Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

ROPSentry: Runtime defense against ROP attacks using hardware performance counters

Sanjeev Das [a,*], Bihuan Chen [b,*], Mahintham Chandramohan [c], Yang Liu [c], Wei Zhang [d]

[a] University of North Carolina at Chapel Hill, USA
[b] Fudan University, China
[c] Nanyang Technological University, Singapore
[d] Hong Kong University of Science and Technology, Hong Kong

ARTICLE INFO

Article history:
Received 16 July 2017
Received in revised form 16 October 2017
Accepted 12 November 2017
Available online 24 November 2017

Keywords:
ROP attacks
Hardware performance counter
Code-reuse attacks
Memory corruption attacks
Runtime memory attacks
Exploit defense

ABSTRACT

Return-Oriented Programming (ROP) is one of the most common techniques to exploit software vulnerabilities. However, existing defense techniques can be defeated by attackers, or suffer from high performance overhead. In this paper, we propose a defense framework, named ROPSentry, to detect ROP attacks at runtime. It is built on the observation that ROP exploits usually trigger different hardware events than normal programs generated by compilers. Hence, we leverage hardware performance counters to track such hardware events and analyze behavioral patterns of ROP attacks. ROPSentry has two approaches. The *ROP-only defense approach* detects ROP attacks via capturing the patterns of ROP exploits, where we propose to sample the hardware performance counters at mispredicted return events instead of at every microinstruction for a low performance overhead. To further reduce performance overhead, we propose a *self-adaptive defense approach* to dynamically switch between low and high sampling rates. It detects the patterns of spraying attacks (i.e., one common ROP payload delivery technique) at a low sampling rate, and then switches to a high sampling rate for detecting the patterns of ROP exploits. Our evaluation on 11 real-world ROP exploits, 50 synthetically generated ROP exploits and 1000 benign websites has shown that, the ROP-only and self-adaptive approaches are effective in detecting ROP attacks with low performance overhead (11% and 1% respectively) as well as low false positive; and they significantly outperform the state-of-the-art techniques in terms of performance overhead without losing the detection accuracy.

1. Introduction

Recent software exploitation trend shows that 80% of all vulnerabilities are exploited via code-reuse or Return-Oriented Programming (ROP) attacks (Rains et al., 2015), with zero-day exploits against commercial off-the-shelf software and open-source software. A ROP attack exploits the presence of *gadgets* (i.e., small instruction sequences ending in a return instruction) in the target program, where multiple gadgets are chained together to build complex yet meaningful attacks. To deliver the ROP payload, spraying attack remains one of the most commonly-used mechanisms (Rains et al., 2015). It can place attacker-controlled code or data at a desired memory

[*] *Corresponding author.*
E-mail addresses: sdas@cs.unc.edu (S. Das); bhchen@fudan.edu.cn (B. Chen).

https://doi.org/10.1016/j.cose.2017.11.011
0167-4048/© 2017 Elsevier Ltd. All rights reserved.

location. Since the inception, many high-profile attacks including the exploits in Internet Explorer (Microsoft corporation) and Adobe Reader (Multi-state information sharing and analysis center, 2009) leverage spraying attack techniques to accomplish their attacks.

ROP can easily evade hardware protection techniques (e.g., data execution prevention (van de Ven, 2004)) and OS mitigation techniques (e.g., address space layout randomization (Team, 2003)) (Snow et al., 2013). While so many defense techniques have been proposed, there has also been a continual evolution in sophistication of ROP exploits to defeat the state-of-the-art defense mechanisms (Carlini and Wagner, 2014; Davi et al., 2014; Göktaş et al., 2014). The existing defense techniques suffer from several key problems. As a result, most techniques can be easily defeated by attackers via carefully crafting code-reuse attack payloads, or require high performance overhead. For example, most ROP mitigations rely on API hooking (Fratric; Pappas et al., 2013) or instrumentation (Davi et al., 2009, 2011; Jacobson et al., 2014). However, hooking-based techniques can be bypassed by hook hopping (Transparent rop detection using cpu performance counters, 2014), and instrumentation-based techniques have high performance overhead. Besides, some defensive mechanisms use predefined policies to detect ROP attacks (Pappas et al., 2013). However, these policies are inflexible and weak, and hence can be bypassed by constructing ROP gadgets that adhere to such policies; and the runtime policy checking usually incurs high performance overhead, making them impractical for real-world deployment.

On the other hand, the state-of-the-art spraying attack detection mechanisms (Cristalli et al., 2016; Ratanaworabhan et al., 2009) can only handle attacks that use system memory allocators, while failing to detect attacks that use custom memory allocators (Chen et al., 2016). Unfortunately, custom memory allocators are becoming popular among software applications in the recent years, which alleviates the applicability of these techniques.

Differently, several ROP mitigations (Cheng et al., 2014; Pappas et al., 2013; Tang et al., 2014; Wicherski, 2013; Yuan et al., 2011) attempt to use low-level hardware events in the CPU (e.g., cache misses or branch mispredictions) via hardware performance counters (HPCs) to detect code-reuse attacks. In brief, modern processors feature hardware-based counters, called as HPCs, which can selectively monitor and measure large number of events at the hardware level. Compared to software behaviors, these hardware events are more difficult for attackers to control directly in evasion attacks. For example, it is easier for attackers to change system calls than to manipulate branch mispredictions in a precise way while performing exploits. However, these techniques suffer from two problems: (1) high performance overhead resulting from the high-frequent interrupts to obtain HPC values and heavy-weight machine learning techniques to process these HPC values; and (2) limited availability of programmable performance counters in the CPU (Tang et al., 2014) that only allows limited events to be monitored at runtime.

To address these problems, in this paper, we propose a defense framework, named *ROPSentry*, to detect ROP attacks at runtime using HPCs with low performance overhead. The key observation underlying ROPSentry is that a ROP attack triggers significantly different hardware events pattern than a normal program generated by a compiler (Pfaff et al., 2015). For example, ROP exploits chain several gadgets ending in return instructions and cause an unexpected program control flow, leading to high branch misprediction. Similarly, a spraying attack usually performs similar memory allocation operations, causing similar performance counter values for load microinstructions. Hence, a ROP attack can be distinguished from a normal program at runtime by capturing the behavioral patterns of spraying attacks and/or ROP exploits from monitored hardware events.

However, ROP gadgets are small in nature, i.e., around 3–5 instructions per gadget. Hence, to capture any malicious pattern, we need to fetch HPC values in a high frequency, which increases the performance overhead during runtime. To tackle this problem, we develop a *ROP-only defense approach* into ROPSentry, which detects ROP attacks by capturing the patterns of ROP exploit precisely through sampling HPCs at mispredicted return events instead of at every microinstruction, based on the observation that ROP exploits cause several return mispredictions. To further reduce performance overhead, we propose a *self-adaptive defense approach*, which leverages the ROP payload delivery mechanism spraying attack (which often involves millions of instructions), to dynamically switch between two sampling rates at runtime such that the performance overhead is negligible. In particular, we initially employ a low sampling rate to detect the spraying attack. Once a potential spraying attack is observed, we switch to a high sampling rate to determine whether a ROP attack is occurred. If yes, we terminate the program; otherwise, we switch back to the low sampling rate to continue the detection of spraying attack. Hence, ROPSentry can detect non-spraying-based ROP attacks as well as spraying-based (using system or custom allocators) ROP attacks. This work provides a complete solution to handle the entire spectrum of ROP attacks, as shown in Fig. 1 (see Section 2.1 for a detailed discussion).

We evaluated the detection accuracy of our framework on 11 real-world ROP exploits and 50 synthetically generated ROP exploits. The results indicated that our framework can detect all the 61 exploits. We also evaluated the performance overhead and the false positive rate of our framework using 1000 benign websites, which demonstrated that our framework can keep the performance overhead at 11% and 1% when using the ROP-only and self-adaptive defense approach respectively, and have zero false positive. Moreover, we compared our framework with several state-of-the-art techniques, including the approaches by Tang et al. (2014) and Cristalli et al. (2016). The results demonstrated that our framework had much lower performance overhead than theirs without losing the detection accuracy.

The main contributions of this work are as follows.

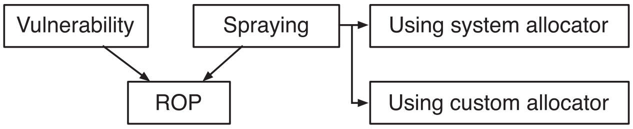


Fig. 1 – Variants of ROP attacks.

- We leverage hardware events to capture behavioral patterns of ROP exploits, and propose a ROP-only defense approach to detect both non-spraying-based and spraying-based ROP attacks at runtime.
- We propose a self-adaptive defense approach to further reduce the performance overhead of detecting spraying-based ROP attacks, i.e., dynamically switching between low and high sampling rate to detect spraying attacks and ROP exploits.
- We have empirically demonstrated that, ROPSentry is effective in detecting ROP attacks at runtime with low performance overhead and low false positive; and it significantly outperforms the state-of-the-art techniques in terms of performance overhead without sacrificing the detection accuracy.

2. Preliminaries and challenges

In this section, we first introduce the preliminaries on ROP attacks and hardware performance counters (HPCs), and then summarize the challenges in existing ROP mitigations.

2.1. *ROP attacks and the variants*

Fig. 2 shows the typical program memory layout of a ROP attack. ROP uses small instruction sequences ending in a return instruction, called gadgets. The ROP payload consists of return addresses that target the gadgets already present in the library or victim code. The stack pivot gadget switches the original stack to attacker-controlled data in the memory via triggering a vulnerability (e.g., Use After Free) in the application to pass the control to the attacker-controlled ROP chain. The execution of the return instruction in a gadget moves the stack pointer to the next return address, which advances the control from one gadget to another. Each gadget contributes certain computations to perform the complete attack.

