2-2014

ROPecker: A Generic and Practical Approach For Defending Against ROP Attack

Yueqiang Cheng
Singapore Management University, yqcheng.2008@smu.edu.sg

Zongwei Zhou
Singapore Management University, xhding@smu.edu.sg

Yu Miao

Xuhua Ding
Singapore Management University, xhding@smu.edu.sg

Huijie, Robert DENG
Singapore Management University, robertdeng@smu.edu.sg

ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks

Yueqiang Cheng[†], Zongwei Zhou[§], Miao Yu[§], Xuhua Ding[†], Robert H. Deng[†]
[†] School of Information Systems, Singapore Management University
[†] {yqcheng.2008, xhding, robertdeng}@smu.edu.sg
[§] ECE Department and CyLab, Carnegie Mellon University
[§] {stephenzhou, superymk}@cmu.edu

***Abstract*—Return-Oriented Programming (ROP) is a sophisticated exploitation technique that is able to drive target applications to perform arbitrary unintended operations by constructing a gadget chain reusing existing small code sequences (gadgets). Existing defense mechanisms either only handle specific types of gadgets, require access to source code and/or a customized compiler, break the integrity of application binary, or suffer from high performance overhead.**

In this paper, we present a novel system, ROPecker, to efficiently and effectively defend against ROP attacks without relying on any other side information (e.g., source code and compiler support) or binary rewriting. ROPecker detects an ROP attack at run-time by checking the presence of a sufficiently long chain of gadgets in past and future execution flow, with the assistance of the taken branches recorded in the Last Branch Record (LBR) registers and an efficient technique combining offline analysis with run-time emulation. We also design a sliding window mechanism to invoke the detection logic in proper timings, which achieves both high detection accuracy and efficiency. We build an ROPecker prototype on x86-based Linux computers and evaluate its security effectiveness, space cost and performance overhead. In our experiment, ROPecker can detect all ROP attacks from real-world examples and generated by the general-purpose ROP compiler *Q*. It has small footprints on memory and disk storage, and only incurs acceptable performance overhead on CPU computation, disk I/O and network I/O.

I. Introduction

Return-Oriented Programming (ROP) is a code-reuse security exploitation technique introduced by Shacham et al. [12]–[14]. By chaining together existing small instruction sequences (gadgets) from target programs, ROP empowers a remote adversary to perform Turing-complete computation without injecting any malicious code. Due to its great threat, recent years have witnessed many proposed methods (Table I) to defend against ROP attacks [1]–[9], [11], [15].

The approaches, such as DROP [1], ROPDefender [2], ROPGuard [3] and Return-less kernel [4], only focus on the ROP gadgets ended with return instructions (e.g., ret-based), allowing the adversary to use other gadgets (e.g., jmp-based). In addition, the first two schemes [1], [2] also incur high overhead to their protected applications. Defense mechanisms, such as CFLocking [5] and G-Free [6], aim to defend against all types of ROP attacks, but they require the knowledge of side information (e.g., source code and/or customized compiler tool chain). In fact, this side information is often unavailable to the end users in the real world.

Recent proposals, such as ILR [7], Binary stirring [8], IPR [9], CCFIR [10] and KBouncer [11], cover all ROP attack types, achieves good attack-type coverage and run-time efficiency, and requires no side information. However, they all leverage binary rewriting technique to instrument the code, in the purposes of randomly shuffling instructions, enforcing control flow integrity, or monitoring abnormal control transfers. Binary instrumentation breaks the integrity of the binary code, which raises compatibility issues against security mechanisms, such as Window 7 system library protection, Integrity Measurement Architecture (IMA) [16], and remote attestation. In addition, KBouncer [11] only monitors the application execution flow on selected critical paths, e.g., system APIs. It inevitably misses the ROP attacks that do not use those paths.

This paper presents the first generic and practical ROP countermeasure, called *ROPecker*, that effectively and efficiently defends against all types of ROP attacks without requiring source code access, customized compiler support and binary rewriting, as summarized in Table I. We observe that the distinguishing feature of an ROP attack is that *the execution flow of a victim application consists of a sufficiently long sequence of gadgets chained together by branching instructions*. Thus, ROPecker analyzes all gadgets located in the target application binary and shared libraries via offline pre-processing. During run-time check points, ROPecker identifies the gadget chain in the *past* execution flow, using the history of taken branches recorded in the Last Branch Record (LBR) registers, and also inspects the *future* execution flow to detect any ROP gadget chain, leveraging the information from offline analysis and occasional instruction emulation.

We also propose a novel *sliding window* mechanism to decide run-time check points, which sets the most recent visited code regions of the protected application as executable, and leaves the application code outside of the window as non-executable. Any attempt to execute the code beyond the window boundary automatically triggers the ROP checking logic, while the application execution within the window remains unaffected. The sliding window design takes advantage of the temporal and spatial locality of application code and the sparsely distribution of meaningful gadgets across the application code base, to achieve both run-time efficiency and detection accuracy.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
 ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23156

TABLE I. THE COMPARISON OF SEVERAL TYPICAL ROP DEFENDING APPROACHES.

	ROP Types	No Source Code	No Binary Rewriting	Run-time Efficiency
DROP [1]	Ret-based	√	X	X
ROPDefender [2]	Ret-based	√	X	X
ROPGuard [3]	Ret-based	√	X	√
Return-less Kernel [4]	Ret-based	X	√	√
CFLocking [5]	All	X	√	√
G-Free [6]	All	X	√	√
ILR [7]	All	√	X	√
Binary Stirring [8]	All	√	X	√
IPR [9]	All	√	X	√
CCFIR [10]	All	√	X	√
KBouncer [11]	All	√	X	√
ROPecker	All	√	√	√

We build an ROPecker prototype on x86-based Linux platform, though our design can be extended to other commodity operating systems. We experiment ROPecker with real world ROP attacks and those generated by the ROP compiler Q [17]. The results demonstrate that ROPecker successfully detects all of them. We also evaluate the space cost of ROPecker on the Ubuntu Linux 12.04 distribution. The experiment results show that the database for all 2393 shared libraries under under */lib* and */usr/lib* is surprising small, which can be compressed to about $19MB$ using *bzip2*. Moreover, We evaluate the performance by running several macro-benchmark (e.g., SPEC INT2006, bonnie++, and Apache server httpd) and micro-benchmark tools. The results show that ROPecker introduces reasonable performance overhead on CPU computation, disk I/O, and network I/O. Specifically, ROPecker incurs only 2.60% overhead on average on CPU computation, 1.56% overhead on disk I/O, and 0.08% overhead on typical (4KB) HTTP communications.

Contributions. In specific, we make the following contributions:

- Design the first generic and practical ROP countermeasure to protect legacy applications from all types of ROP attacks without side information and binary rewriting.
- Propose novel techniques combining sliding window, offline gadget analysis with run-time instruction emulation to achieve high efficiency without compromising detection accuracy.
- Implement an ROPecker prototype on x86-based Linux platform and evaluate its security effectiveness, space cost and run-time performance.

Organization. The rest of the paper is structured as follows. In Section II and Section III, we briefly describe the background knowledge, and highlight our system goals, threat model and assumptions. We introduce in detail the design rationale, system architecture and implementation of ROPecker in Sections IV, V, and VI, respectively. Section VII discusses the parameter effects on the performance and accuracy, and Section VIII presents the security, space and performance evaluation of ROPecker. In Sections IX and Section X, we discuss several subtle attacks and compare our system with the existing work. At last, we conclude this paper and discus the future work in Section XI.

II. BACKGROUND

A. Return-Oriented Programming

The main idea of ROP attack is to reuse instructions from existing code space to perform malicious operations. There are two major steps to launch an ROP attack: (1) to identify a set of useful instruction sequences, called *gadgets*, from the entire code segment, e.g., the application code and the shared libraries; (2) to link the selected gadgets into a gadget chain through a crafted payload. Note that the gadgets are not limited to using aligned instructions, e.g., on x86 platform, a sequence of unaligned instructions may also be converted to a valid gadget.

A typical gadget has a code section for computation operations (e.g., assigning a value to a general CPU register), and a link section manipulating the control flow to link gadgets. The control flow manipulation is achieved through the *indirect branch* instructions such as *ret* and indirect *jmp*/*call* instructions[1]. According to the difference of the link section, ROP attacks are classified into ret-based ROP and jmp-based ROP or JOP. In a real-life ROP attack, the adversary may mix both types of gadgets. The gadgets used in ROP attacks typically have the following features.

