ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications

Lu Si(✉), Jie Yu, Lei Luo, Jun Ma, Qingbo Wu, and Shasha Li

College of Computer, National University of Defense Technology, Changsha 410073, China
{lusi,jackyu,luolei,majun,wuqingbo}@ubuntukylin.com, shashali@nudt.edu.cn

Abstract. Return-oriented Programming (ROP) is a new exploitation technique that can perform arbitrary unintended operations by constructing a gadget chain reusing existing small code sequences. Although many defense mechanisms have been proposed, some new variants of ROP attack can easily circumvent them.

In this paper, we present a new tool, ROP-Hunt, that can defend against ROP attacks based on the differences between normal program and ROP malicious code. ROP-Hunt leverages instrumentation technique and detects ROP attack at runtime. In our experiment, ROP-Hunt can detect all types of ROP attack from real-world examples. We use several unmodified SPEC2006 benchmarks to test the performance and the result shows that it has a zero false positive rate and an acceptable overhead.

Keywords: Return-oriented Programming · Buffer overflow · Detection · Code reuse attack · Binary instrumentation

1 Introduction

Since the widespread adoption of data execution prevention (DEP) [1], which ensures that all writable pages in memory are non-executable, it's hard for attackers to redirect the hijacked control flow to their own injected malicious code. To bypass DEP mechanism, code reuse attack (CRA) techniques are proposed and have become attacker's powerful tools. Instead of injection code, they reuse instructions already residing in the attacked vulnerable process to induce malicious behaviors. Return-into-libc technique [37] is one simple practice of it, in which the attacker uses a buffer overflow to overwrite the return address stored in the stack to the address of the library function chosen to be executed. Traditional return-into-libc attack leverages libc functions and cannot support arbitrary computation on the victim machine.

Return-oriented Programming (ROP) is another code reuse attack technique, which executes short instruction sequences called gadgets instead of an entire function. ROP was first demonstrated by Shacham [35] for the ×86 platform,

G. Wang et al. (Eds.): SpaCCS 2016, LNCS 10066, pp. 131–144, 2016.
DOI: 10.1007/978-3-319-49148-6_12

and was subsequently extended to other architectures [13,16,23,26]. It has been proved that ROP can perform Turing-complete computation [36]. Some tools have been developed that allow attackers to construct arbitrary malicious programs using ROP automatically [22,24,33,34].

In the last few years, a number of software and hardware defenses have been proposed to mitigate ROP-based attacks. For example, DROP [17] and DynIMA [20] will trigger an alarm if the small instruction sequences each ending with a *ret* instruction are executed consecutively. ROPdefender [21] maintains a shadow stack and verifies all return addresses. Li et al. [27] proposed a compiler for the ×86 platform that avoids issuing "$0 \times c3$" bytes that can be used as unintended return instructions. Further more, it replaces intended call and return instructions with an indirect call mechanism. However, these mechanisms only focus on the ROP gadgets ending with return instructions and can not defeat other types of ROP-like attacks that capture gadgets without return instructions. CFLocking [11] and G-Free [30] aim to defend against all types of ROP attacks, but they require the source code which is often unavailable to the end users in the real world. KBouncer [32] covers all ROP attack types, requires no side information and achieves good runtime efficiency. However, it 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.

ROP attack chains gadgets together to perform complex computations and has its own features: the length of gadget is short, contiguous gadgets are not in the same routine and they all execute system calls in somewhere. Based on these features, we design and implement a tool named ROP-Hunt, which dynamically detects all types of ROP attack by checking whether the execution behavior has these matched features. In ROP-Hunt, based on the hazard degree, we divide the ROP report into two categories: *Warning* and *Attack*.

In summary, the main contributions of our work are:

- We statistically analyze a number of normal applications and latest ROP malicious code, and extract features of the ROP attack.
- We propose a novel approach to protecting legacy applications from all types of ROP attacks without accessing to source code.
- We design and implement a prototype, ROP-Hunt, on ×86-based Linux platform and evaluate its security effectiveness and performance overhead.