Fig. 1 presents the variants of ROP attacks, which can be (1) non-spraying-based ROP attacks and (2) spraying-based ROP attacks. Non-spraying-based ROP attacks are possible via exploiting vulnerabilities such as heap overflow. Attackers first put the ROP payload in the heap, and then exploit the heap vulnerability to control the stack pointer and program counter. Attackers can also leverage spraying attacks to spray the ROP payload in the heap, and use some offensive techniques such as stack pivoting to execute the ROP payload. They can use either system allocators (e.g., malloc) or custom allocators (Liu, 2013) (e.g., Jemalloc) to allocate memory for achieving the spraying.

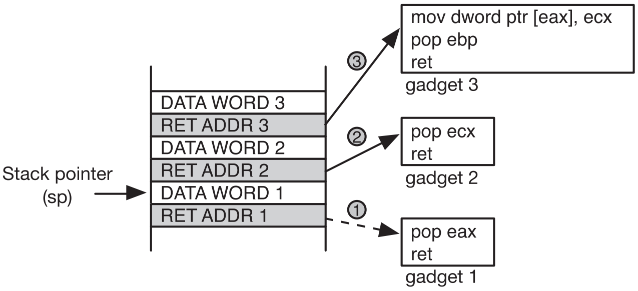


Fig. 2 – The memory layout of a ROP exploit.

2.2. *Hardware performance counters*

Performance monitoring unit (PMU) was initially introduced in Pentium processor with a set of model-specific performance monitoring counters (Intel Manual, 2016). It allows the selection of performance parameters that can be monitored. This hardware-level CPU feature provides a mechanism for software to monitor and count performance events of interest. The initial aim was to use the information obtained from these counters to tune the performance of systems and compilers.

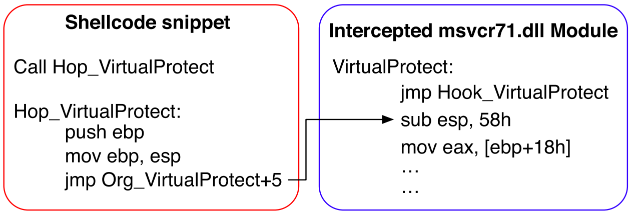
PMU supports two kinds of performance monitoring capability: architectural and non-architectural (i.e., micro-architectural). Architectural performance monitoring supports the counting and sampling of a small set of performance events such as arithmetic and branch instructions. These events are consistent across different processor implementations. Differently, non-architectural performance monitoring monitors a large set of performance events that are specific to the micro-architecture, such as cache, branch prediction and translation lookaside buffer (TLB) (i.e., micro-architectural events). These events might differ from one processor to another and might change with processor enhancements. The complete list of architectural and micro-architectural events is available at Intel's manual (Intel Manual, 2016). Here we only list the events that are used in this work in Table 1.

2.3. *Challenges in practical ROP defense*

Numerous exploit defense techniques have been proposed to mitigate ROP attacks. However, several challenges still remain.

C1. Hooking-based techniques can be bypassed by hook hopping. Some ROP mitigations (Fratric, 2012; Pappas et al., 2013) rely on API hooking to detect ROP attacks, where the hook redirects the function to verify certain properties. Such techniques are also used by commercial security software such as EMET (The enhanced mitigation experience toolkit) and antivirus tools. However, they are not robust as they can be easily bypassed by hook hopping techniques.

Fig. 3 shows a code snippet to illustrate how a few lines of assembly code defeat the inline function hooking mechanism. The shellcode contains its own pre-hook preamble. After its execution, it transfers the execution to skip the function hook *Hook_VirtualProtect*. Similarly, ROP exploit can leverage



```
Shellcode snippet

Call Hop_VirtualProtect

Hop_VirtualProtect:
        push ebp
        mov ebp, esp
        jmp Org_VirtualProtect+5
```

```
Intercepted msvcr71.dll Module

VirtualProtect:
        jmp Hook_VirtualProtect
        sub esp, 58h
        mov eax, [ebp+18h]
        …
        …
```

Fig. 3 – Hopping the *Hook_VirtualProtect* Hook.

Table 1 – Monitored hardware performance events.

Architectural Event	Description	Non-architectural Event	Description
1. Ins	instructions retired	9. Uops	micro-instructions retired
2. Clk	unhalted core cycles	10. Load_Uops	load uops retired
3. Br	branch instructions a	11. Store_Uops	store uops retired
4. Arith	rithmetic instructions	12. Br_Miss	mispredicted branch instructions
5. Call	all call instructions	13. Ret_Miss	mispredicted return instructions
6. Call_D	direct near call instructions	14. Call_D_Miss	mispredicted direct call instructions
7. Call_ID	indirect near call instructions	15. Call_ID_Miss	mispredicted indirect call instructions
8. Ret	near return instructions	16. Br_Far	far branches retired
		17. ITLB_Miss	misses in all ITLB levels
		18. DTLB_Store_Miss	store uops with DTLB miss
		19. DTLB_Load_Miss	load uops with DTLB miss
		20. ICache_Miss	instruction cache misses
		21. LLC_Ref	longest latency cache reference
		22. LLC_Miss	longest latency cache miss
		23. LLC_Load_Miss	load uops with LLC miss
		24. L2_Code_RD	L2 instruction cache access
		25. L2_Code_Miss	instructions that missed L2 cache
		26. Load_Uops_L1_Hit	load uops with L1 data hits

memory information disclosure to detect API hook and utilize specific ROP gadgets to bypass hooks.

C2. *ROP detection policies are inflexible and weak*. Some ROP mitigations (Pappas et al., 2013) use predefined policies to detect ROP attacks. Such policies are generally inflexible and weak, and can be bypassed by constructing the gadgets (or payloads) that adhere to these policies.

For example, the *call-preceded return address* policy specifies that return instructions need to target a valid call-site (a call-preceded instruction), which may include non-intended call instructions. However, this policy can be defeated, as the call-preceded ROP gadgets are turing-complete and thus, real-world ROP exploits can still be launched by conforming to this policy (Davi et al., 2014). Another commonly-used policy is the *chain of short sequences*, where a ROP attack is reported after *N* sequences each consisting of less than *S* instructions. Unfortunately, this policy can be bypassed by introducing new gadget types such as the long no-operation gadget, where the long sequence does not break the semantics of the ROP chain (Davi et al., 2014).

C3. *The performance overhead of the existing mitigations is high*. Both instrumentation-based (Davi et al., 2009, 2011; Jacobson et al., 2014) and policy-based (Pappas et al., 2013) mitigations have high performance overhead because of the inefficient instrumentations and runtime policy checking. Similarly, most HPC-based techniques (Cheng et al., 2014; Pappas et al., 2013; Tang et al., 2014; Wicherski, 2013; Yuan et al., 2011) have high performance overhead during online detection due to two main reasons. First, they use a high sampling rate, which significantly creates high-frequent interrupts to fetch HPC values and thus hinders its practical applications. Second, almost all HPC-based techniques use machine learning techniques, which cause significant runtime overhead as the classifier has to validate the trace expensively at each interrupt. Hence, most techniques do not use it at runtime, but use it during offline analysis.

It is also worth mentioning that the machine learning-based techniques need a large amount of data for training and model building. However, the number of ROP exploits in real world is significantly small, as compared to malware samples. Also, the exploits are very distinct from each other, which limits the practicality of these machine learning-based approaches. In addition, a small manipulation in the exploit technique can significantly change the used features, thus making such defense approach ineffective (Biggio, 2016).

C4. *The resource is limited*. All the HPC-based techniques are limited by the available resource w.r.t. the programmable performance counters in the CPU. This is because, current processors generally have 2–4 programmable performance counters per thread (Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3, 2016). In that sense, all previous approaches can only monitor four hardware events in addition to the fixed events (by using fixed performance counters) to model attack patterns.

C5. *Custom allocator-based spraying attacks are difficult to detect*. In the existing spraying attack detection techniques (Cristalli et al., 2016; Ratanaworabhan et al., 2009), it is not possible to detect spraying attacks that use custom allocators, which generally allocate a big chunk of memory at the beginning of the process and use their own allocation functions to perform memory operations. Custom allocators are quite common in real-world applications, which alleviate the applicability of those techniques.

3. Methodology

To address the challenges in Section 2.3, we propose our ROPSentry framework, which employs a *ROP-only defense approach* (Section 3.1) and a *self-adaptive defense approach* (Section 3.2). We build ROPSentry on the observation that the execution of a ROP attack triggers hardware events in the CPU in a significantly different way than a normal program generated by a compiler. Such hardware events are side channels that can be easily monitored through the HPCs present in every modern CPU. Therefore, a ROP attack can be distinguishable from a normal program at runtime by capturing the patterns of ROP attack from monitored hardware events. In that sense, we address the challenges *C1*, *C2* and *C5* by leveraging hardware events that are much more difficult for attackers to be controlled directly in evasion attacks than software behaviors.

Scope. In this work, we focus on both non-spraying-based ROP attacks, and spraying-based ROP attacks that use custom or system allocators, and thus provide a complete solution against the entire spectrum of ROP attacks.

3.1. *ROP-only defense approach*

In this approach, ROPSentry fetches the HPC values with a sampling rate (i.e., s_h) to detect the patterns of ROP exploits (see Section 3.1.1). If we observe that a ROP exploit pattern is occurred, we detect a ROP attack and terminate the program.