Small Size. A gadget's code section is usually small, e.g., consisting of 2 to 5 instructions [18], which leads to the lack of the functionality of a single gadget. Though the gadgets with large code sections can perform more operations, they inevitably lead to more side effects and some of them may conflict with each other, e.g., a gadget accidentally changes the stack pointer, which may lead to the failure of the execution of the next gadget. In fact, the adversary usually prefers to collect the gadgets only with the intended operations, instead of using long gadgets[2]. Thus, a real ROP attack usually needs many such small gadgets, e.g., as illustrated in [14], a jump-oriented Turing-complete JOP needs up to 34 small gadgets.

[1]Traditionally, the direct branch instructions can not be used as the link section, since their destinations are fixed. However, we identify the gadget gluing attack (Section IX-B) that may leverage direct branches to foil our detection. For completeness, we discuss the extension of ROPecker to mitigate such attacks in Section IX-B.

[2]Our mechanism can be tuned to detect gadgets of different length, as discussed in Section VII-B

Sparse Distribution. Although the gadgets are distributed across the entire code space, the ones meeting for the adversary's needs are not guaranteed to exist due to the sparse density. To have higher success probability, the adversaries usually need a large code base to collect enough gadgets to perform the malicious operations. The experiment results of Q [17] imply that the adversaries has low possibility to launch a meaningful ROP attack, if we can limit the size of the executable code within $20KB$ at any time. If we can further reduce the size, the possibility will consequently go down.

B. Last Branch Record

In our solution, we leverage Last Branch Record (LBR) registers to provide reliable information about the execution trace of the protected application. LBR are dedicated CPU registers widely available on modern Intel and AMD processors. LBR provides a looped buffer to store the sources and destinations of the most recently executed branch instructions. The length of the buffer is limited, e.g., the Intel i5 only has 16 register pairs for recording the branches. It leads to that the new records inevitably override the old ones when the LBR buffer gets full.

The LBR functionality is disabled by default, and can only be enabled/disabled through certain Model Specific Registers (MSRs). Without the kernel privilege (i.e., ring-0), the user space applications cannot modify the value in the LBR MSRs. The LBR can be configured to only record the branches taken in user space. However, when recording user-space branches, the LBR does not distinguish branches in different processes. Thus, we have to filter out the unavoidable noise records to get the ones relevant to a specific process. Note that the branch recording is performed by the hardware processor and it introduces almost *zero* overhead for application executions.

III. Problem Definition

A. System Goals

Our goal is to design a security system to *detect* and *prevent* ROP attacks at run-time with the following features.

G1: Generic. We aim to protect binary applications against all types of user space ROP attacks. The ROP gadget chain can be constructed by ret-based gadgets, jmp-based gadgets, or both of them.

G2: Transparent. Our system should transparently work for the legacy binary applications. In addition, 1) it does *not* rely on source code or customized compiler tools; 2) it does *not* instrument the binary code.

G3: Efficient. We aim to minimize the performance overhead. Our system should not incur high performance overhead to protected applications, as well as the OS and other applications that coexist on the same platform.

B. Threat Model

In this paper, we focus on defending against application-level ROP attacks. We consider a remote adversary attacking a target application by manipulating inputs in order to launch an ROP attack. We suppose that the adversary knows all implementation details of the target application and can send arbitrary inputs. Nonetheless, it cannot subvert the target platform's hardware (e.g., MMU) and the operating system services (e.g., page table permissions) to their favor.

C. Assumptions

We assume that both the processor and the operating system enable the Data Execution Prevention (DEP) mechanism. In fact, the DEP mechanism is supported by default in modern operating systems. We do not assume that the Address Space Layout Randomization (ASLR) mechanism is enabled. In essence, our attack detection mechanism does not rely on ASLR. The application is not created with malicious purpose, e.g., it is not maliciously compiled to contain abundant gadgets within a small code base. Furthermore, we do not assume that the application is released and distributed with side information (e.g., source code and debugging information). We do not attempt to protect any self-modifying applications, because they require writable code segment, conflicting with the DEP mechanism.

IV. Design Rationale

Essentially, our approach is to capture the application execution features at proper moments and then to identify the existence of the ROP gadget chain. The detection accuracy and efficiency are affected by two critical design issues. The first issue is the types of run-time features to capture. Ideally, the features should be *reliable* in the sense that the adversary can not modify them to evade detection, and *sound* in the sense that they provide a *solid* evidence to infer the presence of ROP attacks. The second is the timing of detection. A poorly designed timing may either miss critical information which leads to a low detection success rate or introduces unnecessary checks with a high performance loss.

Hallmark of ROP Attacks. Most existing ROP countermeasures are based on catching run-time abnormalities of the ROP-infected application, e.g., call-ret violation [1], [2] or deviation from the control flow graph (CFG) or call graph (CG) [5], [19]. *None* of these approaches can achieve all our goals listed in Section III. For instance, the method of catching any broken call-ret pairing is not applicable to jump based ROP attacks. It is extremely difficult to extract control flow information *completely* and *accurately*, without source code access. In Section X, we elaborate more on the disadvantages of using those abnormalities.

In this paper, we observe that the hallmark of ROP attacks is that *the execution flow of a victim application consists of a long sequence of gadgets chained together by branching instructions*. Because all gadgets can be located from the constant binary code [12]–[14], the history of branches in an execution flow, enhanced by prediction of future branches, can be the *solid* evidence to decide whether the execution comprises a gadget chain. This hallmark applies to all kinds of ROP attacks, achieving the goal of G1.

Based on this observation, we propose a novel ROP defense mechanism, which relies on a payload detection algorithm: 1) identifying past executed gadget chain using LBR information, and 2) searching potential gadget chain in future execution flow, assisted with information from offline

gadget analysis, and/or occasional run-time instruction emulation (Section V-C). If the total number of gadgets chained together in both past and future detection exceeds a chosen threshold, ROPecker reports the ROP attack and terminates the application. Note that the adversary can not tamper with the LBR values due to the hardware and the OS protection, i.e., the accessing of LBR registers needs kernel privilege. Furthermore, when the detection algorithm is executing, the monitored application is suspended and cannot change its future execution flow. Thus, all information collected is *reliable* for the detection.

An offline pre-processor is introduced to analyze the instruction and gadget information. The results cover the most frequently used gadget cases and leaves only minor cases to run-time analysis/emulation, which greatly reduces the ROPecker run-time overhead without sacrificing detection accuracy (Sections V-A and VI-A).

Timing of Checking. To meet the goal of G2, we refrain from inserting code in critical execution paths of the protected application [3], [11] to trigger our ROP checking. In addition, it is obviously inefficient to monitor every branch instruction of the application. Periodic sampling may incur less cost, but it is prone to miss the ROP attacks taking place within the sampling interval.

In this paper, to meet the goal of G3, we propose a *sliding window* technique which can catch an ROP attack in a timely fashion without a heavy performance toll (Section V-B). Shifting along with the execution flow, a sliding window refers to the portion of the application code region which are set executable by ROPecker, while the instructions outside of the window are non-executable. Therefore, attempting to execute an instruction out of the window triggers an exception where the ROP checking logic is invoked. This design is advantageous for the following two reasons. Firstly, the window size is small comparing with the large code base that is needed in launching a meaningful ROP attack. As there are not enough gadgets for an ROP attack in one sliding window, an attack always triggers our ROP checking. Secondly, it is highly efficient since there is no intervention for executions within the sliding window. In addition, the window covers the most recent, non-contiguous code pages visited by the monitored application, meaning that we can take full advantage of the temporal and spatial locality feature of the application execution [20]. This guarantees that the ROP checking logic is not frequently invoked.

A prerequisite of using the sliding window technique is that the adversary cannot bypass the checking or disable it. Despite of lacking the kernel privilege, the adversary may mislead the kernel through certain special system calls, e.g., *mprotect* and *mmap2*. The adversary may leverage ROP to invoke those system calls with malicious inputs, which leads the kernel to set the code pages of the monitored application that contains gadgets to be executable, evading the ROPecker's detection, or even to disable the DEP mechanism and allow code injection. Thus, ROPecker have to intercept those system calls and check the existence of the ROP gadget chain before passing them to the kernel.