The remainder of the paper is organized as follows: In Sects. 2 and 3, we describe the ROP attacks and analyze the features of them. The design and implementation of ROP-Hunt are illustrated at Sect. 4. Sections 5 and 6 discuss the parameter selections and delay gadget respectively. Section 7 presents the security and performance evaluation of ROP-Hunt. Section 8 examines its limitations. Finally, we conclude this paper and discuss the future work in Sect. 9.

2 ROP Attack

Without injecting new code into the programs address space, ROP attacks consist of short instruction sequences, which are called gadgets. Each gadget

performs some small computation, such as adding two registers or loading a value to memory, and ends with return instruction. We can chain gadgets together and transfer the control flow from one gadget to another by writing appropriate values over the stack.

Figure 1 illustrates a general ROP attack workflow. In step 1, the attacker exploits a memory-related vulnerability of a specific program, e.g., a buffer overflow, and moves the stack pointer (ESP) to the first return address. For example, Aleph in [31] uses stack smashing techniques to overwrite the return address of a function. Return address 1 is injected at the place where the original return address was located, and the value of ESP will be automatically changed to this point. In step 2, execution is redirected to the first gadget by popping return address 1 from the stack. The gadget is terminated by another return instruction which pops return address 2 from the stack (step 3) and redirects execution to the next gadget (step 4). Each gadget is executed one by one in this way until the attacker attains his goal.

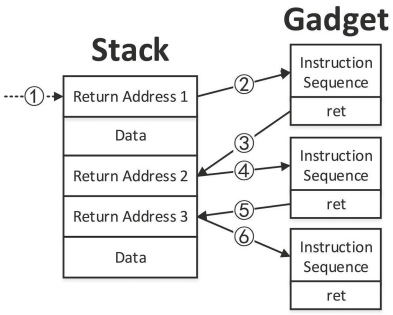
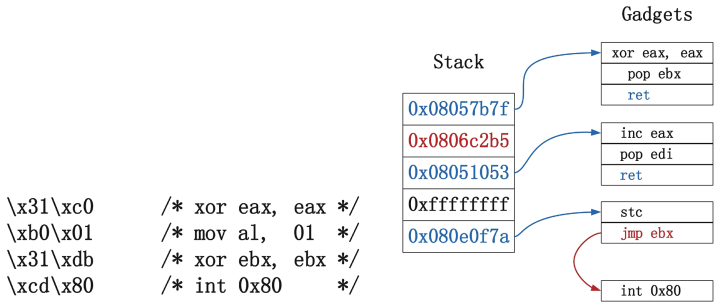


Fig. 1. A general ROP attack

Recently, some new variants of ROP attack without using *ret* instructions were proposed. Checkoway et al. [15] found it is possible to perform return-oriented programming by looking for a pop instruction followed by an indirect jump (e.g., *pop edx*; *jmp* [*edx*]). This instruction sequence behaves like returns, and can be used to chain useful gadgets together.

Jump-Oriented Programming (JOP) [12] is another variant of ROP attack which uses register-indirect jumps instead of returns. JOP uses a dispatcher table to hold gadget addresses. Each gadget must be followed by a dispatcher, which is an instruction sequence that can govern the control flow. The dispatcher is used as a virtual program counter and translates the control flow to an entry in the dispatch table, which is the address of a particular jump-oriented functional gadget. At the end of a functional gadget, the attacker uses an indirect jump back to the dispatcher. Then, the dispatcher advances the pointer to the next functional gadget. A simple case of dispatcher is *add edx,* 4; *jmp* [*edx*].

Call Oriented Programming (COP) [14] was introduced by Nicholas Carlini and David Wagner in 2014. Instead of using gadgets that end in returns, the



(a) Traditional Shellcode (b) ROP Malicious Code

Fig. 2. A simple mixed ROP attack

attacker uses gadgets that end with indirect calls. COP attack does not require a dispatcher gadget and gadgets are chained together by pointing the memory-indirect locations to the next gadget in sequence.