However, the sampling frequency to fetch HPC values heavily influences the practicality of the approach, i.e., a high sampling frequency improves detection accuracy at the price of performance overhead, and a low sampling frequency reduces performance overhead at the cost of detection accuracy. Based on the fact that return misses are very common in ROP attacks but not very common in normal programs because of the well-designed branch prediction unit (Branch Prediction Accuracy), we propose a novel return miss-based sampling technique to sample the hardware events, i.e., fetching HPC values at every several return misses.

Different from the existing approaches (Tang et al., 2014), which mainly use a microinstruction-based sampling technique, our approach can clearly differentiate between ROP chain execution and normal program execution and has lower performance overhead. For example, 21,918 microinstructions, on average, are executed for every 10 return misses during the whole ROP attack (for the ROP exploits we used in the evaluation), while only 30 microinstructions are executed for every 10 return misses during the specific ROP chain. Therefore, even the ROP chains that have only 30 microinstructions can be detected by our return miss-based sampling. However, if events are monitored at every 100 microinstructions, the performance overhead is increased by about 200 times. In this way, we address the challenge *C3*.

3.1.1. *ROP exploit pattern*

A ROP exploit often executes multiple gadgets to perform arbitrary computations. Each gadget is a small instruction sequence that ends mostly in a branch instruction (e.g., call, return, and jump). Most current ROP exploits use gadgets ending in a return instruction, which often cause an unexpected control flow of the program. This results in high branch mispredictions during the exploit-code execution. However, the branch prediction unit in modern processors is well designed to achieve high prediction accuracy (generally > 95%). We can only profile 4 different events (excluding 3 fixed events) out of hundreds of events at a time due to limited HPCs. Therefore, we select the most relevant events (which is 26 in total) based on our understanding of ROP exploits and hardware events used in the literature (Pfaff et al., 2015; Tang et al., 2014; Wicherski, 2013; Yuan et al., 2011) (listed in Table 1) to model the behavior of ROP exploits. For example, an abnormal control flow of the program during the execution of exploits can result into cache and TLB misses. While, a ROP chain may have a higher number of return instructions and return misses because of several return gadgets execution. We run five normal programs as well as their corresponding ROP attacks using five real-world ROP exploits from the Metasploit framework (Metasploit framework) to determine the best set of hardware events that can be used to model the behavior of ROP exploits.

It can be commonly observed from the five ROP exploits that, there is a *simultaneous* spike, as highlighted by a red rectangle in the first five rows in Fig. 4, during the exploit-code execution for the values of performance counter *Ret_Miss*, *Ret*, *ITLB_Miss* and *LLC_Miss* after the normalization by *Ret* or *Ins*. It is only observed in ROP attacks but not in normal programs, and thus can be used to clearly differentiate the ROP exploit from the rest of the code execution. For the fifth row in Fig. 4, there is a second spike for the metric *Ret_Miss/Ret*, which does not occur simultaneously in the other three metrics. It indicates that *a combination of metrics can effectively reduce false positive*.

Specifically, ***Ret_Miss/Ret*** represents the percent of mispredicted return instructions in each sampling interval. The branch prediction unit uses the call stack at runtime to predict the return address for each function return. However, a ROP exploit manipulates return instructions to execute arbitrary computations and causes abnormal program control flow. Since a ROP exploit occurs over a small length of instructions, the return miss rate in this small length becomes substantially high. As observed from the five ROP exploits, the return miss rate is in the range of 0.9 and 1.3. This means the return miss rate is close to 100% during a ROP exploit, which is a strong evidence to justify that a ROP exploit might be performed. Note that the return miss rate may exceed above 1.0 because *Ret_Miss* is speculative but *Ret* is the number of actually executed return instructions, and not all the return instructions that are speculated get executed in the execution stage of CPU pipeline. Hence, we empirically set the threshold value of return miss rate to 0.9 to detect ROP exploits, i.e., the policy 1: $Ret\_Miss/Ret \geq 0.9$.

Ret/Ins measures the percentage of return instructions in each sampling interval. As ROP exploits use return instructions to orchestrate the ROP control flow (i.e., to redirect the control flow from one gadget to another), the rate of return instructions becomes substantially high during the ROP chain. For the five ROP exploits, this ratio is between 0.2 and 0.4. Thus, we empirically set its threshold value to 0.2, i.e., the policy 2: $Ret/Ins \geq 0.2$.

ITLB_Miss/Ins measures the percent of misses in all instruction translation lookaside buffer (ITLB) levels over the total number of instructions in each sampling interval. ITLB is a cache that memory management hardware uses to improve the virtual address translation speed. It is well designed in modern processors such that the ITLB miss rate is very low, i.e., around 0.01–1% (Patterson and Hennessy, 2013). However, the gadgets in a ROP exploit are already present in the original program and spread over different program segments. As a result, the executed ROP chain code is spread over many pages and causes many ITLB misses, and thus the ITLB miss rate is substantially high. As observed from the five ROP exploits, this ratio is between 0.8 and 1.6 during the ROP chain. Therefore, we set its threshold value to 0.8 to detect ROP exploits, i.e., the policy 3: $ITLB\_Miss/Ins \geq 0.8$.

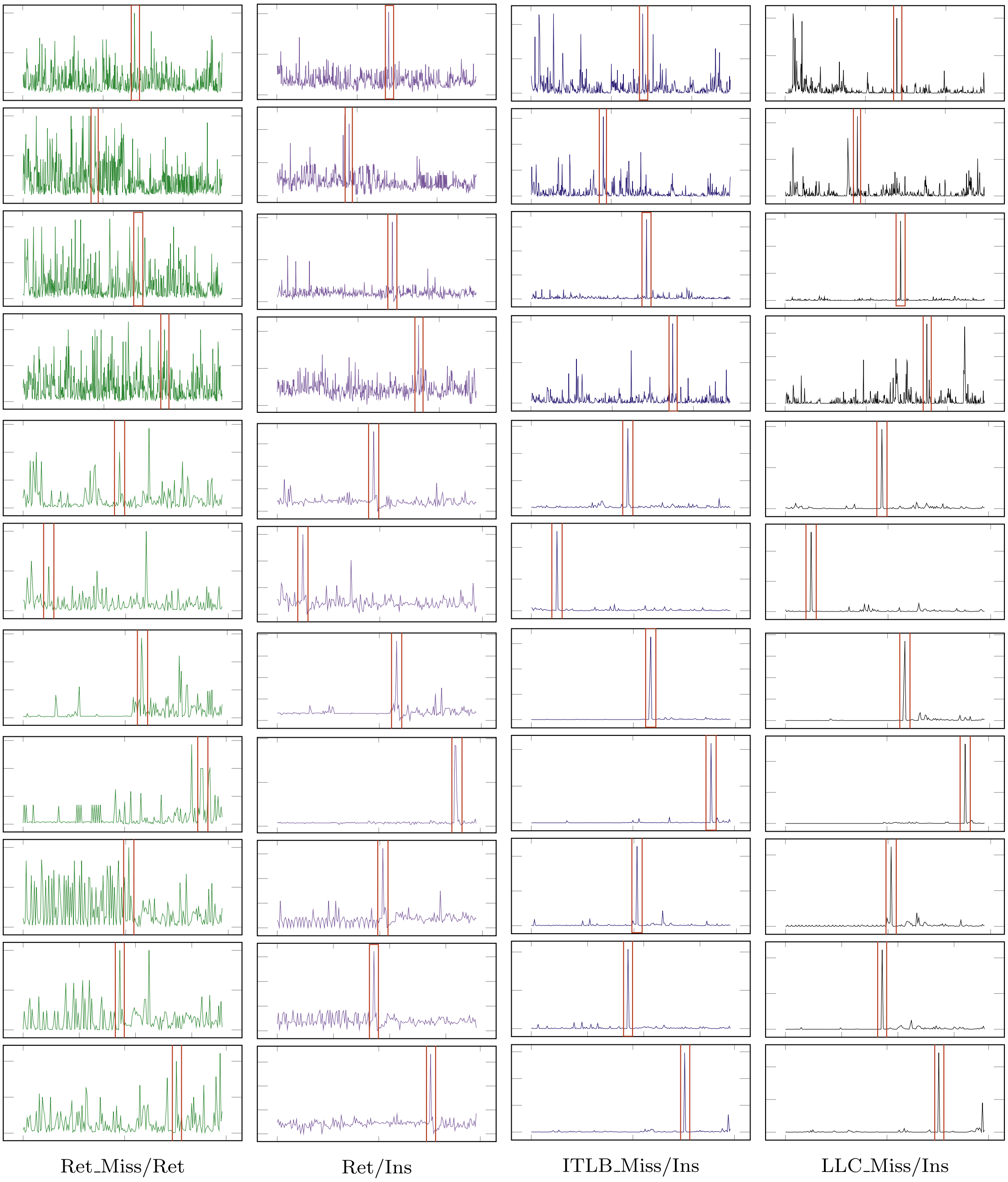


Fig. 4 – The ROP Exploit Behavior of the 11 ROP Exploits over the Execution Time. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

LLC_Miss/Ins represents the percent of misses in the last level cache (LLC) over the total number of instructions in each sampling interval. LLC is the last cache before RAM in the memory hierarchy. An abnormal control flow during ROP exploit leads to branch mispredictions and ITLB misses. While handling an ITLB miss, initial cache levels and LLC are accessed, first for page table entry and later for instructions. LLC misses occur because of the unavailability of data in LLC, eventually leading to fetch data from main memory or disk. Thus, the underlying reason for the high LLC miss rate in ROP exploits is their unexpected program control flow. In the case of the five ROP exploits, the LLC miss rate is in the range of 2.0 and 3.0. This ratio exceeds above 1.0 because LLC miss occurs not only for instructions but also for data. We conservatively set the threshold value of LLC miss rate to 2.0, above which a ROP exploit is detected, i.e., the policy 4: *LLC_Miss/Ins* ≥ 2.0.