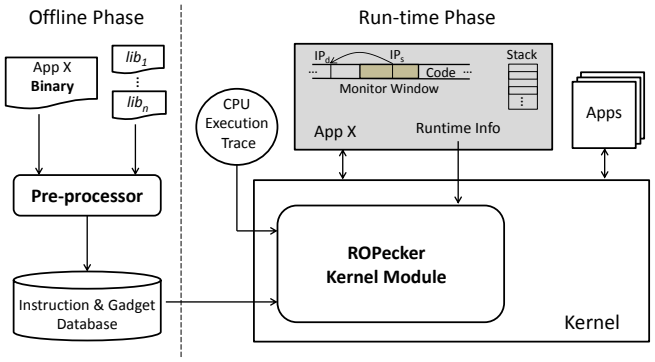

Fig. 1. **The architecture of ROPecker.** The shaded square represents a protected application.

V. System Overview

In this section, we elaborate on the design of the ROPecker system. As depicted in Figure 1, the general work-flow of ROPecker proceeds in two phases: an offline pre-processing phase followed by a run-time detection phase. In the offline pre-processing phase, the ROPecker pre-processor disassembles the binary code of the protected application X and system libraries that it depends on, analyzes each instruction and gadget in the disassembled code, and saves the results in a Instruction & Gadget (IG) database (Section V-A). During the run-time detection phase, the ROPecker kernel module that implements our ROP attack detection logic is invoked, when the application X executes outside the sliding window or invokes risky system calls (Section V-B). The kernel module suspends the application X, and performs ROP attack detection by analyzing the CPU execution trace, the stack and code segments of the protected application, and the IG database, and if necessary, invoking instruction emulation (Section V-C). Note that ROPecker can protect multiple applications in parallel.

A. Offline Pre-processing

In this phase, the ROPecker pre-processor extracts all instruction information (e.g., offsets, types, and alignment) of the protected applications as well as the shared libraries that they depend on. The pre-processor also identifies the potential gadgets from the application binary and libraries, analyzes their impacts on the code stack and the CPU instruction pointers (e.g., pop and $push$ instructions can change the CPU stack pointer). The pre-processor constructs a IG database to store all the above instruction and gadget information. The pre-processing of the shared libraries (e.g., libc) is an one-time effort, as their results can be re-used among different protected applications. Note that our pre-processor does *not* depend on a perfect disassembler that can analyze the entire application binary and figure out each instruction completely and accurately. Our pre-processor starts from the first byte of the application code segment, and tries to disassemble a pre-defined number (e.g., 6) of instructions (as is analyzed in Section VII-B). If this instruction sequence ends with an indirect branching instruction (e.g., *ret*, *jmp %eax*), and does not contain any direct branching instruction (e.g., *call 0x8049000*), it will be treated as a potential gadget. The pre-processor further analyzes all instructions in this gadget to determine its impact on the CPU instruction and stack pointers.

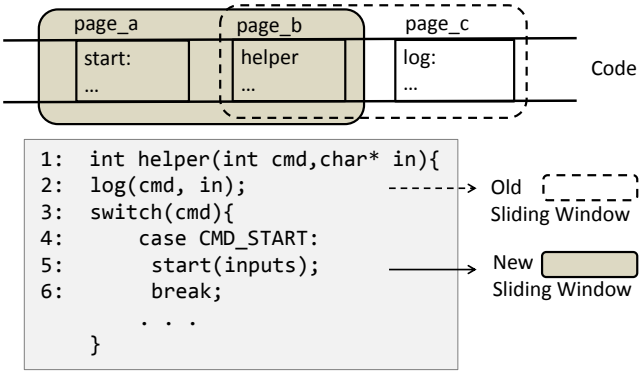

Fig. 2. **The sliding window and its update.** In this example, the size of the sliding window is 2 pages. When the execution flow reaches to *log* function (line 2), the sliding window is *(page_b, page_c)*. Later the window is updated to *(page_a, page_b)* when the *start* function (line 5) is executed. Note that the code pages in the sliding window can be non-contiguous.

This procedure is simply redone in each byte of the application binary. In short, we only require the pre-processor to correctly analyze a short sequence of code, instead of disassembling the entire application code all at once. Thus, our pre-processor can base on any existing linear-sweeping disassembler, such as objdump, or diStorm [21]. We will introduce the detailed implementation of the pre-processor and the IG database in Section VI-A.

B. ROP Detection Triggers

Before describing the run-time detection phase, we first introduce two types of events that trigger the detection logic: execution out of the sliding window and invocation of the risky system calls.

1) Sliding Window: As explained in previous sections, application code pages outside of the sliding window are *all* set as non-executable by ROPecker kernel module. Consequently, any attempt to execute an instruction out of the window triggers a page fault exception, and the exception handler invokes the ROPecker checking logic. If no ROP attack is detected, ROPecker slides the window by replacing the oldest page with the newly accessed code page. In this way, the window always covers the most recent *active* pages. For instance, the sliding window in Figure 2 consists of *page_b* and *page_c* when the execution flow reaches to the *log* function (line 2). When the *start* function (line 5) is executed, *page_a* is added to the sliding window and *page_c* is deleted.

Due to the temporal and spatial locality of application [20], the usage of the sliding window significantly reduces the number of times of the ROP checking, and thereby reduces the performance overhead. The window size is a critical parameter to keep the balance between the detection accuracy and performance. Increasing the window size can decrease the frequency of the invocation of the ROP checking and thus reduce the system performance overhead, but at the same time, it increases the possibility for the adversary to launch an ROP attack only using the gadgets within certain sliding window. According to the experiment results of Q [17], we recommend using $16KB$ or $8KB$ as the window size. Note that it is safe to use a relatively larger window in certain cases. For instance, the target application has been processed by a gadget-elimination tool, such as G-Free [6]. In that case, the gadgets in the target application are expected to be extremely sparse.

2) Risky System Calls: Some system calls such as *mprotect* and *mmap2* are risky because they can allow the adversary to disable the DEP mechanism so that no exception is raised when executing outside of the sliding window. To intercept those security system calls, we modify the system call table such that the kernel invokes the ROP checking logic prior to serving the system call request. If an ROP payload is detected in the application context, ROPecker rejects the request and terminates the application execution; otherwise it transfers the execution flow back to the requested system call handler. Note that we only intercept the system calls that are risky and invoked by the protected applications.

C. Run-time ROP Detection

The ROPecker kernel module implements the ROP checking logic which detects the existence of an ROP gadget chain in the past and future execution flow of the protected application during each trigger. Figure 3 depicts the flow chart of the detection logic. It begins with condition filtering, which filters out the irrelevant events that are not triggered by the protected application. The second step is the past payload detection, which identifies the gadget chain from the branching instruction history saved in LBRs. The last step is the future payload detection that identifies a potential gadget chain that will be executed right after the triggers. If the total number of the chained gadgets exceeds a pre-defined threshold (as is analyzed in Section VII-A), ROPecker concludes that an ROP attack is caught and terminates the application. Otherwise it updates the sliding window and resumes the original execution flow.

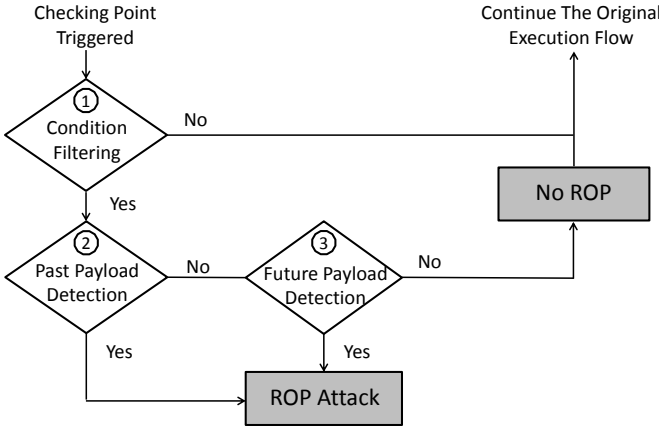

Fig. 3. **The ROP checking logic.**

1) Condition Filtering: In this step, the ROP module checks whether the triggering event is due to the target application. For page fault exceptions, ROPecker can distinguish the exceptions triggered by the sliding window from others, because the page fault error codes relevant to the sliding window indicates executing instructions in a non-executable code page, whereas the normal execution of applications never trigger page faults with this kind of error code. For system

call invocation, the kernel module compares the process ID of the invoker against the IDs of the target applications to select relevant system calls. The condition filtering is efficient since it only involves several integer comparisons.

