We estimated the number of libraries and versions that ROPminer requires. Since the libraries that attackers can adopt depend on the target application, we assume that the exploit target is 32-bit MS Office 2007, 2010, and 2013, which are major applications as targets. First, we searched the libraries commonly used for ROP chains on the target applications. Second, we investigated the number of different versions for each library. Then we collected all of them from the National Software Reference Library (NSRL) [18] and third-party file-sharing services, for obtaining the mean update intervals. The NSRL is a project which collects software from various sources and gathers file profiles computed from the software.

Table 6 shows the number of versions and the mean update intervals of the well-used libraries. We searched and collected all versions of the three libraries: MSVCR71.DLL, MSVCRT.DLL, and MSCOMCTL.OCX. MSVCR71.DLL has only one version because the target applications use the same one located in `C:\Program Files\Microsoft Office\Office1X\ADDINS`. MSVCRT.DLL and MSCOMCTL.OCX have several versions for each in the target applications. However, due to the update interval (e.g., several libraries are updated immediately after the previous update), we found that the versions one has to focus on are two or three for each of them. In addition, if one knows the period in which the updates are already done on the systems to defend, the libraries may be more limited.

6.6 Concept Drift and Update Interval of Model

We also discuss the concept drift of the models of ROPminer and the required intervals of updates. The concept drift of the ROP chain detection occurs depending on the update of the library used for the ROP gadgets; therefore, the model update is required regarding it. As shown in Table 6, library updates are rarely done (less than once a year) on the three libraries that are used for ROP chains. Also, a model update may be required when the structure of the ROP chains used in the wild is changed drastically, e.g., the appearance of ROP chains with a large amount of junk code. However,

we did not observe such a ROP chain while we conduct the experiments. Hence, the model-update frequency is not so high with ROPminer.

6.7 Applicability to Other Code Re-use Attacks

We discuss the applicability of ROPminer to other code re-use attacks, jump-oriented programming (JOP) [19] and call-oriented programming (COP) [20]. Although ROPminer mainly focuses on ROP in this paper, it is also effective against JOP and COP depending on its implementation. JOP has a characteristic region named a "dispatch table", which contains a number of JOP gadget addresses used for the JOP attack. Since the table includes the consecutive characteristic byte sequence of ROP gadget addresses, i.e., addr[1-4], ROPminer can detect the existence of the table. COP chains have a similar architecture except for the absence of the dispatcher gadget. Due to the existence of a region in which COP gadgets are stored ("COP gadget table"), ROPminer can detect the COP chains embedded in the data. Note that a ROPminer operator has to collect the JOP/COP gadgets that end with indirect jump/call instructions for creating a JOP/COP gadget dictionary, instead of gathering ROP gadgets. We also note that our method should learn models of ROP, JOP, and COP separately for better detection because transition probabilities of them may slightly differ from each other.

6.8 Robustness against Evasion

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

6.8.1 Evasion Method

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

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

We considered two evasion methods that change the sequence pattern. The first is to dynamically change the alignment of ROP chains. ROP chains in 32-bit environments generally have 4-byte alignment; therefore corrupting the alignment is sometimes evasive against several static ROP detection methods that assume that the chains are aligned. Figure 13 describes an example of alignment corruption. In the figure, attack gadgets indicate that the gadgets are for executing arbitrary code of attackers,