Based on these observations from the five ROP exploits, we determine the best set of hardware events to model the behavior of ROP exploits as *Ret_Miss*, *Ret*, *ITLB_Miss*, *LLC_Miss* and *Ins*. Based on these events, we specify the pattern of ROP exploits in Pattern 1.

Pattern 1. All the policies about *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* are *simultaneously* satisfied in a given sampling interval at runtime.

The ROP-only defense approach can achieve high detection accuracy of ROP exploits but at the cost of high but still acceptable performance overhead. Note that the performance overhead is lower than the previous approaches (Cristalli et al., 2016; Tang et al., 2014) (see Section 4). To further reduce the overhead, we propose a self-adaptive defense approach below without sacrificing the detection accuracy.

3.2. *Self-adaptive defense approach*

The self-adaptive approach aims to reduce the performance overhead of our ROP-only approach, by first sampling at a low frequency to detect spraying attack that is commonly used by ROP exploits for payload delivery. The workflow is shown in Fig. 5 as a state transition diagram. In this approach, the ROP attack, when executed, can be in any of the following states: 1) normal state, 2) ROP state, and 3) termination state. In the normal state, we fetch HPC values with a low sampling rate (i.e., s_l) to detect the pattern of spraying attack (see Section 3.2.1). If we observe a spraying pattern, we suspect that a spraying attack occurs and move to the ROP state. In the ROP state, we fetch HPC values with the high sampling rate (i.e., s_h) to detect the potential pattern for any ROP exploit (see Section 3.1.1). If we observe that a ROP exploit pattern is occurred, we detect the ROP attack and terminate the program; otherwise, we move back to the normal state and switch back to the low sampling rate. The two sampling rates are empirically determined (see Section 4).

Based on the facts that, spraying attack is a common ROP payload delivery mechanism, and a spraying usually involves millions of instructions while ROP gadgets contain around 3–5 instructions, we design this self-adaptive sampling technique that respectively employs low and high sampling rates to detect the patterns of spraying attack and ROP exploit and dynamically switches between them. In this way, we address the challenge C3, i.e., keeping performance overhead at an even lower level.

Meanwhile, our self-adaptive sampling technique also dynamically configures the four programmable performance counters in the CPU to monitor two different sets of hardware events, i.e., one for detecting spraying attacks and the other for detecting ROP exploits. In this way, we address the challenge C4, i.e., breaking the limit of four programmable performance counters.

3.2.1. *Spraying attack pattern*

A spraying attack typically fills the heap with copies of NOP sled and shellcode that are large enough for the shellcode to land at a predetermined address and to execute arbitrary code after the exploitation of a vulnerability. The spraying by itself does not perform any malicious operation, but it significantly increases the reliability of executing the exploits. Therefore, a spraying attack usually performs similar memory operations and executes similar code for a number of times. Based on this understanding, we select the most relevant performance counters (including those for ROP exploits), as listed in Table 1; and run five normal programs and their corresponding ROP attacks using five real-world ROP exploits from the Metasploit framework to determine the best set of hardware events that can be used to model the behavior of spraying attacks.

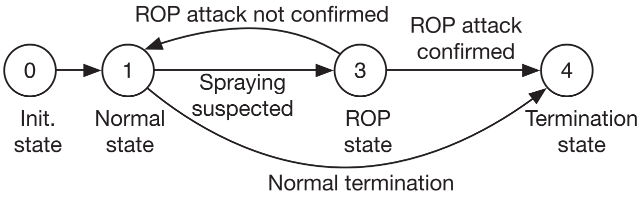


Fig. 5 – Workflow of our self-adaptive defense approach.

The commonly observed behavior across the five spraying attacks used in the five ROP exploits is that, the values of the performance counter *Br*, *Load_Uops* and *Load_Uops_L1_Hit* after the normalization by *Ins* or *Uops*, all keep nearly constant during the spraying attack, as highlighted by the red rectangles in Fig. 6. Further, this is only observed in spraying but not in normal programs, and hence can be used to clearly differentiate the spraying attack from the rest of the program execution.

In particular, ***Br/Ins*** represents the percentage of branch instructions in each sampling interval. Its value keeps consistent during the spraying attack process due to the execution of similar code for a number of times. ***Load_Uops/Uops*** and ***Load_Uops_L1_Hit/Uops*** respectively represent the percentages of load microinstructions and load microinstructions with L1 cache hit in each sampling interval. Their consistent values during the spraying attack are explained by the similar memory allocation operations to fill the heap.

Limited by the four programmable performance counters in the CPU, we determine the best set of hardware events to model the behavior of spraying attacks as *Br*, *Load_Uops*, *Load_Uops_L1_Hit*, *Uops* and *Ins*, where *Ins* can be monitored by the fixed performance counter in the CPU. Based on these monitored events, we specify the spraying pattern in Pattern 2.

Pattern 2. *Br/Ins*, *Load_Uops/Uops* and *Load_Uops_L1- _Hit/ Uops* remain consistent *simultaneously* during a given time window (i.e., w_{hs}) at runtime.

w_{hs} is empirically determined (see Section 4). Once in the normal state, we use a sliding window algorithm to check the consistency during w_{hs}. In a given window, we observe if the ratio of the previous value to the present value of counter remains around 1 at a given sampling interval. To remove the noise, we ignore some anomalous cases, which may not be consistent in a given window. If the consistency lasts for w_{hs}, we suspect that a spraying attack occurs and move to the ROP state, where we use Pattern 3 below to specify the ROP pattern, which is an extension to Pattern 1. If there is an inconsistency of the counter values for the monitored events during a given window, we move the position of the window to the next sampling interval. The only difference between Pattern 1 and 3 is that the policies will be checked for a time window w_{rop} instead of the whole running time of the ROP attack, since we need to switch back to the normal state.

Pattern 3. All the policies about *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* are *simultaneously* satisfied in a given time window (i.e., w_{rop}) at runtime.

w_{rop} is empirically determined (see Section 4). Once in the ROP state, at each interrupt, we check the satisfaction of the four policies for ROP detection. If satisfied, we confirm that a ROP exploit occurs; if not satisfied in w_{rop}, we move back to the normal state.

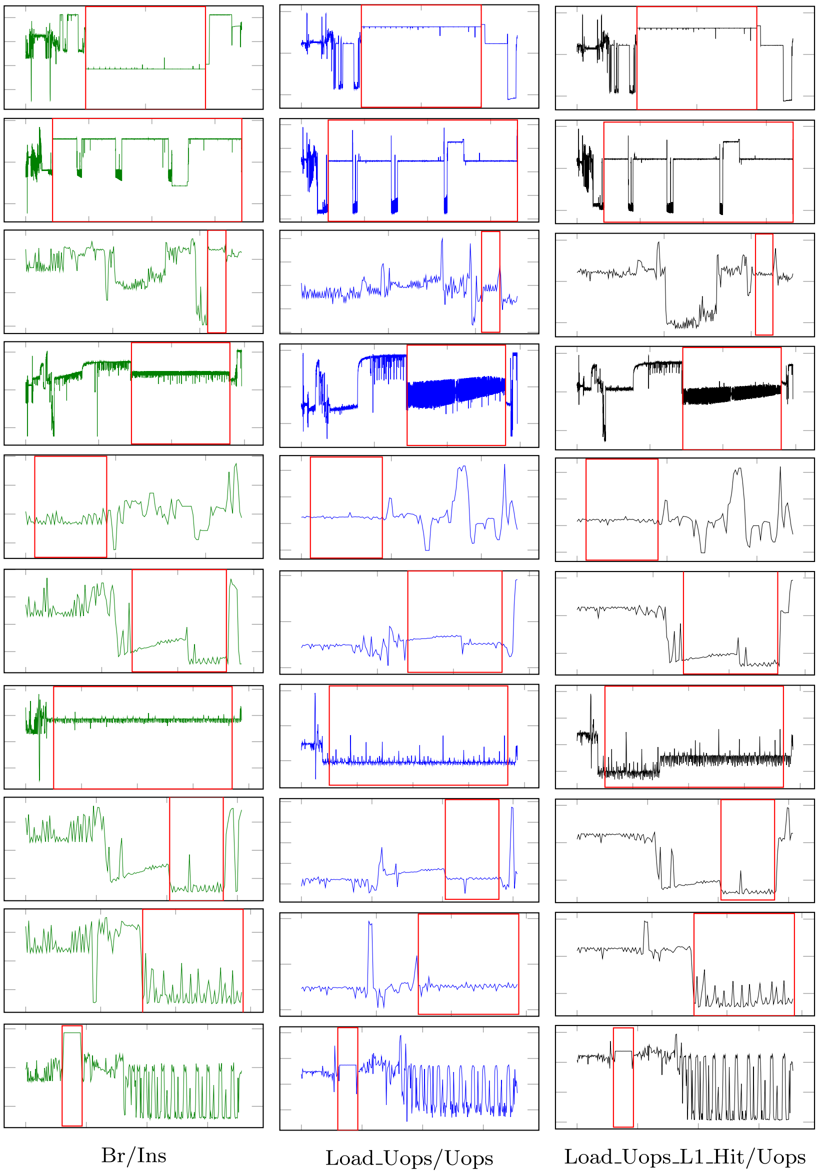


Fig. 6 – The spraying attack behavior of the 10 spraying-based ROP exploits over the execution time. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Note that our approach can only suspect but cannot confirm spraying attacks because we do not observe the content in the heap memory. Inspecting memory content at runtime will significantly impact the performance. Instead, we employ a conservative approach, i.e., every occurrence of the pattern is suspected as a spraying attack, but we do not flag it as an attack. We rather detect ROP exploit pattern in order to confirm the ROP attack. This approach is meaningful as a spraying attack is a payload delivery mechanism used by attackers to assist in executing the ROP chain.