2) Past Payload Detection: As introduced earlier, the recent branches are recorded in the LBR registers. Each LBR entry records the source and destination address of an executed branch instruction. To identify a gadget chain, ROPecker verifies the following two requirement on each LBR entry, starting from the most recent one: 1) the instruction at the source address is an indirect branch instruction, and 2) the destination address points to a gadget. Note that the verification is very efficient, because ROPecker simply queries the IG database to get the answer for 1) and 2). Once one LBR entry fails to satisfy *either* requirement, ROPecker stops LBR checking and outputs the length of the identified gadget chain. If the identified gadget chain length exceeds the threshold, ROPecker reports the ROP attack and terminates the application. Otherwise, it moves to the future payload detection with the length of identified chain as input parameters.

Recall that the LBR value is not application/process specific. Thus, we face the challenge of sifting out those records belonging to the protected application, since noisy records may increase the false negative or false positive rates of the ROPecker attack detection. Our solution is to traverse the LBR queue backwards starting from the most recent one, until we reach one LBR entry that indicates a context switch. The context switch entry can be recognized by its unique feature. Namely, its source address is in kernel space and its destination address is in user space. Other types of branches must have the source and destination addresses in the same space. The ROPecker module does not use LBR entries older than the context switch one, because the older entries are likely noisy records (e.g., belong to unprotected applications). Interestingly, if the context switch entry with its source address in the ROPecker module, it is exactly the mark left by the previous detection. In this case, the following LBR entries are still in our monitored application.

Another challenge is from the limitations of the LBR mechanism. In LBR, the new records will override the old ones when the LBR buffer gets full, and the current implementation of LBR does not provide any mechanism to intercept the override event or to backup the previous ones. Thus, ROPecker can not get the whole historical taken branches. In addition, the noise records may occupy several entries in the limited LBR registers. Thus, in certain cases, the identified gadget chain is not long enough to confirm an ROP attack, necessitating additional information collecting from the future payload detection.

3) Future Payload Detection: This step predicts any incoming gadget along the execution flow. According to the gadget types, i.e., ret-based and jmp-based gadgets, we apply two methods to handle them respectively.

It is relatively easy to handle ret-based gadgets, because their destinations are stored on the application stack and the relative positions can be calculated according to the gadget instructions. For instance, in the gadget *(pop %eax, pop %ebp; ret)*, the two *pop* instructions move the stack pointer 8 bytes towards the stack base, while the instruction *ret* retrieves the top integer of the application stack as its next destination and increases the stack pointer 4 bytes at the same time. Following their stack operations, we can correctly re-position the stack pointer and get the destination of the next gadget. The ROPecker pre-processor performs this costly instruction analysis and stack pointer calculation offline and stores the results in the IG database (Section VI-A). At run-time, the ROPecker kernel module simply queries the database and retrieves the results efficiently.

Dealing with the jmp-based gadgets is more challenging, as their destinations are dependent on the semantics of the instructions in the gadget. It is obvious that enumerating all possible combinations of a gadget in the offline processing phase consumes enormously large storage space. We propose to emulate such jmp-based gadgets at run-time to reveal the destinations of the jump instructions. Since most gadgets only have a few instructions, and the number of jmp-based gadgets are relatively smaller comparing with ret-based ones in a normal application, the overhead of gadget emulation is limited. Moreover, because the offline databases cover most popular gadget cases, ROPecker seldomly triggers its emulation step. The performance evaluation results in Section VIII-C2 also demonstrate this point.

To avoid any side effect on the actual context due to the emulation, we build a new instruction emulator that emulates the instructions in a temporally created environment with an initial context identical to the present one. The implementation details of the emulator is provided in Section VI-B4.

VI. Implementation

To evaluate the effectiveness and performance of our approach, we have implemented a prototype ROPecker for x86 32bit Ubuntu 12.04 with kernel 3.2.0-29-general-pae. The ROPecker system consists of two key components: 1) an offline pre-processor which aims to generate a database accelerating run-time checking, and 2) a loadable kernel module which responds to the run-time events and invokes the ROP checking algorithm.

A. Offline Pre-processor

Given a target application, the ROPecker pre-processor collects the application binaries and their dependent shared libraries. For each binary, the pre-processor extracts the size of the executable code segments from the binary header. For a code segment of n bytes, the pre-processor allocates a bit-vector of $n/2$ bytes, such that each byte in the executable code segment corresponds to a 4-bit slot in the bit-vector. The slots are indexed by the byte offset in the executable code segment. As shown in Table II, the slot values are assigned according to: 1) the type and optional alignment information of the instruction starting from this byte; and 2) the type and stack modification behavior of the gadget (in our case, a maximum of six contiguous instructions that ended with an indirect branch) starting from this byte. During emulation, instruction information is used to decide whether the end of the emulated gadget is reached, which is an indirect branch instruction. During run-time gadget chain detection, we use the gadget information to identify gadgets and their chaining points (e.g., a gadget manipulates the stack pointers

TABLE II. THE FORMAT OF THE BIT-VECTOR TABLE.

Value	Instruction Information		Gadget Information	
	Instruction Type	Emulation Decision	Gadget Type	Payload Detection
0000	Aligned non-branch	Continue emulation.	Not a gadget	Stop gadget chaining
0001	Aligned direct branch	Stop emulation	Not a gadget	Stop gadget chaining
0010	Aligned indirect branch (ret)	Stop emulation	Ret-based gadget	Stack offset 4
0011	Aligned indirect branch (call *,jmp *)	Stop emulation	Jmp-based gadget	Need emulation
0100	Aligned non-branch	Continue emulation	1) Stack pivoting	Need emulation
			2) Jmp-based gadget	
			3) Stack offset too large	
0101 to 1110	Aligned non-branch	Continue emulation	Ret-based gadget	Stack offset 4*(Value-4)
1111	Unaligned	Stop Emulation	Unaligned gadget	ROP Found

to point to the next gadget). The bit-vector tables from all binaries form the IG database. The IG database covers the most frequently used instruction types, gadget types and their stack manipulation behavior, and leaves only the rare cases to the run-time emulator. This design minimizes the invocations of our emulator and thus reduces the run-time detection overhead.

To build the IG database, we develop a tool to analyze the gadget information of a given binary, based on the disassembly library diStorm [21] (around $11K$ SLOC). Specifically, for each byte in the executable code segment, we run the gadget-analysis tool to disassemble six instructions starting at that byte. If an indirect branch instruction is found, we further analyze the stack manipulation behavior of the gadget, and store the result in the IG database. The alignment information in our implementation is extracted from *readelf* and *objdump* outputs. Note that we could use other disassembler tools (e.t., IDA Pro or the tool in [22]) to verify the alignment information.

The alignment information is **optional** since it is platform-dependent and the alignment analysis may not be completely accurate for certain applications. Thus, end users can selectively or completely disable the alignment checking. The main benefit of the alignment checking is to increase the bar for launching ROP attacks on x86 platform, because the adversary has to give up using unaligned gadgets. By doing so, the adversary generally needs more gadgets to launch an ROP attack. According to the analysis on the Q data set, the adversary generally needs two more gadgets for an ROP attack. Moreover, the adversary also needs more code base. For example, Q generally requires $100KB$ or more code base for constructing an ROP payload with only aligned gadgets.

We choose bit-vectors as our pre-processing database instead of hash tables, because the bit-vector indexing/query is more efficient and stable. For the bit-vector, the time cost of a query is only one memory access without false positive, while for a hash table, the time cost of a query is the computation cost of the hash function plus one or more memory access if there is a collision. In addition, the bit-vector is also space efficient, as indicated in our experiment (Section VIII-B).

B. Kernel Module

We build the ROPecker module as a loadable kernel module, which can be automatically installed when the system boots up. The module consists of $7K$ SLOC, where a large portion (around $4.4K$) is attributed to the instruction emulator [23]. The ROPecker module undertakes four main tasks: 1) to set up ROP checking triggers; 2) to collect the address locations of the shared libraries, 3) to check the branching history; and 4) to emulate instructions.

1) ROP Checking Triggers: The module inserts hooks in the Interrupt Descriptor Table (IDT) and the system call table to intercept the page fault exceptions and risky system calls, respectively.

Sliding Window Setup. We use the NX (Never eXecute) page table permission in any DEP-capable modern processor, to set up the sliding window. ROPecker initially sets the NX bits in the loaded virtual memory pages of the application and library code. During execution, the application will trigger page faults by trying to execute code in non-executable pages. Once the page fault is captured, the ROPecker module identifies the relevant faults by using process ID and page fault error code. The error code is a special combination $(PF\_INSTR|PF\_USER|PF\_PROT)$, which means that the page fault exception is triggered due to the protection violation during instruction fetching from user space. In the normal application execution, the code (NX pages) without execution right will never be executed. Thus, this error code confirms that the exception is triggered by the sliding window mechanism.