To evade current protection mechanisms, attackers prefer to use combinational gadgets. Figure 2 shows a very simple mixed ROP attack constructed by only 4 short gadgets. It is derived from a traditional shellcode [3] which exits the running process on ×86 architecture. We used *exit(n)* (*n* represents a non-zero integer) system call instead of *exit(0)* for convenience. The system call number is stored in *eax* and the parameter is stored in *ebx*. DROP [17] and DynIMA [20] only detect contiguous ret-based gadgets and the attacker can leverage this simple ROP malicious code to evade these two defense mechanisms.

3 Features of ROP Attack

The key to ROP attack detection is finding the differences between ROP malicious code and normal programs. One of the important factors in ROP is the gadget length. [20] found that instruction sequences used in ROP attacks range from two to five instructions. DROP [17] found that the number of the instructions in the gadget is no more than 5. Kayaalp et al. [25] extracted gadgets from standard C library and conducted studies on average gadget lengths. The result showed that as the gadget length grew the number of side effects grew linearly making them increasingly more difficult to use.

There are also some other factors being considered in present detecting mechanisms. DynIMA [20] reports a ROP attack if three of small instruction sequences were executed one after another. Fan Yao et al. [38] found that it is relatively hard to find gadgets within short distances.

Based on the experience of writing ROP malicious code, we find out other two features. First, contiguous gadgets, no matter ending with jump or call instructions, do not locate in the same routine. Second, shellcodes always leverage system call to transfer the flow of control to the kernel mode.

In computer programming, routine is a sequence of code that is intended to be called and used repeatedly during the execution of a program. In high-level languages, many commonly-needed routines are packaged as functions. In the traditional ROP attacks, each gadget ends with the return instruction. At most time, they are not in the same routine except recursive returns. We extract gadgets from glibc by ROPGadget [8], which is an open source tool to search gadgets, and construct some JOP malicious code with the algorithm proposed by [12]. We find that it is extremely hard to use contiguous gadgets that are in the same routine.

ROP malicious code is the derivation of shellcode and bases on the traditional shellcode to construct gadgets. We analyze all 247 shellcodes from [5] and find that 212 of them invoke system call at least once. However, to evade the IDs detecting mechanisms, other shellcodes encrypt or self-modify payloads and do not use "*int* 0×80" directly to avoid containing sensitive data (e.g., cd 80). But anyway they will invoke system call at runtime to get higher privilege. [2] invokes *_kernel_vsyscall* function that uses *sysenter* instruction to transfer the control flow from user mode running at privilege level 3 to operating system. However, *sysenter* instruction provides a fast entry to the kernel and also can be considered as another kind of system call.

We consider the (i) small gadget size, (ii) the execution of system call and (iii) contiguous candidate gadgets are not in the same routine as the ROP attack's most representative characteristics. Based on these three differences between ROP malicious code and normal programs, we develop a tool named ROP-Hunt, which dynamically detects ROP attack by checking whether the execution trace deviates from the normal execution route. We will show the design of ROP-Hunt in the next section.

4 ROP-Hunt Design and Implementation

Based on the features of ROP attack, we propose our approach to efficiently detect ROP attacks. Since we assume no access to source code, we make use of instrumentation technique that allows to add extra code to a program to observe and debug the program's behavior [29].

4.1 Assumptions and Definitions

In this paper, we define the number of instructions in a gadget as *G_size*. Candidate gadget refers to the gadget that *G_size* is greater than the threshold *T0*. The length of contiguous candidate gadget sequence is defined as *S_length*, and *Max(S_length)* represents the maximum values of *S_length*.

In order to simulate the real environment, we make the following assumptions:

1. We assume that the underlying system supports DEP [1] model that prohibits writing to executable memory. In this case, code injection based attacks are impossible. Modern processors and operating systems already enable DEP by default.

2. We assume that the attacker is able to perform a buffer overflow [19,31,39], a string formatting attack or a non-local jump buffer (using *setjmp* and *longjmp* [4]) to mount a ROP attack.
3. We assume that the attacker operates in the user mode and the vulnerability exploited to initiate the attack does not lead to a privilege escalation.
4. We assume that we have no access to source code.