4. Evaluation

We implement a prototype of the proposed approach on x86, a 32-bit version of Windows 7 Professional SP1. In this section, we evaluate the detection accuracy and performance overhead of the proposed ROPSentry framework. All the experiments were conducted on a virtualized environment with the following specifications: Intel Xeon Processor, 8 GB RAM, and 32-bit Windows 7 Professional SP1. We employed a reverse shell, i.e., *reverse_tcp*, from the Metasploit framework (Metasploit framework) to launch ROP exploits on Internet Explorer 8. Note that we uninstalled the Microsoft updates KB2744842 and KB2799329 in order to successfully launch the ROP exploits.

Our subjects include three benchmarks: 10 spraying-based and 1 non-spraying-based real-world ROP exploits from the Metasploit framework as listed in Table 2, 50 synthetically generated ROP exploit variants, and 1000 benign websites that are top websites in the ranking of Alexa. Among the 10 spraying-based exploits, 3 of them use custom allocators, while 7 of them use system allocators to realize spraying. Therefore, the ROP exploits we choose for our experiments are representative for the entire spectrum of ROP exploits as described in Section 2.1. Note that, the number of ROP exploit samples is small in our experiments, as is similarly done in the literature (Cristalli et al., 2016; Pappas et al., 2013; Tang et al., 2014). The reason is that the number of publicly available working exploits is small and it is also challenging to make the exploits work in a given environment.

Using the three benchmarks, we aim to answer the following research questions through the experiments.

Q1. How is the generality of the observed behavioral patterns of ROP exploits and spraying attacks?
Q2. How are the parameters (i.e., the sampling rate s_l and s_h and the time window w_{hs} and w_{rop}) tuned with respect to detection accuracy and performance overhead?
Q3. What is the accuracy of our framework?
Q4. What improvement can be achieved by our framework over the state-of-the-art techniques?

4.1. *Generality of ROP exploit and spraying attack patterns*

We used the first five ROP exploits in Table 2 to empirically specify the patterns of ROP exploits and spraying attacks (see Section 3.1.1 and 3.2.1); and launched the remaining ROP exploits in Table 2 to evaluate the generality of the patterns.

Fig. 4 and 6 respectively show the value of the metrics that are used to model the behaviors of ROP exploits and spraying attacks over the execution time. For clarity, we only show the time segments that contain the ROP exploit or spraying attack. These metrics are summarized from the first five ROP exploits in Table 2, i.e., the first five rows in Fig. 4 and 6. We can clearly see that the remaining exploits in Table 2 also exhibit similar behaviors with respect to these metrics, as highlighted by red rectangles. Specifically, during a ROP exploit, *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* all have a large spike at the same time; and during a spraying attack, *Br/Ins*, *Load_Uops/Uops* and *Load_Uops_L1_Hit/Uops* all keep almost constant despite some small spikes because of the noise from other processes. Such noise is caused as our current implementation cannot separate the performance counter events from other processes during context switches. However, it can be reduced by employing the sliding window algorithm to compute the consistency.

Table 2 – ROP exploits from the metasploit framework.

#	Description	CVEs
1.	Adobe Flash Casi32 Int. Ovf.	2014–0569
2.	Adobe Flash Domain Memory UAF	2015–0359
3.	IE COALineDashStyleArray Int. Ovf.	2013–2551
4.	IE CButton UAF	2012–4792
5.	IE Fixed Table Col Span Heap Ovf.	2012–1876
6.	IE Execommand UAF	2012–4969
7.	IE CDisplayPointer UAF	2013–3897
8.	IE Same ID property	2012–1875
9.	IE Option Element UAF	2011–1996
10.	Adobe Flash Kern Parsing Int. Ovf.	2012–1535
11.	IE CGenericElement UAF (non-spraying)	2013–1347

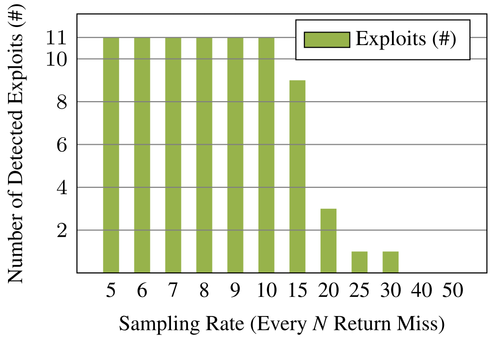


Fig. 7 – Detected exploits w.r.t. sampling rate.

Summary: these observations from Fig. 4 and 6 positively answer Q1 that the behavioral patterns can still hold for different ROP exploits and spraying attacks, and therefore can be used to model the behavior of ROP exploits and spraying attacks.

4.2. *Accuracy and overhead of ROP-only defense approach*

We tuned the sampling rate (s_h) for our ROP-only defense approach w.r.t. the detection accuracy and performance overhead using the 11 ROP exploits and 1000 benign websites.

Fig. 7 reports the number of detected exploits with respect to the sampling rate in terms of return misses (i.e., at every *N* return misses). We can see that the detection rate is high at high sampling frequency and it decreases with the decreasing of the sampling frequency. This is because when a large value for *N* is used, the small-length ROP chain cannot be differentiated from the normal benign operations.

To estimate the performance overhead, we run the Alexa-ranked websites at different sampling rates of return misses. Here we choose benign websites for performance estimation because exploits run for short time and occur rarely, whereas the benign website can run for a longer time. Since our runtime defense approach must be activated all the time, it is better to estimate the overhead by running large number of websites, as is similarly done in Cristalli et al. (2016) and Ratanaworabhan et al. (2009). In Fig. 8, we show the performance overhead that is incurred due to sampling at different intervals (by line graph) and overall detection process (by bar graph). It can be seen that the performance overhead due to sampling at high sampling frequency is much higher than that at low sampling frequency. Generally, the performance

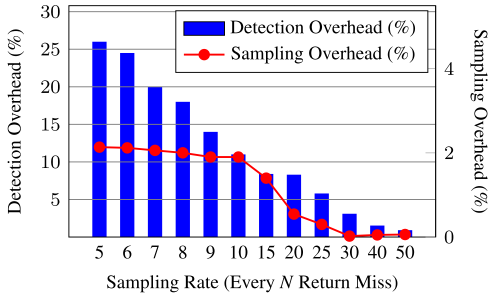


Fig. 8 – Performance overhead w.r.t. sampling rate.

overhead due to sampling is at a low level, e.g., only around 2% even with the highest sampling frequency. On the other hand, the performance overhead due to overall detection process is significantly high at the highest sampling frequency (40% when $N = 5$), which decreases at low sampling frequency. Thus, in our approach, we find that, as depicted in Fig. 7, 10 is an appropriate value of return misses for the sampling, i.e., hardware events are sampled at every 10 return misses to detect ROP chains. At this sampling rate, we observe that the overall performance overhead is around 11%.

Compared to the existing approach (Tang et al., 2014) that samples at 512K instructions, we observe that our sampling overhead (2%) is slightly higher than their approach (1.5%). However, their ROP exploit detection accuracy is < 70%, while our approach has 100% detection rate. In terms of overall detection overhead, we cannot compare with their approach, because they performed only offline analysis and had no results for online detection. However, we believe that the overall detection overhead caused by their approach, if used for runtime detection, must be significantly higher than our approach. This is because, they used machine learning-based approach to perform the detection that involves a lot of computations, finally leading to significant overhead during runtime. Comparatively, our approach is lightweight, involving only several instructions and thus having less overhead during runtime. Hence, the ROP-only defense approach, with 11% performance overhead and high detection accuracy, is already a significant achievement compared to existing techniques.

Summary: based on our study on real-world ROP exploits, we set the high sampling rate (s_h) for ROP exploits to every 10 return misses, which answers Q2. With this configuration, our ROP-only defense approach can detect all the 11 ROP exploits while keeping the performance overhead around 11%.

4.3. *Accuracy and overhead of self-adaptive defense approach*

Our self-adaptive defense approach relies on several parameters: the two sampling rate s_l and s_h to monitor the hardware events for spraying attacks and ROP exploits, and the two time window w_{hs} and w_{rop} to detect spraying attacks and ROP exploits. Since s_h was empirically set in Section 4.2, here we empirically set the other three parameters w.r.t. the detection accuracy and performance overhead.

The Time Window (w_{hs}) for Spraying Attacks. Spraying attacks usually involve millions of instructions. Fig. 9 shows the size of the sprays of the 10 spraying-based ROP exploits in Table 2 in millions of instructions and microinstructions. This ROP exploits benchmark demonstrates that, the size of a spray varies with the ROP exploit, and a spray can be observed from 30 to 2500 million microinstructions. In our approach, we safely take 20 million microinstructions as the time window for spraying attacks, i.e., if the summarized metrics are all consistent for at least 20 million microinstructions, we suspect that the pattern of spraying attack is detected and then switch to ROP detection. This time window may cause false positives for spraying attacks, but we have a further step to detect ROP chains. We use microinstruction as the unit for time windows and sampling rates as it is the unit of execution in CPU pipeline.