Risky System Call Interception. The system calls to intercept are *mprotect*, *mmap2* and *execve*. For the *mprotect* and *mmap2* calls, the module checks the request before passing it to the kernel. Any request to change a read-only code region to writable is rejected. Any request to change the data regions to executable is rejected. Then, the module invokes the ROP checking algorithm to ensure that there is no ROP gadget chain in the current stack. Otherwise ROPecker will return the system call with an error code to indicate the failure of the request.

The system call *execve* is able to start a new process with some prepared input parameters. For instance, the adversary may use the libc function *system* or directly invoke the system call *execve* to open a shell, in which the adversary is able to execute arbitrary commands. The end user can create a simple policy: ROPecker directly rejects all such requests for all selected applications. In most cases, it works well. However, for some legitimate applications, they may also send such requests for certain purposes. The first policy may lead to the non-coexistence of ROP with these applications. To support such applications, ROPecker can be configured to launch the ROP checking algorithm to verify if there is an ROP attack. If not, ROPecker passes the request to the kernel.

By doing so, ROPecker and such applications can coexist, but the performance overhead may slightly increase due to the extra checking.

2) Memory Mapping Acquisition: ROPecker should acquire the virtual memory mapping of the protected application and its shared libraries. If the Address Space Layout Randomization (ASLR) mechanism is enabled, the shared libraries are loaded to random addresses in different application instances. In the commodity OSes, there is *no* exported interface for kernel modules to get the mappings of a particular application. Thus, in order to locate the exact memory mappings for the target application, the ROPecker module has to intercept the mapping-manipulation operations or analyze the corresponding kernel data structures to construct the mappings.

Operation Interception. For the memory mapping manipulation operations, they are driven by the application requests through certain system calls. In a Linux system, the kernel reserves the mapping region for the application binary in the system call *execve*, while in the system call *mmap2/mummap*, the kernel reserves/releases the mapping region for shared libraries. By intercepting such system calls, ROPecker can get the mapping information from the parameters and the return values. For instance, in the system call *mmap2*, the base address of a shared library can be obtained either from the return value or the first parameter, and the length of the occupied memory region can be obtained from the second parameter. In addition, we need to intercept the *open* and *close* system calls, as they provide the name of the shared libraries. The names will be used in the database installation phase.

Data Structure Analysis. Certain shared libraries (e.g., the *library loader*) are loaded by the kernel by default, rather than being driven by the application request. Thus, we can not find their mapping information by intercepting system calls. To handle such cases, we can analyze the corresponding data structures to get the needed information. In a commodity operating system, the kernel usually has one or more dedicated data structure to maintain the memory mapping information. The Virtual Memory Area (VMA) is the data structure in Linux that maintains the start and end of memory mappings/segments. Each library is represented by a VMA structure (i.e., memory segment). The name of the shared library is also linked to the VMA structure. In addition, all VMAs are linked together and the list header can be easily found following the task structure. Thus, by traversing the VMA list, we can easily locate the corresponding VMA and get the memory mapping information. Note that the traversing is only done once, since the library locations are fixed when the application starts to run.

3) LBR Access: The $IA32\_DEBUGCTL$ MSR is the control register in the LBR mechanism. Through this MSR, ROPecker can enable, disable and manipulate the recording behaviors, e.g., it allows the LBR to only record user space branches. The LBR values are stored in the MSR registers, from *MSR_LASTBRANCH_**k**_FROM_IP* and *MSR_LASTBRANCH_**k**_TO_IP*, where k is the number of the branch record, e.g., the range of k is 0-16 on the Intel i5 processors. To read and write such LBR MSRs, ROPecker must leverage the privileged instructions such as *rdmsr* and *wrmsr*.

4) Instruction Emulation: The ROPecker module creates a shadow environment for emulating instructions. In the shadow environment, the initial context is initialized using the context of the interrupted application. It is challenging to create the shadow virtual address space. The virtual address space is quite large, e.g., $3GB$ in an x86-32bit system. To get a complete copy, the time and space cost will be extremely high. In addition, the shadow environment will be immediately dropped when the current round of the checking finishes. To save cost, we borrow the *copy-on-write* idea from the Linux kernel. Specifically, we make the virtual address space read-only for the emulator, meaning that the emulator can freely *read* any address but can not write. When a write operation is needed, ROPecker creates a mapping table which records the destination address together with the new value. Later, any reading or writing to this address will be redirected to the corresponding table entry. After the checking algorithm, the mapping table is cleared for avoiding polluting the next round emulation.

VII. Parameter Effects

In this section, we discuss and evaluate the effects of ROPecker parameter tuning, in terms of performance and security (accuracy).

A. Gadget Chain Threshold

The gadget chain threshold affects the performance of the monitored processes. Bigger threshold generally incurs higher performance overhead. However, the performance degradation is very limited according to our experiment results. Figure 4 indicates that ROPecker stops the ROP detection after one or two attempts in up to 96.8% cases. The gadget chain threshold also affects the detection accuracy. An ideal threshold should be smaller than the minimum length min_{rop} of all ROP gadget chains, and at the same time, be larger than the maximum length max_{nor} of the gadget chains identified from normal execution flows. The threshold can be any number between max_{nor} and min_{rop}, and the choice does not affect the performance, because the ROP detection algorithm always stops before it reaches the threshold. If max_{nor} is larger than min_{rop} for certain applications, it is impossible to find out an ideal threshold and ROPecker inevitably produces false positives and/or false negatives.

To find max_{nor}, we measured the lengths of gadget chains from many normal applications, which include 17 popular Linux tools (e.g., *grep*, *find*, *less* and *netstat*) under directory /bin and /user/bin, 12 benchmark tools of SPEC INT2006, and 3 *large* binaries (the video processing tool ffmpeg 2.2.0, the graphic processing tool graphics-magick-1.5.1, and the Apache web server httpd-2.4.6). In our experiments, ROPecker counts the lengths of the identified gadget chain in each invocation. As shown in Figure 4, no gadget is detected in 84.37% of the ROPecker invocation. The counts of detected gadget chain drops when gadget chain length increases. The detected chains with 6, 7, 8, 9 and 10 gadgets together only occupy 0.00006% in the total measurements. 10 is the largest length of gadget chain we have detected in the experiments. Note that the percentage results also reflect the efficiency of our ROP detection algorithm on normal applications, i.e., in the major cases (84.37%) the future payload detection stops the analysis

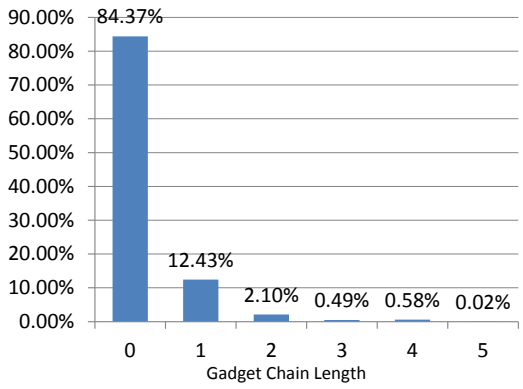

Fig. 4. **The length of payload measurement results.** The number (0 to 5) is the gadget chain length. In the major cases (84.37%), there is no gadget identified at all.

after one attempt. To decide min_{rop}, we also measured the ROP payload collected from the real world ROP attacks and generated by the general-purpose ROP compiler Q [17]. The shortest aligned gadget chain contains 17 gadgets and the longest one has 30 gadgets. Based on the above results, we can safely choose a number from 11 to 16 as the threshold for major applications. Given a specific application, we can even choose a smaller threshold. For instance, the longest gadget chain in Apache server has only 4 gadgets, so the threshold for Apache can be 5. In this way, any gadget chain that attempts to use more than 5 gadgets will be detected.

B. Gadget Length

In our implementation, we assume a gadget contains no more than 6 instructions. This is true for most of gadgets in the real-world ROP attacks. However, a smart adversary may attempt to use *long gadgets* (with more than 6 instructions) to bypass our ROP detection. Fortunately, long gadgets usually introduce more side effects, and hence the adversary has less opportunity to collect enough *suitable* long gadgets to form a valid chain for a meaningful ROP attack. If an adversary has to use consecutive short gadgets, it is likely to be detected by ROPecker. However, a smarter adversary may carefully insert long gadgets into consecutive short gadgets to make the length of each segmented gadget chain not exceed the gadget-chain threshold, e.g., use a long gadget for every 10 gadgets. Counting possible gadgets in a single window transition only, ROPecker cannot reveal this type of ROP attack.