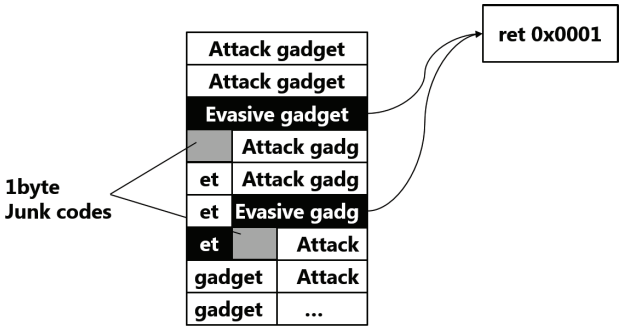


Fig. 13 Alignment corruption attacks

whereas evasive gadgets are used for corrupting the alignment. As you can see in the figure, the gadget of the RET 0x0001 instruction can cause a one-byte increase in the stack pointer, which shifts and corrupts the alignment. This evasion method can bypass several existing detection methods such as n-ROPdetector [7] and STROP [8]. The second is to change the sparseness of the ROP gadget addresses in a ROP chain. Since ROP chains used in the wild generally contain the minimum amount of required junk code, ROP gadget addresses are located densely. If attackers leverage the gadgets of RET 0xXXXX (big two-byte value) and pack much junk code into the generated space, the ROP chains contain much sparser ROP gadget addresses. This method can also evade several detection methods.

6.8.2 Robustness of ROPminer

ROPminer is robust against the evasion methods that change the byte pattern of ROP addresses because it takes account of all the byte patterns available for ROP gadget addresses as described in Sect. 3.3.2. The evasion that changes the junk code to arbitrary code is considered to be almost ineffective against ROPminer. Since our method assumes that the junk code contains random bytes and has few characteristics by nature, changing it does not significantly harm the detection accuracy.

The evasion by changing the constant values is possibly effective; however, there is a limitation in that a certain number of constants cannot be changed. This is because plenty of static symbol constants have been defined by Windows APIs. For example, if one wants to enable write and execute access to a memory region with the VirtualProtect API, 0x00000040 (PAGE_EXECUTE_READWRITE) is necessary for the argument. Moreover, several constants that are important for ROP attacks cannot take fully arbitrary values dependent on OSes. For instance, address values taken by memory allocation and memory protection APIs on Linux have to be a multiple of the page size (the size is typically 0x1000); therefore, the values will be 0xXXXXX000. Due to these limitations, evading detection by ROPminer on the basis of byte pattern changing is difficult.

The method of corrupting the alignment is ineffective against ROPminer since ROPminer uses byte-wise

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

6.9 Labeling of Training Data

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

6.9.1 Manual Labeling

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

6.9.2 Dynamic Detection-Assisted Labeling

Dynamic detection-assisted labeling uses existing dynamic-based ROP attack detection systems such as ROPdefender [21] and EMET [22], which raise an exception when the ROP attack is detected, as well as a debugger. We first prepare the environment which is vulnerable to the malicious files for training. This may require several environments because which OSs and applications are vulnerable to the exploitation are sometimes ambiguous. Then attach the application to the debugger and open the malicious file. Since the debugger catches an exception caused by the detection system, the debugger stops around the beginning of the ROP chain. Thus, an operator can use the information to analyze the ROP chain.

6.9.3 Taint-Based Automatic Labeling

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

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

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

6.10 Practical Applicability

As shown in Sect. 5, ROPminer has a 0.03 FPR which seems relatively high for realistic deployment, despite its high TPR. Therefore, we assume that ROPminer is used as a filter on networks to make use of these characteristics. The filter we suppose is located before dynamic analysis sandboxes and narrows down the files which are input to the sandboxes. Only the files detected as malicious by the filter are input to the sandboxes. Since analysis by generic sandboxes consumes much more time than that by ROPminer, the prefiltering of ROPminer can accelerate the whole analysis process. Moreover, sandboxes can re-inspect false positives of ROPminer; therefore this application style can compensate for the shortcoming (false positives) of ROPminer.

7. Other Related Work

7.1 ROP Detection by Dynamic Analysis

Since Shacham et al. proposed ROP [1], [6], a number of studies that dynamically detect ROP have been made. ROPdefender [21] and ROP Monitor [23] are proposed methods that adopt control-flow integrity (CFI). DROP [24] employs rule-based detection based on the size of executed gadgets. However, Göktaş et al. [25] proved that

attackers can succeed with ROP attacks even under size-based detection. Several randomization based measures are also proposed. Shuffler [26] proposes a method of continuously randomizing the memory space in a short interval to make JIT-ROP ineffective. Code shredding [27] extends ASLR to the byte granularity randomization of program code location. Return address protection (RAP) [28] provides defense by encrypting the return addresses on a stack. ROPMEMU [29] offers a method for analyzing sophisticated ROP chains on memory by leveraging multi-path execution. EigenROP [30] is similar to ROPminer in that it uses statistical learning for detection. The difference is that it is based on microarchitecture-independent run-time features.