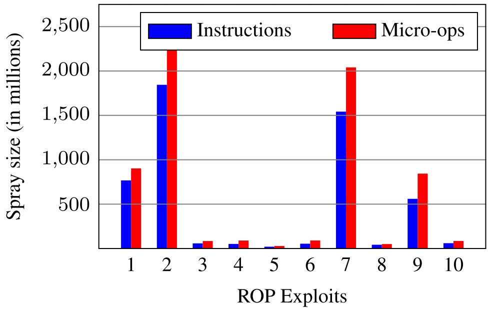


Fig. 9 – The heap spray size of the 10 ROP exploits.

Table 3 – Detected exploits w.r.t. s_l and w_{rop}.

s_l	w_{rop}					
	1000	2000	3000	4000	5000	6000
256K	3	7	5	6	9	5
512K	5	4	4	7	8	7
1024K	5	4	8	6	9	7
2048K	6	8	7	8	8	7
3072K	5	7	7	7	7	7
4096K	3	9	9	10	10	8
5120K	4	7	7	10	7	10
6144K	6	6	8	**10**	9	9
8192K	5	5	8	9	8	8

Boldfaced indicates selected configuration.

The Low Sampling Rate (s_l) for Spraying Attacks and Time Window (w_{rop}) for ROP Exploits. As shown in Fig. 9, spraying attacks occur over millions of microinstructions; and the lowest spray size is 28 million microinstructions. Thus, we explore the low sampling rate (s_l) for spraying attacks by varying it from every 256K to every 8192K microinstructions. For each fixed s_l, we also vary the time window (w_{rop}) for ROP exploits from 1000 to 6000 sampling intervals. Here s_h is set to every 10 return misses, and w_{hs} is set to 20 million microinstructions.

Table 3 shows the number of detected ROP exploits from the 10 ROP exploits for each configuration of s_l and w_{rop}. We can see that, if s_l is low (e.g., every 8192K microinstructions), our approach may miss a ROP chain after switching to ROP detection. This is because a ROP chain has only hundreds of microinstructions, while s_l is every millions of microinstructions; i.e., the ROP chain may already occur in the last sampling interval of spraying attack. On the other hand, if s_l is high (e.g., every 256K microinstructions), we need to have a large w_{rop} for ROP detection, since the ROP chain will be executed for several sampling intervals.

As shown in Table 3, a large w_{rop} does not always lead to a high exploit detection rate. For example, when s_l is at every 6144K microinstructions, more exploits are detected when w_{rop} is 4000 sampling intervals than it is 5000 sampling intervals. This is because the size of the spray in some ROP exploits is extremely large, where several switches between spraying attack detection and ROP detection happen, and the large w_{rop} may make the ROP chain executed during the spraying attack detection. As a result, the ROP chain is missed. Fig. 10 depicts such a scenario, where the ROP chain is missed with large w_{rop} as our approach switches back to the spraying attack detection

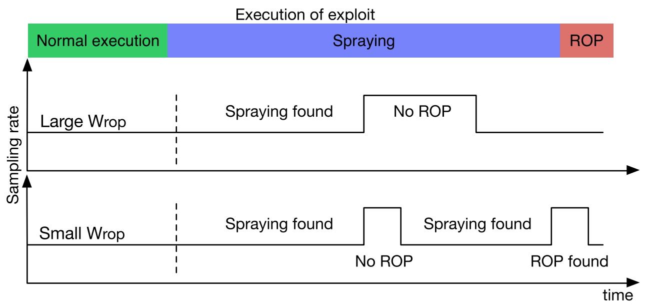


Fig. 10 – Impact of w_{rop} on ROP detection.

when the ROP chain is executed, and a small w_{rop} can detect the ROP chain.

For performance overhead, thanks to our return miss-based sampling technique and our self-adaptive sampling technique, the performance overhead on the Alexa-ranked websites is 1%, as shown in Figs. 11 and 12 w.r.t. varying s_l and w_{rop}.

Summary: based on our study on real-world ROP exploits, we set the low sampling rate (s_l) for spraying attacks to every 6144K microinstructions, the high sampling rate (s_h) for ROP exploits to every 10 return misses, the time window (w_{hs}) for spraying attacks to 20 million microinstructions, and the time window (w_{rop}) for ROP exploits to 4000 sampling intervals, which answers Q2. With this configuration, our approach can detect all the 10 ROP exploits while keeping the performance overhead under 1%. Compared to our ROP-only approach, the self-adaptive approach significantly reduces the performance overhead by maintaining the high detection accuracy.

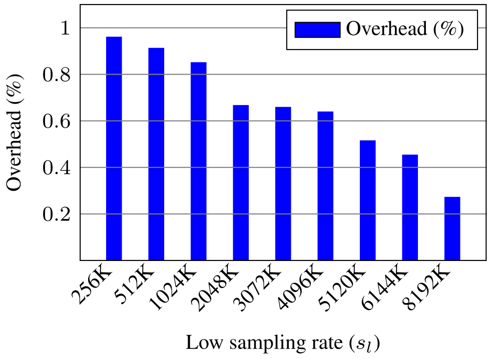


Fig. 11 – Overhead w.r.t. s_l (w_{rop} is 4000 sampling intervals).

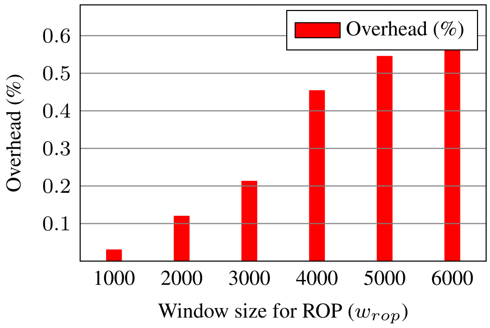


Fig. 12 – Overhead w.r.t. w_{rop} (s_l is every 6144K uops).

Table 4 – False positives in 1000 benign websites.

Policy	1	2	3	4	1,2	1,2,3	1,2,4	1,2,3,4
F.P.	948	918	149	168	117	0	0	0

4.4. *False negative and false positive*

To evaluate the false negative rate of ROPSentry, we ran the two approaches on 50 synthetically generated ROP exploits, which are variants of the published exploits presented in Table 2. Since the Metasploit framework does not allow the change of spraying attack and ROP chain, we manually created the variants of exploits by varying the spray memory allocation loops and shellcode size, following the same procedure as proposed in Ratanaworabhan et al. (2009). Both ROP-only and self-adaptive defense approaches can detect all the 50 exploit variants, which indicated the generality of the patterns of ROP exploit and spraying attack and the effectiveness of our framework.

We also evaluate the false positive of our framework by running the top 1000 websites, ranked by Alexa, on Internet Explorer 8. We loaded the first page of the website, and observed the performance counter values for spraying attacks and ROP exploits by our framework. To show the ROP exploit detection accuracy of different policies (see Section 3.1.1), we analyze the false positive with different combinations of the policies. As shown in Table 4, one individual policy leads to extremely high false positives, which is also evidenced in Fig. 4 by the many spikes during the normal executions. However, by combining policy 1 and 2 that are based on return misses and return instructions, the false positive rate is significantly reduced to 11%. After we further combine policy 3 (ITLB misses) and 4 (LLC misses) respectively, there is no false positive. The reason is that, (1) the first two policies might also hold true in certain conditions, e.g., during exceptions; however, such cases do not have high ITLB misses and LLC misses; and (2) policy 4 is closely related to policy 3 as high ITLB misses often lead to high LLC misses. Therefore, when all the policies are combined, we have zero false positive. Note that without using LLC miss, we can still achieve zero false positives (shown in Table 4). We prefer to keep LLC miss as we have the flexibility of four programmable performance counters, and it does not introduce significant overhead. Note that both ROP-only and self-adaptive defense approaches have the same result as they use the same pattern to detect ROP exploits.

Summary: based on these observations, we can positively answer Q3 that, the false negative and false positive of our framework for detecting ROP exploits are very low.

4.5. *Comparison with state-of-the-art tools*

Enhanced Mitigation Experience Toolkit (EMET) (The enhanced mitigation experience toolkit) is the-state-of-art tool developed by Microsoft that aims to make exploitation harder. Therefore, we compared our approach with EMET. EMET has several mitigation techniques that make it harder to perform the exploits; however, they are not sufficient to completely protect from the exploits. Specifically, EMET offers one

Table 5 – Comparison with EMET techniques.

Exp.	Heap Spray	StackPivot	SimExec.	Caller Check	MemProt	LoadLib.	Ours
1	×	✓	×	✓	×	×	✓
2	×	✓	×	✓	×	×	✓
3	×	✓	✓	✓	×	×	✓
4	✓	✓	×	✓	×	×	✓
5	×	✓[a]	×	×	×	×	✓
6	✓	✓	✓	×	×	×	✓
7	✓	✓	✓	✓	×	×	✓
8	×	✓	×	×	×	×	✓
9	✓	✓	✓	✓	×	×	✓
10	×	✓	✓	✓	×	×	✓
11	×	✓	✓	×	×	×	✓

[a] IE crashed but no EMET warning message.

spraying attack mitigation technique, and five ROP mitigation techniques (StackPivot, SimExecFlow, Caller Check, MemProt, and LoadLibrary (Vlaszaty and Rohani, 2014)).

Table 5 compares the exploit detection capability of our approach against EMET v5.5. As observed from our experiments, EMET techniques, including Heap Spray, SimExecFlow, Caller Check, MemProt and LoadLibrary are ineffective against most of the exploits. The most effective technique in EMET is StackPivot; however, as it performs the check at each critical API, it can be easily defeated by using the hook hopping technique as explained in Section 2.3. Differently, our approach can detect all the 11 ROP exploits using low-level hardware events. Here we do not compare their performance overhead because EMET is not open-sourced and we cannot get the detailed performance information for the comparisons.