4.2 System Overview

Figure 3 shows the flow chart of ROP-Hunt. According to the features of ROP that we have analyzed in Sect. 3, ROP-Hunt monitors the program dynamically, intercepts the system call instruction and three control flow sensitive instructions: *call*, *jmp* and *ret*. There are two categories of ROP report: *Warning* and *Attack*. *Warning* indicates that there is a serious risk that the process is under a ROP attack. Since it have not invoked a system call to visit the underlying system sources, we believe that it is not ready to do any meaningful attack. If the statistic values break the thresholds and a system call is being invoked, ROP-Hunt will kill the process appending with an *Attack* report.

– *Report Warning:* When ROP-Hunt recognizes these three instructions (call, indirect jump and return), it checks whether the length of instruction sequence is greater than *T0*. If not, it extracts the target address and the current instruction address. Especially for *ret* instruction, the target address will be popped from the stack. Then ROP-Hunt checks whether the two addresses locate in the same routine. If not, we record the instruction sequence as a candidate gadget. Next, we count the maximum length of contiguous candidate gadgets *S_length*. If *S_length* is less than or equal to *T1*, we will set the potential attack flag to *True* and raise a *Warning*.
– *Report Attack:* System call is the only way to transfer the flow of control from user space to kernel space. When a system call instruction is recognized, ROP-Hunt checks whether the potential attack flag is *True*. If the condition is satisfied, ROP-Hunt will report an *Attack* and terminate the process.

4.3 Implementation Details

To demonstrate the effectiveness and evaluate the performance of our approach, we have developed a prototype implementation for the ×86 32-bit version of Ubuntu 14.04 with kernel 3.19. For our prototype, ROP-Hunt, we used the binary instrumentation framework Pin [28] (version 2.14).

We incorporated ROP-Hunt directly into the Pin Framework. Pin is a tool for the instrumentation of programs and instruments all instructions that are actually executed. There are two kinds of working mode in Pin, probe mode and just-in-time (JIT) mode. In JIT mode, Pin can intercept each instruction before it is executed by the processor, even if the instruction was not intended by the programmer.

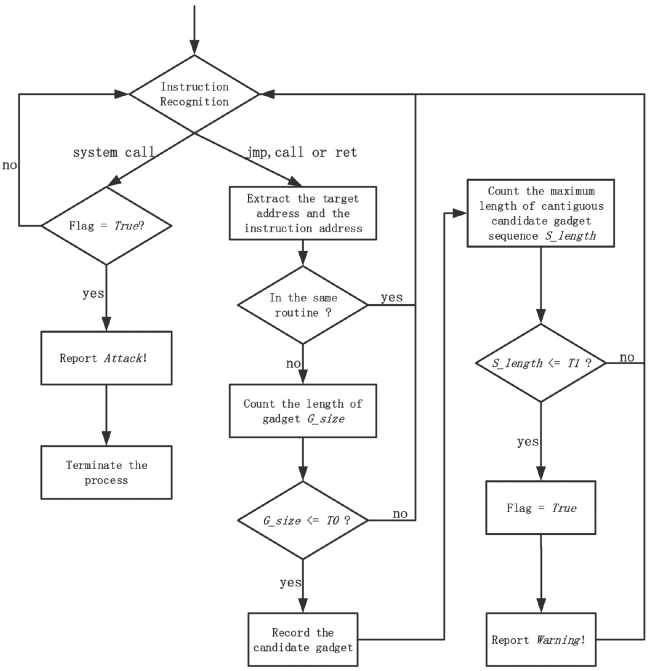


Fig. 3. Work flow of ROP-Hunt

To instrument a binary at runtime, we have to determine where code is inserted and what code to execute at insertion points. Pin provides instrumentation tools which are called Pintools. Pintools are written in the C/C++ programming language using Pin's rich API and allow to specify your own instrumentation code. We designed and implemented our own Pintool to detect ROP attacks in the Pin framework.

The overall architecture of the runtime system is depicted in Fig. 4. Our architecture consists of the Pin Framework and the Pintool ROP-Hunt. Pin is the engine that jits and instruments the program binary. Pin itself consists of a virtual machine (VM), a code cache, and instrumentation APIs invoked by Pintools. The VM consists of a JIT compiler, an emulator and a dispatcher. When a program is started, the JIT compiles and instruments instructions, which are then launched by the dispatcher. The compiled instructions are stored in the code cache in order to reduce performance overhead if code pieces are invoked multiple times. The emulator interprets instructions that cannot be executed directly.