To *mitigate* long gadget attacks, we propose a solution to extend ROPecker. When mixing short gadgets with long gadgets, the adversary creates many segmented gadget chains though each of them is shorter than the threshold. If we accumulate their lengths in several consecutive sliding windows, the accumulated length may longer than the maximum one collected from the normal execution flows. We evaluated this extension in the experiment, where we choose to measure *three* consecutive sliding windows and count the total number of gadgets. According to the measurement results, the minimum accumulated length of the tested ROP attacks collected from real world samples and generated by Q is 43, while the maximum accumulated length of the major tested applications is 13. The gap still tolerates a certain degree of long-gadget attack, allowing our algorithm to distinguish ROP attacks from normal executions. For a particular application, the gap is even larger, e.g., the maximum length of *htediter* is 9. Even so, if the adversary inserts too many long gadgets, our detection sill fails. The *extra* space cost for the multiple-window accumulation extension is small, since ROPecker only needs a short cycle buffer (i.e., the length is 3 in our example) to maintain the gadget lengths. In addition, the *extra* time cost is also small, because the checking only needs one integer comparison.

TABLE III. THE CONFIGURATIONS OF THE EXPERIMENT MACHINE.

Configurations	Descriptions
CPU	Intel i5 M540 with two 2.53GHZ cores
Memory	4GB DDR3 1333MHZ
Network Card	Intel 82577LM Gigabit
Disk	320G ATA 7200RPM

C. Sliding Window Size

The sliding window size is also a critical parameter for ROP detection. Larger window size offers better performance, but it may give more opportunities for the adversary to collect enough gadgets within the sliding window to launch a meaningful ROP attack. Thus, we should select a proper window size to balance the performance and security. According to the statistical results from Q [17], the adversary has low possibility to launch a meaningful ROP attack, if we can limit the scope of the executable code at any time within $20KB$. To be safety, we recommend setting the window size smaller than $20KB$. In our experiment we set the window size as $8KB$ (2 pages) and $16KB$ (4 pages). According to the performance results in Section VIII-C, the performances with larger window size are generally better, but the performance differences are not that big. Even in some cases, they can achieve almost the same performance.

VIII. EVALUATION

We evaluate the security effectiveness, the space cost and the performance overhead of ROPecker. For the security evaluation, we verify our algorithm with real ROP attacks and those generated by Q. For the space evaluation, we measure the disk and memory cost consumed by ROPecker. For the performance evaluation, we use the SPEC CPU2006 benchmark suite [24], which is processor, memory and compiler stressing, Bonnie++ [25] which is disk I/O stressing and Apache server httpd-2.4.6 [26] which is network stressing. The source code of the SPEC CPU2006 benchmark suite is compiled with gcc, g++ version 4.6.3 with the default Makefile. We run our experiments on a system with the configurations shown in Table III.

A. Security Evaluation

To verify the strength of ROPecker, we conduct several tests on two real-world examples and a number of payloads generated from Q. The experiment results show that ROPecker can detect all these attacks with zero false negatives.

In the first test, we use a small program (demonstrated in ROPEME [27]) that has a simple stack buffer overflow triggered by a long input parameter. We use ROPEME to

analyze the program and generate usable gadgets. We manually chain them together to craft an ROP attack that causes the program to start a shell which can facilitate the following attacks. We then verify our system with a realistic program: a Linux Hex-editer (*htediter*) (2.0.20). The ROP example on the htediter can be found on the web site [28]. An appropriately chosen long input can trigger a stack overflow. The template of the ROP payload is also available on the website. We only replace the gadget pointers according to the complied binary in our system.

At last, we test ROPecker against the gadget chains generated by Q. Q uses semantic program verification techniques to identify the functionality of gadgets and generates a valid ROP payload. Specifically, we generate the payloads for the applications under directory */bin/* and */usr/bin/*. To avoid manually exploiting each application, we build a new tool as a simulator to simulate the application execution environment. We assume that the adversary has compromised the process and successfully make the execution flow start from the gadget chain. Once a gadget jumps out of the simulated sliding windows, an algorithm will count the number of gadgets in the rest of chain. If the number is larger than the threshold, an ROP attack is identified. Specifically, the detection algorithm itself can successfully detect 100% all the payloads generated from 253 applications under directory */bin* and directory */usr/bin/*.

B. Space Evaluation

The space cost includes disk and memory costs, which are mainly from the generated databases. In our experiment, the databases for all 2393 shared libraries under */lib* and */usr/lib* of the Ubuntu Linux 12.04 distribution is about $210MB$. On average, a database is $90KB$. For example, the database of libc-2.15 is $832KB$, and the ld-2.15 database is $68KB$. *Note that the size of a database is only related to the size of the original binary file, rather than the detection parameters (i.e., gadget length, gadget chain threshold and sliding window size).* To further reduce the space cost on the disk, the database can be compressed. The experiment results show that all databases of shared libraries can be compressed to about $19MB$ using *bzip2*. The compression rate reaches up to 91%, which also indicates that the potential gadgets are sparse in the binary files.

At run-time, the databases are dynamically loaded into memory according to the demands of the monitored processes. To further reduce memory cost, we allow the loaded databases to be shared by all processes. For instance, if one process has loaded the *libc* database, all other monitored processes can share it without reloading. Thus, the memory cost of ROPecker is not large. Even if ROPecker loads all databases of the shared libraries into memory, the memory cost is still acceptable. The memory space effects from gadget chain threshold, gadget length and sliding window size are quite small and limited. To maintain n sliding windows for a process, ROPecker only needs an n-length list recording the base addresses of the code pages within the sliding windows. Due to the smallness of gadget length and gadget chain threshold, they only need *limited* data structures *temporal* available in the gadget chain identification procedure.

TABLE IV. THE MICRO-BENCHMARK RESULTS FOR SYSTEM CALL INTERCEPTION.

Operation	open	close	execve	mmap	mprotect
Time (μs)	0.03	0.04	0.86	0.05	0.03
Operation	munmap	pre-exception		post-exception	
Time (μs)	0.04	0.03		0.01	

TABLE V. THE TIME COST IN ROP CHECKING.

Operations		**Time** (μs)
Past Gadget Chain Detection		0.07
Future Gadget Chain Detection	W/o Emulation	0.91
	W/ Emulation	2.61

C. Performance Evaluation

In this experiment, we use micro- and macro-benchmark to evaluate the performance of ROPecker. We also evaluate its performance impact to the target applications and other applications in the same platform.

1) Micro-Benchmark: The micro-benchmark is used to evaluate the time cost of each operation introduced by ROPecker. The time cost introduced by system call interception may affect the performance of other applications, but other operation costs (e.g., database installation) are confined to the protected application.

To measure the time cost on the other applications, we run ROPecker without any target application. The system call interception code in the *execve* needs to check the application name, while others only check the process ID (PID). The checking costs are listed in Table IV. The experiment results show that the time overhead introduced to other applications is quite small.

We then use the *htediter* as an example to measure the time cost of loading the IG database. Note that the time is dependent on the size of the database. For instance, it takes $283\mu s$ to load the database for libc-2.15 ($849.93KB$), and $51.07\mu s$ for ld-2.15 ($65.55kB$). Although the loading is relatively high, it is only performed once in the whole life cycle of the protected application. In addition, the database installations are mostly done in the application initialization stage, before the application starts execution. Note that the databases of the shared libraries will not be re-loaded if they have already been loaded by other protected applications.

We also measured each step of the ROP checking algorithm. As shown in Table V, the performance overhead is quite low. Note that the total time cost for a particular checking is not the sum of all these steps. The reason is that some branches (e.g., the payload checking with emulation) may not be taken in that round. In fact, payload detection (with emulation) is rarely performed. In our test cases, the emulation is only invoked in about 1.7% of all detection invocations.

2) Macro-Benchmark: The macro-benchmark tests the overall performance impacts of ROPecker on the target application and the system. We choose three aspects to evaluate: CPU computation, disk I/O and network I/O. Considering accuracy, we set the size of the sliding window always under the upper bound ($20KB$). Specifically, we choose $8KB$ (2 pages) and $16KB$ (4 pages) as the size of the sliding window in our experiments.