We also attempted to compare our approach with the existing HPC-based defense techniques in terms of the detection accuracy and performance overhead. However, to the best of our knowledge, none of them are publicly available. Instead, we compared our approach with the technique in Tang et al. (2014) based on the data shown in their paper. Since the performance overhead of HPC-based mitigation is mainly determined by two factors: the sampling frequency and the data analysis technique, we conduct the comparison by these two factors. First, our sampling rate is around every 6144K microinstruction at low sampling rate, while their sampling rate is every 512K instructions. Second, we use light-weight techniques to detect spraying attacks and ROP exploits, which do not involve large amount of computation and therefore, the overall performance overhead is kept under 1%, as discussed in Section 4.3. However, they use heavy machine learning techniques to analyze the data at each PMI, which can cause high overhead as it involves large amount of computation. In comparison, their sampling at 512K itself has performance overhead of 1.5% and on top of it, the ROP detection rate at this sampling rate is just 70%. As stated earlier (Section 4.2), the overhead will be significantly high if the detection part is performed during runtime, because of the machine learning based implementation.

Besides, we compared our framework with the most recent technique Graffiti (Cristalli et al., 2016), which aims to detect spraying attacks. Graffiti can only detect system allocator-based spraying attacks, but cannot handle custom allocator-based spraying attacks. We also attempted to run our framework on the 6 exploits used by them. All exploits date back to 2011 or before. We could only run two exploits: CVE-2011-1996 and CVE-2010-2883; and our framework can detect both of them. We could not run the other 4 exploits as they can run successfully only on Windows XP, which is an old platform. However, our framework supports the more advanced Windows 7 platform. In terms of execution overhead, we have around 11% and 1% overhead using our ROP-only and self-adaptive approach, while Graffiti has 23% overhead for only detecting spraying attacks.

Summary: the comparisons with EMET and the approach in Tang et al. (2014) and Cristalli et al. (2016) answers Q5 that our framework can significantly improve performance overhead without sacrificing the detection capability.

5. Discussion

Our approach significantly raises the bar for performing a ROP attack, however, it has some limitations. First, our ROP exploit detection technique is based on the abnormal behavior in terms of return misses, return instructions, ITLB misses and LLC misses. Among them, return miss is one critical evidence in current ROP exploits. It can be defeated by some chains of gadgets: interrupt return gadgets (Li and Carlini, 2015), indirect call-gadgets-ret (COOP attack (Schuster et al., 2015)), and indirect call-gadgets-indirect jmp chains. However, these are more theoretical exploits and are not common in real world; and our approach can be enhanced with more metrics to consider indirect call and jmp. For example, we can use indirect jmp misses and indirect call misses to detect such theoretical exploits. To the best of our knowledge, no COOP-based ROP exploits are publicly available till date, and thus we cannot incorporate such type of attacks in this work.

Second, the patterns to model the behavior of spraying attacks and ROP exploits are empirically determined by only a small number of real-world ROP exploits, which may hinder the effectiveness of our approach although our experiments have shown promising results. On one hand, the number of publicly-available working ROP exploits is small (similar experience and experiments have been shown in other work (Pappas et al., 2013; Tang et al., 2014)). On the other hand, our approach can be easily extended to integrate new patterns with the emerging of new ROP exploits. For example, one can easily integrate a defense against COOP exploits (Schuster et al., 2015) by monitoring additional hardware events (e.g., indirect calls) using HPC and applying a new rule to detect them at runtime. This may not require rebuilding the defense that is already built for the existing exploits. Similarly, the four parameters (two sampling rates and two time windows) in our self-adaptive sampling techniques are also empirically determined based on a small number of real-world ROP exploits. As a result, our approach might miss ROP exploits. One possible scenario is that, after performing the spray, the attacker waits until the ROP detection duration (w_{rop}) has been elapsed (by performing some normal code execution) and the state returns to normal. During the normal state, the attacker can choose to trigger the ROP payload to bypass the detection. However, the complexity of performing a successful ROP exploit will highly increase. Because the spraying in the heap memory might have been destroyed due to normal code allocation in the heap, making the spraying ineffective. This will certainly reduce the chance of performing a successful exploit. One possible solution is to dynamically determine the sampling rates and time windows according to the behavior of spraying attacks and ROP exploits at runtime. Even though the empirical values for spraying and ROP attacks may vary with new ROP exploit samples, the key characteristics that are used by our approach to model spraying and ROP attacks will remain the same. Moreover, we have already evaluated our approach against real-world exploits and their multiple variants by changing the number of gadgets in ROP chains and spraying sizes. Our evaluation have shown a very high accuracy for these exploits. Our approach covers all the spectrum of the practically known ROP exploits. Therefore, we strongly believe that our approach can detect all the known ROP exploits.

We also want to discuss whether an attacker can create a mimicry attack by knowing the internal details of ROPSentry. The key problem with this attempt is that the attacker has no direct control to manipulate all these hardware events to perform the attack successfully. While one may be able to control architectural features (e.g., branch instructions and return instructions), it is almost impossible for the attacker to manipulate all the low-level non-architectural features such as *LLC_Miss*, *ITLB_Miss*, *Load_Uops* and *Load_Uops_L1_hit*, as these events cannot be directly controlled. It may be possible that an attacker can fool one of the hardware events, however, it is almost impossible to manipulate multiple hardware events simultaneously, while still performing a successful attack. Thus, we believe that our framework is robust against mimicry attacks.

We clarify that there may be false positives for the spraying attack model, because spraying behavior is similar to some benign behaviors. For example, if a program continuously allocates memory in the heap memory, it may appear as a spraying attack (e.g., while loading a game). However, we consider an attack, only when spraying is followed by a ROP exploit. We conservatively consider any spraying as a trigger point and then look for ROP. Therefore, even if we have false positives on spraying model, this does not affect our overall detection.

In general, common ROP attacks consist of more than 15 gadgets (Cheng et al., 2014), and most of them end in a return instruction. Accordingly, from our experiments we set the sampling interval at 10 return misses. However, we do not rule out the possibility that there may not be less than 10 gadgets, but such attacks are not common. Moreover, with less than 10 gadgets, it will be difficult to build the ROP exploit under the existing OS/compiler defenses. One can also argue that our approach can be bypassed by normalizing the return misprediction rate, such as by executing normal functions between the gadgets. While this may appear valid, it will be extremely difficult to build such a ROP chain. This is because normal code execution between gadgets will modify the registers and memory layout that are essential to perform a ROP attack. Conceptually, the gadgets are executed in order to set up some registers, memory that are essential for a ROP attack.

6. Related work

In this section, we focus our discussion on the ROP exploit defense techniques that use hardware features and are closely related to our work.

Recently, several hardware-based solutions have been proposed in the literature to overcome the limitations in program instrumentation and hooking based ROP mitigation techniques. One of the pioneers is kBouncer, proposed by (Pappas et al. (2013), that leverages on the Last Branch Record (LBR) feature available in modern CPUs to record the branch targets taken by the target program. At each system API invocation, kBouncer verifies the integrity of the running program by evaluating the proposed CFI policies against the LBR stack. Later, ROPecker (Cheng et al., 2014) extended the idea proposed in kBouncer by offline analysis and emulation in an attempt to predict ROP attacks. However, as discussed by C2 in Section 2.3, weak CFI policies can be defeated by specially crafted ROP gadgets (e.g., call-preceded gadgets) and further, API hooking is not a robust technique as it can be evaded by hook hopping (see *C1* in Section 2.3).

Yuan et al. (2011) studied how Branch Trace Store (BTS), a debugging mechanism that allows to record all branch instructions, along with several other HPC features such as ITLB misses and branch misses can be used to detect code injection attacks. Unfortunately, it incurs heavy performance overhead due to the significant number of memory accesses. Similarly, Wicherski (2013) used single branch prediction event as an indicator to detect kernel-level ROP attacks. However, it is shown that all of these techniques are vulnerable to several evasive techniques such as those in Carlini and Wagner (2014), Davi et al. (2014), and Göktaş et al. (2014). Their usage on a single type of feature (i.e., HPC event) is the root cause for their incapabilities.

HPC data has also been used for other security defenses, e.g., malware detection (Demme et al., 2013), program integrity (Malone et al., 2011), rootkits detection (Singh et al., 2017). These techniques are orthogonal to ours and are not aimed for the exploit detection. Tang et al. (2014) proposed a system to detect drive-by attacks by leveraging HPC data combined with machine learning. However, they focused more on the post-ROP behavior (i.e., malicious behavior) of the program, where they reported a high malicious behavior detection rate but a low ROP attack detection rate. Graffiti (Cristalli et al., 2016) is the most recently proposed approach, which implements a hypervisor-based memory analysis technique for detecting spraying attacks. As discussed in Section 4.5, its main limitation is that it cannot handle custom allocator-based spraying attacks and thus it cannot protect against modern spraying-based attacks. Also, it incurs much higher overhead (23%) as compared to ours (1%).

Concurrent to our work, two approaches – HDROP (Zhou et al., 2014), SIGDROP (Wang and Backer) – were proposed recently that utilize HPCs to build defense against ROP exploits. At a high level, both these techniques use mispredicted return events to determine ROP. However, HDROP requires instrumentation of source code during compilation to insert check points. Also, it has substantially high overhead (19%–30%). On the other hand, SIGDROP excercises strict policies for ROP detection, which can be defeated by the determined attacker with little effort. Its first detection policy requires that the number of return instructions must be equal to the number of mispredicted return instructions during the sampling interval. This policy can be easily bypassed by the attacker with minor changes in ROP chain. For example, an attacker can carefully insert just one redundant call-ret paired gadgets, which will not have any misprediction at return instruction due to its corresponding call instruction. Its second policy states that number of instructions in ROP chain should be less than 36 for 6 gadgets. This can be bypassed by inserting just one large NOP gadget. Also, it evaluates the performance overhead on SPEC and Unix benchmarks. These benchmarks mainly perform repetitive and similar code execution, which is likely to have low misprediction at return instructions, as compared to the complex programs such as browser. Comparatively, our detection approach is more flexible in terms of number of mispredicted return events and relies on multiple events, which are harder to manipulate simultaneously.