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

7.2 Other Attack Code Detection

Gu et al. [31] provided a system which first takes a virtual memory snapshot of the target process, followed by detecting the shellcode in it based on emulation and malicious system call identification. Polychronakis et al. [32] proposed a method that detects shellcode based on emulation and runtime heuristics such as kernel32.dll resolution, process memory scanning, and SEH-based GetPC. SHELLOS [33] is a system that leverages hardware virtualization to efficiently and accurately detect shellcode by directly executing instruction sequences on the CPU. Iwamoto et al. [34] proposed a method for detecting shellcode in malicious documents based on entropy calculation and emulation. OfficeMalScanner [35] is a major tool that can detect shellcode by checking for the existence of the well-known heuristics that shellcode employs.

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

7.3 Byte-Level Malicious File Detection

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

Tabish et al. [36] proposed a method that can detect malware without signatures. The method divides the byte-level contents of the target file into 1KB blocks. Then, it extracts a set of statistical features computed on the N-gram of each block and classifies them as malicious or benign using a decision tree with boosting. If the portion of blocks classified as malicious exceeds a threshold, the file is also classified as malicious. Since the main purpose of this method is not detecting ROP-based malicious files but detecting malware, it is not designed to be aware of ROP chains. Because general ROP chains are less than 1KB, just one or two blocks of the target file turns to malicious by the existence of a ROP chain with this method. Therefore, detecting ROP-based malicious files by the method is sometimes difficult because the portion of malicious blocks does not increase much. Note that some ROP-based malicious files using heap spraying has much larger ROP chains; thus, they can be detected by the method.

Smutz et al. [37] proposed a method of randomizing contents and encodings in a file, which is inspired by ASLR. This can prevent a certain rate of exploits from execution. This approach is quite different from ROPminer in that it does not focus on detecting attack code; therefore it should be used together.

8. Conclusion

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

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," Proc. 14th ACM Conference on Computer and Communications Security (CCS '07), pp.552–561, ACM, 2007.
[2] Sophos, "Office exploit generators." https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/sophos-office-exploit-generators-szappanos.pdf. (accessed: 2020-01-07).
[3] McAfee, "Threadkit exploit kit." https://www.mcafee.com/enterprise/ja-jp/threat-center/threat-landscape-dashboard/exploit-kits-details.threadkit-exploit-kit.html. (accessed: 2020-01-07).
[4] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E.P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," Proc. Fourth European Workshop on System Security (EUROSEC '11), pp.1–6, ACM, 2011.
[5] B. Stancill, K.Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi, "Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets," Lecture Notes in Computer Science, vol.8145 (Proc. 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '13)), pp.62–81, Springer, 2013.
[6] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," ACM Transactions on Information and System Security (TISSEC), vol.15, no.1, pp.1–34, 2012.
[7] Y. Tanaka and A. Goto, "n-ropdetector: Proposal of a method to detect the rop attack code on the network," Proc. 2014 Workshop on Cyber Security Analytics, Intelligence and Automation (SafeConfig '14), pp.33–36, ACM, 2014.
[8] C. YoungHan and L. DongHoon, "Strop: Static approach for detection of return-oriented programming attack in network," IEICE Trans. Commun., vol.E98-B, no.1, pp.242–251, 2015.
[9] C. Jämthagen, L. Karlsson, P. Stankovski, and M. Hell, "eavesrop:

Listening for rop payloads in data streams," Lecture Notes in Computer Science, vol.8783 (Proc. 17th International Conference on Information Security (ISC '14)), pp.413–424, Springer, 2014.
[10] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," Proc. 2013 IEEE Symposium on Security and Privacy (SP '13), pp.574–588, IEEE, 2013.
[11] Microsoft, "Vmmap." https://docs.microsoft.com/en-us/sysinternals/downloads/vmmap. (accessed: 2019-11-19).
[12] L.R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," Proc. IEEE, vol.77, no.2, pp.257–286, IEEE, 1989.
[13] H. Li and B. Sun, "Attacking interoperability: An ole edition," Blackhat USA briefings 2015, https://www.blackhat.com/docs/us-15/materials/us-15-Li-Attacking-Interoperability-An-OLE-Edition.pdf. (accessed: 2017-03-21).
[14] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," digital investigation, vol.6, pp.S2–S11, Elsevier, 2009.
[15] Microsoft Azure, "Bing search apis." https://azure.microsoft.com/en-us/services/cognitive-services/search/. (accessed: 2017-03-28).
[16] VirusTotal, "Virustotal." https://www.virustotal.com/. (accessed: 2017-03-09).
[17] Solar-Designer, "'"Return-to-libc" attack." Bugtraq. Aug. 1997.
[18] National Institute of Standards and Technology, "National software reference library." https://www.nist.gov/software-quality-group/national-software-reference-library-nsrl. (accessed: 2017-08-09).
[19] T. Bletsch, X. Jiang, V.W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," Proc. 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11), pp.30–40, ACM, 2011.
[20] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," Proc. 23rd USENIX Security Symposium (USENIX Security '14), pp.385–399, USENIX Association, 2014.
[21] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," Proc. 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11), pp.40–51, ACM, 2011.
[22] Microsoft, "Emet." https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit. (accessed: 2019-08-29).
[23] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," Lecture Notes in Computer Science, vol.6503 (Proc. 6th International Conference on Information Systems Security (ICISS '10)), pp.140–155, Springer, 2010.
[24] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," Lecture Notes in Computer Science, vol.5905 (Proc. 5th International Conference on Information Systems Security (ICISS '09)), pp.163–177, Springer, 2009.
[25] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," Proc. 23rd USENIX Security Symposium (USENIX Security '14), pp.417–432, USENIX Association, 2014.
[26] D. Williams-King, G. Gobieski, K. Williams-King, J.P. Blake, X. Yuan, P. Colp, M. Zheng, V.P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," Proc. 12th USENIX conference on Operating Systems Design and Implementation (OSDI '16), pp.367–382, USENIX Association, 2016.
[27] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks," Proc. 28th Annual Computer Security Applications Conference (ACSAC '12), pp.309–318, ACM, 2012.
[28] PaX Team, "Rap: Rip rop," Hacker to Hacker Conference (H2HC) 12th Edition, https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf. (accessed: 2017-03-21).
[29] M. Graziano, D. Balzarotti, and A. Zidouemba, "Ropmemu: A framework for the analysis of complex code-reuse attacks," Proc. 11th ACM Asia Conference on Computer and Communications Security (ASIACCS '16), pp.47–58, ACM, 2016.
[30] M. Elsabagh, D. Barbará, D. Fleck, and A. Stavrou, "Detecting rop with statistical learning of program characteristics," Proc. Seventh ACM Conference on Data and Application Security and Privacy (CODA '17), pp.219–226, ACM, 2017.
[31] B. Gu, X. Bai, Z. Yang, A.C. Champion, and D. Xuan, "Malicious shellcode detection with virtual memory snapshots," Proc. 29th IEEE Conference on Computer Communications (INFOCOM '10), pp.974–982, IEEE, 2010.
[32] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Comprehensive shellcode detection using runtime heuristics," Proc. 26th Annual Computer Security Applications Conference (ACSAC '10), pp.287–296, ACM, 2010.
[33] K.Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "Shellos: Enabling fast detection and forensic analysis of code injection attacks," Proc. 21st USENIX Security Symposium (USENIX Security '11), pp.183–200, USENIX Association, 2011.
[34] K. Iwamoto and K. Wasaki, "A method for shellcode extraction-from malicious document files using entropy and emulation," International Journal of Engineering and Technology, vol.8, no.2, pp.101–106, 2016.
[35] F. Boldewin, "Analyzing msoffice malware with officemalscanner." http://www.reconstructer.org/code/OfficeMalScanner.zip. (accessed: 2016-01-15).
[36] S.M. Tabish, M.Z. Shafiq, and M. Farooq, "Malware detection using statistical analysis of byte-level file content," Proc. ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics (CSI-KPD '09), pp.23–31, ACM, 2009.
[37] C. Smutz and A. Stavrou, "Preventing exploits in microsoft office documents through content randomization," Lecture Notes in Computer Science, vol.9404 (18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '15)), pp.225–246, Springer, 2015.