Our Pintool, ROP-Hunt, consists of a record unit and a detection unit which contains instrumentation routines and analysis routines. The detection unit leverages instrumentation APIs to communicate with Pin and the record unit just stores the statistic values at runtime.

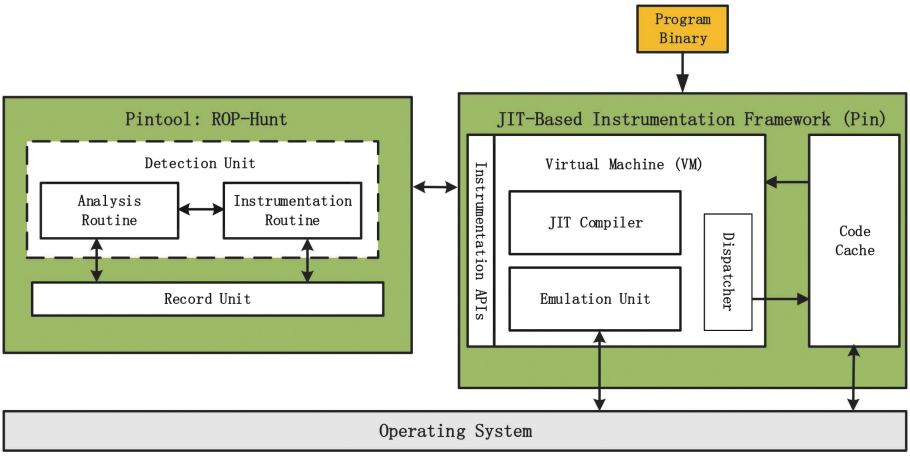


Fig. 4. Implementation of ROP-Hunt within pin framework

4.4 Instrumentation and Analysis Routines

As mentioned in Sect. 4.1, one of the key points is to recognize the instruction types. The instrumentation routines of our ROP-Hunt use the inspection functions *INS_IsSyscall(INS ins)*, *INS_IsSysenter(INS ins)* provided by the Pin APIs to determine whether the current instruction is a system call or a system enter, and use *INS_IsIndirectBranchOrCall(INS ins)* to determine whether the current instruction is a branch instruction. If the current instruction is an indirect jump, call or return instruction, then we invoke an analysis function that extracts the addresses of both the current instruction and the target.

POP-Hunt assigns each routine an ID. The ID is globally unique, i.e., an ID will not appear in two images. If the same routine name exists in two different images (i.e., they are in different addresses), each will have a different ID. If an image is unloaded and then reloaded, the routines within it will most likely have different IDs than before. ROP-Hunt leverages the function *PIN_InitSymbols()* to initialize the symbol table and read symbols from the binary. Since then, we can get the routine ID by the address.

The record unit allocates data space to each thread respectively. We use the thread local storage (TLS) from the Pin APIs to avoid that one thread accesses the record of another thread.

5 Parameter Selections

We have to determine the thresholds of the two factors which represent the features of ROP: the number of instructions in the gadget (*G_size*), the length of contiguous candidate gadget sequence (*S_length*).

The gadget size threshold (*T0*) affects the detection accuracy. Bigger threshold generally incurs higher false positive. To find *T0*, we used two well-know