Similar to Tang et al. (2014), Pfaff et al. (2015) also proposed HadROP machine learning based approach to learn important hardware events to model ROP exploit and further to build the classifier. Our approach is significantly different in terms of sampling and further in analysis of ROP. They use sampling based on the fixed no. of cycles whereas we use return miss based sampling. Second, for the analysis they use machine learning based approach while we use heuristics of ROP behavior for its detection. Compared to our approach, the overall slowdown is substantially high (43%) for higher accuracy (98%) because of machine learning classifier.

7. Conclusion

We proposed and implemented a defense framework, *ROPSentry*, to detect ROP exploits at runtime in this paper. The proposed framework provides a complete solution to handle the entire spectrum of ROP attacks. ROPSentry detects ROP attacks by analyzing the behaviors of spraying attacks and ROP exploits using a combination of hardware events, and reduces the performance overhead by a return miss-based sampling technique and an adaptive sampling technique. We evaluated ROPSentry on 11 real-world and 50 synthetically generated ROP exploits and 1000 benign websites, which shows promising results that ROPSentry can effectively detect ROP exploits at runtime with low performance overhead and low false positive rate. In the future, we plan to investigate the possibility to dynamically determine the key parameters in our framework.

Acknowledgment

This research is supported (in part) by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

Alexa. Available from: http://www.alexa.com/. [Accessed January 2017].

Biggio B. Machine learning under attack: Vulnerability exploitation and security measures, in: Proc. of IHMMSEC; 2016. pp. 1–2.

Branch Prediction Accuracy. Available from: http://www.realworldtech.com/cpu-perf-analysis/5/. [Accessed January 2017].

Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses, in: USENIX Sec. Symp.; 2014. pp. 385–399.

Chen X, Slowinska A, Bos H. On the detection of custom memory allocators in c binaries. Empir Softw Eng 2016;21(3):753–77.

Cheng Y, Zhou Z, Miao Y, Ding X, Deng H. Ropecker: A generic and practical approach for defending against ROP attack, in: Proc. of NDSS; 2014. pp. 1–14.

Cristalli S, Pagnozzi M, Graziano M, Lanzi A, Balzarotti D. Micro-virtualization memory tracing to detect and prevent spraying attacks, in: USENIX Sec. Symp.; 2016. pp. 431–446.

Davi L, Sadeghi A-R, Winandy M. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks, in: Proc. of STC; 2009. pp. 49–54.

Davi L, Sadeghi A-R, Marcel W. Ropdefender: A detection tool to defend against return-oriented programming attacks, in: Proc. of ASIACCS; 2011. pp. 40–51.

Davi L, Sadeghi A-R, Lehmann D, Monrose F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection, in: USENIX Sec. Symp.; 2014. pp. 401–416.

Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, et al. On the feasibility of online malware detection with performance counters. ACM SIGARCH Comp Arch News 2013;41(3):559–70.

Fratric I., ROPGuard: Runtime prevention of return-oriented programming attacks, 2012.

Göktaş E, Athanasopoulos E, Polychronakis M, Bos H, Portokalidis G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard, in: USENIX Sec. Symp.; 2014. pp. 417–432.

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3, Tech. rep.; 2016.

Jacobson ER, Bernat AR, Williams WR, Miller BP. Detecting code reuse attacks with a model of conformant program execution, in: Proc. of ESSoS; 2014. pp. 1–18.

Li X, Carlini N. iROP: Interesting ROP Gadgets, in: SOURCE Boston Conference; 2015. Available from: http://documentslide.com/software/i-rop-interesting-rop-gadgets-v10.html. [Accessed March 2017].

Liu Z. Advanced heap manipulation in windows 8. Available from: https://media.blackhat.com/eu-13/briefings/Liu/bh-eu-13-liu-advanced-heap-WP.pdf.

Malone C, Zahran M, Karri R. Are hardware performance counters a cost effective way for integrity checking of programs, in: Proc. of STC; 2011. pp. 71–76.

Metasploit framework. Available from: http://www.metasploit.com/. [Accessed January 2017].

Microsoft corporation. Microsoft security advisory (961051); Available from: http://www.microsoft.com/technet/security/advisory/961051.mspx. [Accessed March 2017].

Multi-state information sharing and analysis center. Vulnerability in adobe reader and adobe acrobat could allow remote code execution; 2009. Available from: http://www.msisac.org/advisories/2009/2009-008.cfm. [Accessed March 2017].

Pappas V, Polychronakis M, Keromytis AD. Transparent ROP exploit mitigation using indirect branch tracing, in: USENIX Security; 2013. pp. 447–462.

Patterson DA, Hennessy JL. Computer organization and design, fifth edition: the hardware/software interface. 5th ed. San Francisco, CA: Morgan Kaufmann Publishers Inc.; 2013.

Pfaff D, Hack S, Hammer C. Learning how to prevent return-oriented programming efficiently. In: Engineering secure software and systems. Springer; 2015. p. 68–85.

Rains T, Miller M, Weston D. Exploitation Trends: From Potential Risk to Actual Risk, RSA Conference; 2015.

Ratanaworabhan P, Livshits VB, Zorn BG. Nozzle: A defense against heap-spraying code injection attacks, in: USENIX Sec. Symp.; 2009. pp. 169–186.

Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi A-R, Holz T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications, in: Proc. of SP; 2015. pp. 745–762.

Singh B, Evtyushkin D, Elwell J, Riley R, Cervesato I. On the detection of kernel-level rootkits using hardware performance counters, in: Proc. of ASIACCS, ASIA CCS '17, ACM; 2017. pp. 483–493.

Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: Proc. of SP; 2013. pp. 574–588.

Tang A, Sethumadhavan S, Stolfo SJ. Unsupervised anomaly-based malware detection using hardware features, in: Proc. of RAID; 2014. pp. 109–129.

Team P., Pax address space layout randomization (ASLR). 2003.

The enhanced mitigation experience toolkit. Available from: https://support.microsoft.com/en-us/kb/2458544. [Accessed January 2017].

Transparent ROP detection using CPU performance counters, THREADS Conference; 2014. Available from: https://www.trailofbits.com/threads/2014/transparent_rop_detection_using_cpu_perfcounters.pdf.

van de Ven A., New security enhancements in red hat enterprise Linux v. 3, update 3, Raleigh, North Carolina, USA: Red Hat. 2004.

Vlaszaty B, Rohani H. Test the effectiveness of the enhanced mitigation experience toolkit using well-known attacks on well-known binaries. Available from: https://www.os3.nl/_media/2013-2014/courses/ot/bas_hoda.pdf; 2014.

Wang X., Backer J., SIGDROP: Signature-based ROP detection using hardware performance counters, arXiv preprint arXiv:1609.02667, 2016.

Wicherski G. Taming ROP on sandy bridge, in: SysScan Conference; 2013.

Yuan L, Xing W, Chen H, Zang B. Security breaches as PMU deviation: detecting and identifying security attacks using performance counters, in: Proc. of APSys; 2011. p. 6.

Zhou H, Wu X, Shi W, Yuan J, Liang B. Hdrop: Detecting ROP attacks using performance monitoring counters, in: ISPEC; 2014. pp. 172–186.

Sanjeev Das received his PhD in Computer Engineering from Nanyang Technological University, Singapore in 2016. Currently, he is a postdoctoral research scholar in the University of North Carolina at Chapel Hill, USA. Previously, he worked as a software engineer in IBM India Pvt. Ltd. from 2010–2012. His research interests include system security, exploit, vulnerability, malware analysis and defense.

Bihuan Chen received his Bachelor and PhD degrees in computer science from Fudan University in 2009 and 2014. Now he is a postdoctoral research fellow in Nanyang Technological University. His research currently focuses on self-adaptive systems, program analysis, and software testing.

Mahinthan Chandramohan received the B.Eng. degree from Nanyang Technological University, Singapore. He is currently working toward the Ph.D. degree at the School of Electrical and Electronic Engineering, NTU, Singapore. His research interests include malware analysis, vulnerability detection at machine code level and machine learning with applications in software security.

Yang Liu received his Bachelor and PhD degrees in computer science from National University of Singapore (NUS) in 2005 and 2010, and continued with his postdoctoral research in NUS. Since 2012, he joined Nanyang Technological University as an Assistant Professor. His current research focuses on software engineering, formal methods and security, and particularly specializes in software verification using model checking techniques, which led to the development of a state-of-the-art model checker, Process Analysis Toolkit.

Wei Zhang received her PhD degree in Electrical Engineering from Princeton University with Wu Prize for research excellence. She joins Hong Kong University of Science and Technology in 2013 and establishes Reconfigurable System Lab. She was an assistant professor in Nanyang Technological University, Singapore (2010–2013). Her research interests include reconfigurable system, FPGA-based design, low-power high-performance multicore system, electronic design automation, embedded system and emerging technologies. Dr. Zhang currently serves as the Associate Editor of IEEE TVLSI, and the Area Editor of Reconfigurable Computing for ACM TECS. She has published more than 60 technical papers and authored three book chapters.