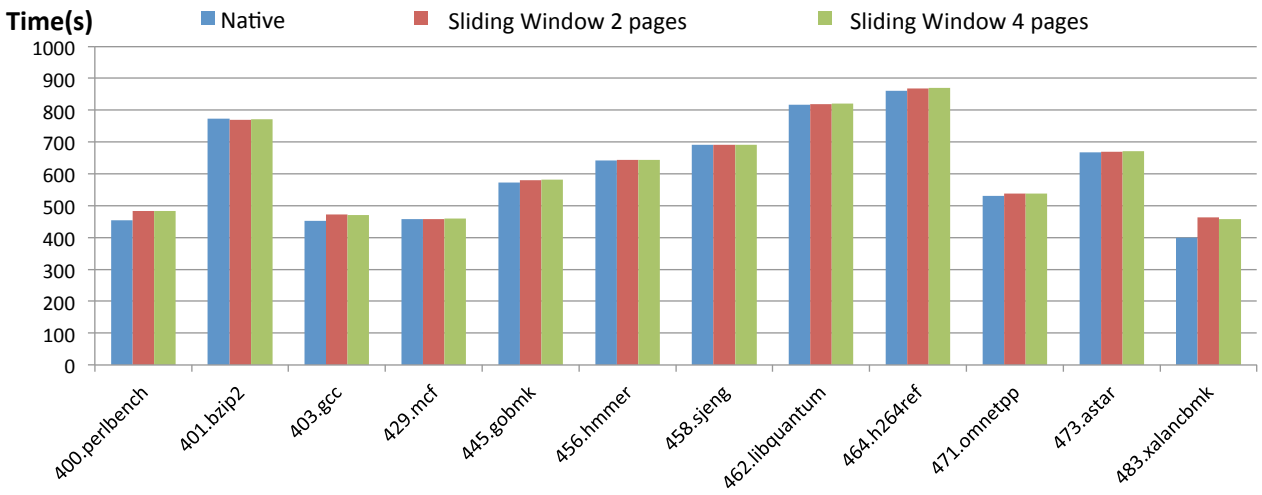

Fig. 5. **The SPEC INT2006 Benchmark Results.**

SPEC CPU Benchmark. We choose the benchmark tool SPEC CPU2006 benchmark suite to evaluate the computation performance. Specifically, we run the testing suits with and without ROPecker. The results are illustrated in Figure 5, which shows that ROPecker only introduces 2.60% performance loss on average.

Disk I/O Performance Evaluation. We choose the benchmark tool Bonnie++ (version 1.96) to evaluate disk I/O performance. The tool sequentially reads/writes data from/to a particular file in different ways. The read/write granularity varies from a character to a block (i.e., 8192 Bytes). Furthermore, we also test the time cost of the random seeking. Figure 6 shows the disk I/O measurement results which show the performance overhead on the disk I/O is quite low, i.e., 1.56% overhead on average.

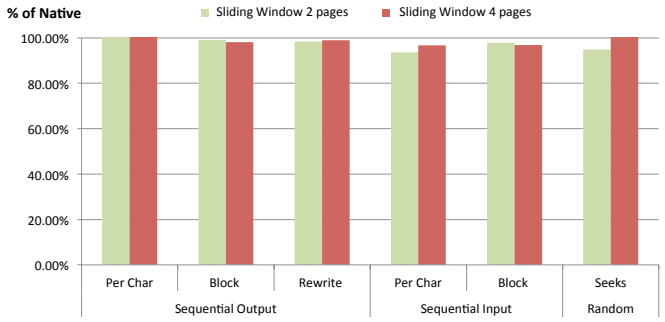

Fig. 6. **The disk performance results.**

Network I/O Performance Evaluation. We choose the Apache web server httpd-2.4.6 to evaluate the network I/O. In our experiments, two machines are directly connected through a network cable. The tool *ab* on the client machine sends network requests (e.g., *ab -n 100000 -c 50 http://192.168.1.11/index.html*) to retrieve a web page on the server running with ROPecker. The Apache web server is configured to work in mpm-worker mode. It has one worker process with 40 threads. Figure 7 shows the experiment results of multiple test scenarios: "HTTP" and "HTTP-4k" represent that the server running HTTP protocol serves the default work page (45 bytes) and a 4K bytes web page, respectively. "HTTP-S" and "HTTPS-4k" denote that the server running HTTPS protocol with a 1024 bit RSA key. The test results indicate that a larger sliding window achieves better performance, and the performance overhead on network I/O is acceptable. For example, the performance overhead is 6.33% when the Apache server delivers the default 45-byte HTTP web pages, and is only 0.08% when serving 4K byte HTTP web page. Even in more computation-bound cases, e.g., HTTPS, the ROPecker overhead is only 9.72%, when choosing the 4-page sliding window.

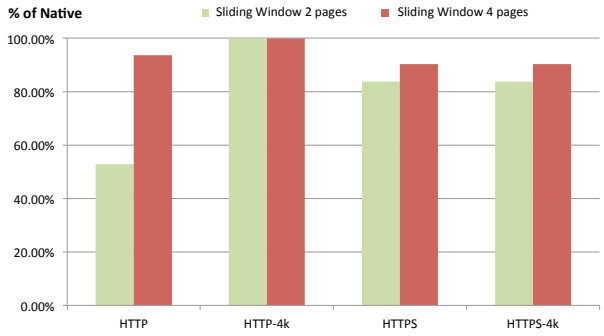

Fig. 7. **The network performance results.**

IX. Discussions

A. Stack Pivoting Attack

The adversary can launch the *stack pivoting* attack [29] to evade ROPecker detection. In this attack, the adversary puts the payload into another memory region (e.g., a heap buffer or the global data region), and manipulates the stack pointer to point to that region. Fortunately, the future payload checking algorithm is able to defeat such attacks, since ROPecker always follows the execution flow to identify the potential gadget chain, without relying on the trustworthiness of the stack information.

B. Gadget Gluing Attack

Our payload detection algorithm is designed based on the assumption that a gadget does not contain direct branch instructions, which is also used in the many previous work [7]–[9], [12], [14], [17]. Therefore, the gadget chain detection stops when a direct branch instruction is encountered. However, the ROP attack in theory can foil our detection by constructing a special gadget which consists of two short code sequences glued together by a direct branch instruction. We call this type of attack as *Gadget Gluing Attack*, which has not been found in real-life to the best of our knowledge.

Gadget gluing attacks are very powerful, as they lead to high ambiguousness between ROP attacks and normal executions. The current version of ROPecker can not well defend against them. To mitigate such attacks, in the future, one possible extension of ROPecker is to follow the direct branch at run-time. If its destination is a gadget, ROPecker treats the whole as a glued gadget. The following step can be repeated one, two or more times (according to the configuration) until getting a gadget or reaching the times limit. This approach may introduce more false positives and/or false negatives.

C. Short Gadget Chain

In certain extreme cases, the gadget chain may only contain one or two gadgets. For such special ROP attacks, our scheme can not detect or prevent them because the length of the gadget chain does not exceed the threshold. However, due to the limited power of the adversary in such attacks, in fact, their goals usually are not directly launching malicious behaviors, while they aim to open doors for facilitating the following ROP attacks. In that way, our scheme is likely to detect the following ROP attacks.

D. Gadgets Within Sliding Windows

In most cases, the gadgets within the sliding windows are not enough for an ROP attack. However, in the whole life-cycle of a process, we can not eliminate the possibility for the adversary to find one sliding window where an ROP attack is possible. For such cases, the current version of ROPecker cannot detect them. One possible solution is to dynamically reduce the window size to lower the possibility of ROP attacks. Specifically, we can observe all the sliding windows that ever occurred during normal executions, analyze each of them to evaluate the possibility of launching an ROP attack, and record the highly possible ones. According to those recorded sliding windows, ROPecker could choose a dynamic way to temporally reduce the window size when the execution flow is running in one of them, and immediately restore the size when the execution flow moves out.

X. Related Work

A. Randomization

Address Space Layout Randomization (ASLR) is proposed to prevent ROP attack by randomizing base addresses of code segments. Given that the adversary has to know the locations of the gadgets to launch an ROP attack, the ASLR technique seems to effectively prevent ROP attacks. However, it has been shown that ASLR can be bypassed by leveraging brute-force attacks [30] or information leakage attacks [31]. In addition, some libraries or applications may not be ASLR-compatible, which allows the adversary to find useful gadgets to circumvent ASLR mechanism.

Facing such difficulties, researchers proposed the *binary stirring* technique [8], which imbues x86 native code with the ability to self-randomize its instruction addresses each time it is launched. Note that the size of the modified binary file increases on average by 73%. The ILR [7] technique is proposed to randomize the location of every instruction in a program for thwarting an attacker's ability to re-use program functionality. The new generated ILR-programs have to be executed on a dedicated Virtual Machine (VM). The run-time performance overhead is relatively low, but the rule files are quite large and their in-memory size are even worse, e.g., the on-disk size of the rule file for the benchmark tool 481.wrf is about 264MB, while its in-memory size reaches to $345MB$. Note that the *whole* database in our system is only $210MB$..