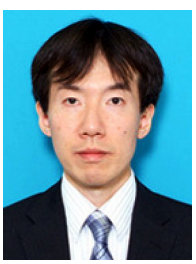
Toshinori Usui received his B.A. degree in environment and information studies from Keio University and M.S. degree in information science and technology from The University of Tokyo, Japan in 2012 and 2015, respectively. He is currently a Ph.D. student in the Department of Information and Communication Engineering, The University of Tokyo. Since joining the Nippon Telegraph and Telephone Corporation (NTT) in 2015, he has been engaged in research and development on malware analysis and detection. He is currently with NTT Secure Platform Laboratories, where he is engaged in the Cyber Security Project.

Tomonori Ikuse received his M.E. from Nara Institute of Science and Technology in 2012. He joined NTT in 2012. He has been engaged in research and development of malware analysis technologies. From 2016 to 2019, he was engaged in security log analysis at the Security Operation Center of NTT Security (Japan) KK.

Yuto Otsuki received his B.E. degree from College of Information Science and Engineering, Ritsumeikan University in 2011, and his M.E. degree from Graduate School of Science and Engineering, Ritsumeikan University in 2013. He also received his D.Eng. degree from Graduate School of Information Science and Engineering, Ritsumeikan University in 2016. From 2016 to 2019, he was engaged in research of malware analysis and digital forensics at NTT Secure Platform Laboratories. He is now with NTT Security (Japan) KK.

Yuhei Kawakoya received his B.E. and M.S. in science and engineering from Waseda University in 2003 and 2005, respectively. He has been engaged in R&D since 2005 on computer security. From 2013 to 2016, he was engaged in R&D of NTT Innovation Institute, Inc. as a software engineer. He is a member of IPSJ and IEICE.

Makoto Iwamura received his B.E., M.E., and D.Eng. in science and engineering from Waseda University, Tokyo, in 2000, 2002, and 2012, respectively. He joined NTT in 2002. He is currently with NTT Secure Platform Laboratories, where he is engaged in the Cyber Security Project. His research interests include reverse engineering, vulnerability discovery, and malware analysis.



Jun Miyoshi received his B.E. and M.E. degrees in system science from Kyoto University in 1993 and 1995, respectively. Since joining NTT in 1995, he has been researching and developing network security technologies. From 2011 to 2016, he was engaged in R&D strategy management of NTT Secure Platform Laboratories. Now he is a research group leader of the Cyber Security Project in the Laboratories. He is a member of IEICE.

Kanta Matsuura received his Ph.D. degree in electronics from the University of Tokyo in 1997. He is currently a Professor of Institute of Industrial Science at the University of Tokyo. From March 2000 to March 2001, he was a visiting scholar at University of Cambridge. His research interests include cryptography, computer/network security, and security management such as security economics. He was an Associated Editor of IPSJ Journal (2001–2005) and IEICE Transactions on Communications (2005–2008). He was Editor-in-Chief of Security Management (2008–2012), and is an Editorial-Board member of Design, Codes, and Cryptography (2010-present). He is a fellow of IPSJ, and a senior member of IEEE, ACM, and IEICE. He is a Vice President of JSSM (Japan Society of Security Management) (2016-present).