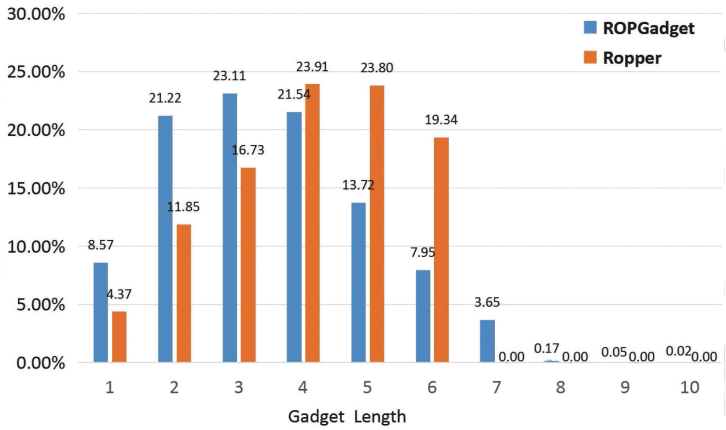


Fig. 5. The size of gadget measurement results

gadget search tools, ROPGadget [8] and Ropper [9], to measure the sizes of gadgets from many normal applications, which include 22 popular Linux tools (e.g., ls, grep, and find) under directory /bin and /usr/bin, and 3 large binaries (Apache web server httpd 2.4.20, mysql 5.6 and python 2.7). We collected 282341 gadgets totally, 125605 from ROPGadget and 156736 from Ropper. As shown in Fig. 5, the largest gadget size is 10 and nearly all gadgets size is less than 8. In the gadget set generated by Ropper, the largest size is 6. We also measured the ROP malicious code collected from the real world ROP attacks, and no gadget size is greater than 6. Based on the above results, we can safely choose 7 as the gadget size threshold (*T0*). If the length of an instruction sequence is not greater than 7, it will be treated as a candidate gadget by ROP-Hunt.

In ROP attack, the attacker chains a few gadgets together to complete an intended operation. To construct a system call operation, the attacker has to use at least 3 gadgets to place the correct parameters in the argument registers and jump to the system call entry. We believe an attacker can not do any meaningful attacks by just using 3 or less gadgets. So we set the *T1* to 3, that is to say, ROP-Hunt checks whether there are more than 3 contiguous gadgets.

6 Delay Gadget

ROP-Hunt is effective under the assumption that usable gadgets are short allowing us to distinguish attacks from normal programs. However, smart attackers may be able to tolerate some of the side-effects in a long gadget and use it in the middle of the attack to evade the detection. Mehmet Kyaalp et al. [25] introduced delay gadget that was long enough to reset the gadget counter used by the signature detector. They made a call to a function that resulted in executing a larger number of instructions. By convention, when a function returns, many

registers such as *ebx*, *esi*, *edi*, *esp*, and *ebp* are saved. That is to say, delay gadget can reduce side-effect greatly.

The purpose of a delay gadget is to avoid detection by signature-based detectors. It neither executes any part of the attack code or corrupts the machine state needed by the attack. It is impossible to conduct ROP attack only by delay gadgets. So when the previous gadget ending with a call invokes a function and the gadgets is longer than the threshold *T0*, ROP-Hunt just ignores this gadget and does not reset the counter. But if the gadget size does not break the threshold *T0*, the counter is still added by one.

7 Evaluation

In this section, we evaluated the security effectiveness and the performance overhead of ROP-Hunt. All experiments were performed on a computer with the following specifications: Intel Core i3 2370M CPU, 4 GB RAM, 32-bit Ubuntu with kernel version 3.19. For the security evaluation, we verify our approach with two real-world ROP attacks and a small program that has a simple stack buffer overflow triggered by a long input parameter. For the performance evaluation, we used 18 C and C++ SPEC CPU2006 [10] benchmarks for our experiments. The benchmarks were compiled using gcc-4.8.3 compiler.

7.1 Security Evaluation

In the first test, we evaluated the effectiveness of ROP-Hunt using two realistic programs: Hex-editer (2.0.20) and PHP (5.3.6). These two templates of ROP malicious code are available on the websites [6,7]. In the vulnerability exploitation of PHP, we inputted a long path name for a UNIX socket to trigger the buffer overflow and then transferred the control flow to the ROP payload. The ROP payload had 31 contiguous gadgets and the largest gadget contained 7 instructions (did not break the threshold *T0*). Therefore, ROP-Hunt raised a *Warning* and set the potential attack flag to *True*. The last gadget in the contiguous gadgets sequence invoked a system call to execute /bin/sh immediately. Hence, ROP-Hunt reported an *Attack* and terminated the process.