B. Compiler-Based Approaches

The control flow integrity [19] is a typical technique to defend against the code reuse attack. However, the traditional control flow works in function level, which leads to the failure of the detection of the ROP attack since its disorder of control flow is on instruction level. CFLocking [5] aims to limit/lock the number of abnormal control flow transfer by recompiling a program. Essentially this technique can not handle the ROP attacks that use unaligned gadgets.

The return-less kernel [4] is a compiler-based approach that aims to remove the *ret* opcode from the kernel image by placing the control data into a dedicated buffer instead of the stack. Obviously this technique only defends against ret-based ROP attack. Following the same idea, Shuo et al. [15] propose a virtualization based approach, which requires the source code and compiler to insert control-data-integrity-checking code into function prologue and epilogue. Again, this approach can not defend against the ROP attack that use jump/call instructions.

G-free [6] is a compiler-based approach, which aims to 1) eliminate all unaligned indirect branch instructions with *aligned sled* [6] and 2) protect the aligned indirect branch instructions to prevent them from being misused. G-free requires side information, i.e., the source code. Moreover, the inserted code may introduce new gadgets, which can be potentially used by the adversary.

C. Instrumentation-Based Approaches

TRUSS [32], ROPDefender [2], DROP [1] and TaintCheck [33] use code instrumentation technique to insert checking code into binary code to detect ROP attack. Such approaches not only break the binary integrity, but also suffer high performance overhead. For instance, the preliminary performance measurements for DROP range from 1.9X to 21X, and the performance overhead for TaintCheck is over 20X. In addition, some of them, such as ROPDefender and DROP only focus on ret-based ROP, rather than handling all types of ROP attack.

To overcome the high performance overhead issue, several new approaches are proposed. Specifically, the IPR [9]

technique is proposed, which aims to smash the gadgets in place without changing the code size. However, many gadgets can not be removed using the in-place technique. In addition, the in-place smashing technique may not always smash a significant part of the executable address space [9], and it is hard to give a definitive answer on whether the remaining un-modifiable gadgets would be sufficient for constructing a meaningful ROP attack. The ROPGuard [3] and KBouncer [11] approaches only add the checking points in the *selective* critical functions (e.g., Windows APIs) by instrumenting the binary code on-the-fly. Although the infrequent invocations of the checking algorithm lead to low performance overhead, they inevitably miss the ROP attacks that do not use those paths. Note that the ROPGuard only works the non-JOP code and KBouncer completely relies on non-fully reliable LBR records, which could be overflowed or polluted during context switches. CCFIR [10] randomly inserts all legal targets of indirect control-transfer instructions into a dedicated *Springboard board*, and then instruments the binaries to limit indirect transfers to flow only to the board. CCFIR suffers from compatibility issues due to the integrity protection of the system shared libraries on certain platforms (e.g., Windows 7). In addition, the compatibility issue may lead to the failure of the ROP defence since the adversary may only use the gadgets from such non-instrumented system libraries.

D. Others

CFIMon [34] is to detect a variety of attacks violating control flow integrity by collecting and analyzing run-time traces on-the-fly. However it has high detection latency that may cause it too late to detect an attack. MoCFI [35] is a framework to mitigate control flow attacks on smart phones. It performs control flow integrity checking on-the-fly without requiring the applications source code, and the experiment results show that it does not induce notable overhead when applied to popular iOS applications. Polychronakis et al. propose a method [36] to identify ROP payloads in arbitrary data. The technique speculatively drives the execution of code that already exists in the address space of a targeted process according to the scanned input data, and identifies the execution of valid ROP code at run-time. The basic idea of the payload detection method has been adopted by our payload checking algorithm.

XI. Conclusions and Future Work

We have presented ROPecker, a novel and universal ROP attack detection and prevention system. We have innovated two techniques as the main building blocks of ROPecker. One is the gadget chain detection algorithm which detects the chain in the past and future execution flow, and the other is the sliding window mechanism which allows the detection algorithm to be triggered with a proper timing so as to achieve high accuracy and efficiency. ROPecker does not require source code, special compiler, or binary instrumentations. In fact, our scheme is complementary to instrumentation- and compiler-based approaches, as well as randomization schemes. Our scheme provides another line to defend against ROP attacks. We evaluated the security of ROPecker by running experiments with real-life ROP attacks and those generated from Q. We also evaluated its performance by running benchmark tools which show acceptable performance loss.

Future Work. The current design and implementation of ROPecker focus on user-space ROP attacks. In the future work, we will extend ROPecker to defend against kernel-space ROP attacks, because the hallmarks of ROP attacks are still there. However, ROPecker should not be placed in the kernel space any more, since it may be disabled by the adversary once the kernel is compromised. An promising way is to move ROPecker into the hypervisor space, where the adversary cannot access. By doing so, the performance penalty will be higher because the host-guest switch cost is higher than the one of kernel-user context switch. The performance loss may also be introduced by interrupts and exceptions, because they may introduce poor temporal and spatial locality for the kernel execution flow, and thereby increase the frequency of the detection algorithm invocation.

In the future work, we also aim to extend ROPecker to support ARM platform. The ARM-based CPUs might not have LBR registers or the registers with similar functionality. Thus, the past payload detection will be removed in such environment. Fortunately, the ROPecker future payload detection still works, unlike other schemes [11] that completely rely on LBR. On ARM platforms, instructions are aligned to 16 bits or 32 bits. This will simplify the disassembler and the gadget analysis of the ROPecker pre-processor. The minor change is that the optional alignment checking does not work on ARM-based platform.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd Davide Balzarotti for their numerous, insightful comments that greatly helped improve the presentation of this paper. The authors are also grateful to Virgil D. Gligor and Edward J. Schwartz for useful discussions. This research is supported by the Singapore National Research Foundation under its International Research Centre Singapore Funding Initiative and administered by the IDM Programme Office, Media Development Authority (MDA).

References

[1] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting Return-Oriented Programming Malicious Code," in *Proceedings of the 5th International Conference on Information Systems Security*, 2009.

[2] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[3] F. Ivan, "Runtime prevention of return-oriented programming attacks," https://code.google.com/p/ropguard/.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating Return-oriented Rootkits with "Return-Less" Kernels," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[5] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating Code-reuse Attacks with Control-flow Locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[6] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[7] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'D My Gadgets Go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[8] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-oriented Programming Using In-place Code Randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[10] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[11] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[12] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[13] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[15] T. Shuo, H. Yeping, and D. Baozeng, "Prevent Kernel Return-oriented Programming Attacks Using Hardware Virtualization," in *Proceedings of the 8th International Conference on Information Security Practice and Experience*, 2012.

[16] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Conference on Security*, 2011.

[18] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, 2011.

[19] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

[20] B. Daniel P. and C. Marco, "Understanding the Linux Kernel, Third Edition, chapter 2 - hardware cache and chapter 9 - demand paging," 2005.

[21] G. Dabah, "A lightweight, Easy-to-Use and Fast Disassembler/Decomposer Library for x86/AMD64," http://ragestorm.net/distorm/.

[22] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[24] "Standard Performance Evaluation Corporation, SPEC CPU2006 Benchmarks," http://www.spec.org/osg/cpu2006.

[25] C. Russell, "Disk Performance Benchmark Tool - Bonnie," http://www.coker.com.au/bonnie++/.

[26] "The Apache Software Foundation, Apache HTTP Server," http://httpd.apache.org/.

[27] Longld, "Payload Already Inside: Data Resue For ROP Exploits," https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf, 2010.

[28] "HT Editor 2.0.20 Buffer Overflow (ROP PoC)," http://www.exploit-db.com/exploits/22683/.

[29] 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," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[31] A. Sotirov and M. Dowd, "Bypassing Browser Memory Protections," Blackhat USA 2008.

[32] S. Saravanan, Z. Qin, and W. Weng-Fai, "Transparent Runtime Shadow Stack: Protection Against Malicious Return Address Modifications," 2008.

[33] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[34] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[35] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones," in *19th Annual Network & Distributed System Security Symposium*, 2012.

[36] M. Polychronakis and A. D. Keromytis, "ROP Payload Detection Using Speculative Code Execution," in *Proceedings of the 6th International Conference on Malicious and Unwanted Software*, 2011.