To further assess ROP-Hunt detection capabilities, we used a simple target program that had a *strcpy* vulnerability (demonstrated in [31]). The program were compiled by gcc-4.8.4 and linked with glibc-2.3.5. We used ROPGadget [8] to analyze the program and generate usable gadgets. We manually chained candidate gadgets together to rewrite 30 representative shellcode from the Shell-Storm Linux shellcode repository [5]. These shellcodes were composed of combinational gadgets which ending with *ret*, *jmp* or *call* instructions. Gadgets longer than 7 instructions were extremely difficult to incorporate due to side effects. The most simple attack required 4 gadgets (greater than *T1*). As we have analyzed in Sect. 3, all shellcodes used system calls to complete attacks. The experimental result showed that ROP-Hunt could detect all these ROP attacks without false positive.

7.2 Performance Overhead

We chose the benchmark tool SPEC CPU2006 benchmark suite [10] to measure the performance of ROP-Hunt. Specifically, we ran the testing suits with and without ROP-Hunt. The results are illustrated in Fig. 6, which shows that applications under protection of ROP-Hunt run on average 1.75×. The slowdown for benchmarks ranges from 1.05× to 2.41×. We compared ROP-Hunt with other ROP detectors based on instrumentation technique. According to the results in [17,21], applications running under ROPdefender and DROP are 2.17× and 5.3×. [18] causes an average slowdown of 3.5×.

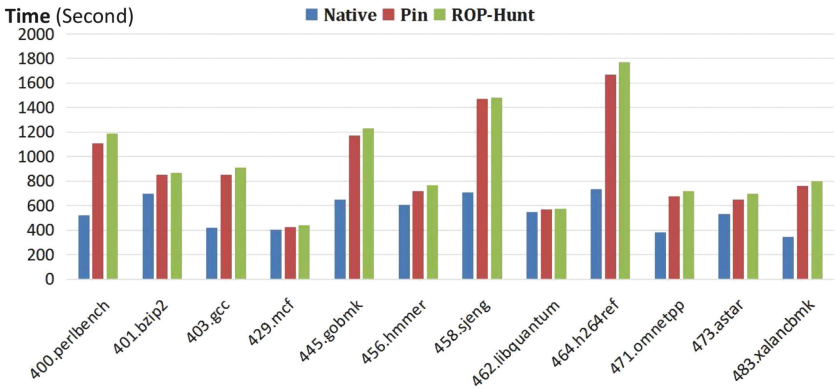


Fig. 6. SPEC CPU2006 benchmark results

The result shows that the Pin framework itself also induces an average slowdown of 1.66×. We believe the performance of ROP-Hunt will be continuously improved with the optimization of Pin Framework.

8 Discussion

We design and implement ROP-Hunt to detect ROP attacks at runtime, and currently ROP-Hunt is based on dynamic binary instrumentation tool Pin. Although ROP-Hunt is effective detecting ROP attacks, there are some limitations. First, ROP-Hunt only detects ROP malicious code on ×86 architecture. However, malicious code can be rewritten on other architectures by ROP technique. We believe that our approach can be deployed to other architectures. Second, ROP-Hunt detects ROP attack with the assumption that all ROP malicious codes meet the thresholds discussed in Sect. 5. Although it is extremely hard, there is a theoretical possibility that some ROP attacks may break this assumption. Finally, ROP-Hunt is implemented by using the jit-based binary instrumentation framework Pin and causes an average slowdown of 1.75×. The performance overhead may be unacceptable for some time-critical applications.

9 Conclusions

ROP is a very powerful exploitation technique used to bypass current security mechanisms. In this paper, we studied and extracted the features of the ROP malicious code. Based on the identification of distinctive attributes of ROP malicious code that are inherently exhibited during execution, we proposed a novel and practical approach for protecting against ROP attack without requiring access to source code. The experimental results showed that our prototype, ROP-Hunt, successfully detects all ROP attacks with no false positive. ROP-Hunt leverages instrumentation technique and adds a runtime overhead of 1.75× which is comparable to similar instrumentation-based ROP detection tools. As part of our future work, we plan to port our prototype implementation to other architectures.

Acknowledgments. We thank the anonymous reviewers for their constructive comments that guided the final version of this paper. We thank National University of Defense Technology for providing essential conditions to accomplish this paper. This work is supported by the NSFC under Grant 61103015, 61303191, 61402504 and 61303190.

References

1. Data execution prevention. http://support.microsoft.com/kb/875352/EN-US
2. Linux/×86 - /bin/sh sysenter Opcode Array Payload. http://shell-storm.org/shellcode/files/shellcode-236.php
3. Linux/×86 - sys exit(0). http://shell-storm.org/shellcode/files/shellcode-623.php
4. Setjmp - set jump point for a non-local goto. http://pubs.opengroup.org/onlinepubs/009695399/functions/setjmp.html
5. Shellcodes database for study cases. http://shell-storm.org/shellcode/
6. HT Editor 2.0.20 Buffer Overflow (ROP PoC). http://www.exploit-db.com/exploits/22683/
7. PHP 5.3.6 Buffer Overflow PoC. http://www.exploit-db.com/exploits/17486
8. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/
9. ROPPER - ROP GADGET FINDER AND BINARY INFORMATION TOOL. https://scoding.de/ropper/
10. Standard Performance Evaluation Corporation, SPEC CPU2006 Benchmarks. http://www.spec.org/osg/cpu2006/
11. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 353–362. ACM (2011)
12. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 30–40. ACM (2011)
13. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 27–38. ACM (2008)

14. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 385–399 (2014)
15. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 559–572. ACM (2010)
16. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In: EVT/WOTE 2009 (2009)
17. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: detecting return-oriented programming malicious code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10772-6_13
18. Chen, P., Xing, X., Han, H., Mao, B., Xie, L.: Efficient detection of the return-oriented programming malicious code. In: Jha, S., Mathuria, A. (eds.) ICISS 2010. LNCS, vol. 6503, pp. 140–155. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17714-9_11
19. Chen, S., Li, Z., Huang, Y., Xing, J.: Sat-based technique to detect buffer overflows in c source codes. J. Tsinghua Univ. (Science and Technology), S2 (2009)
20. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, pp. 49–54. ACM (2009)
21. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: adetection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 40–51. ACM (2011)
22. Dullien, T., Kornau, T., Weinmann, R.P.: A framework for automated architecture-independent gadget search. In: WOOT (2010)
23. Francillon, A., Castelluccia, C.: Code injection attacks on Harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 15–26. ACM (2008)
24. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium, pp. 383–398 (2009)
25. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.: SCRAP: architecture for signature-based protection from code reuse attacks. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013), pp. 258–269. IEEE (2013)
26. Kornau, T.: Return oriented programming for the ARM architecture. Ph.D. thesis, Masters thesis, Ruhr-Universität Bochum (2010)
27. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with return-less kernels. In: Proceedings of the 5th European Conference on Computer Systems, pp. 195–208. ACM (2010)
28. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM Sigplan Notices, vol. 40, pp. 190–200. ACM (2005)
29. Nethercote, N.: Dynamic binary analysis and instrumentation (2004). http://valgrind.org/docs/phd2004.pdf

30. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 49–58. ACM (2010)
31. One, A.: Smashing the stack for fun and profit. Phrack Mag. **7**(49), 14–16 (1996)
32. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 2013), pp. 447–462 (2013)
33. Roemer, R.G.: Finding the bad in good code: automated return-oriented programming exploit discovery (2009)
34. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium, pp. 25–41 (2011)
35. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the ×86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561. ACM (2007)
36. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 121–141. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23644-0_7
37. Wojtczuk, R.: The advanced return-into-lib(c) exploits: PaX case study. Phrack Mag. **0x0b**(0x3a), Phile# 0x04 of 0x0e (2001)
38. Yao, F., Chen, J., Venkataramani, G.: Jop-alarm: detecting jump-oriented programming-based anomalies in applications. In: 2013 IEEE 31st International Conference on Computer Design (ICCD), pp. 467–470. IEEE (2013)
39. Zhang, M., Luo, J.: Pointer analysis algorithm in static buffer overflow analysis. Comput. Eng. **31**(18), 41–43 (2005)