RipRop: A Dynamic Detector of ROP Attacks

Mateus Tymburibá, Rubens Emilio, Fernando Pereira

Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{mateustymbu,rubens,fernando}@dcc.ufmg.br

Abstract. *Return Oriented Programming (ROP) is presently one of the most effective ways to bypass modern software protection mechanisms. Current techniques to prevent ROP based exploits usually yield false positives, stopping legitimate programs, or false negatives, in which case they fail to detect attacks. Furthermore, they tend to add substantial overhead onto the programs they protect. In this paper we introduce RipRop – a tool to detect ROP attacks. RipRop monitors the frequency of indirect branches. If this frequency exceeds a threshold, then it aborts the program under guard. RipRop is very effective: on the one hand, it exposes all the exploits publicly available that we have experimented with; on the other hand, it does not prevent the legitimate execution of any program in the entire SPEC CPU 2006 benchmark suite. To solidify our claims, we provide an argument that building attacks with low-frequency of indirect branches is difficult, and show that RipRop improves current frequency-based techniques to detect ROP attacks.*

Link to Video: `https://www.youtube.com/watch?v=EouKW1UUutU`

1. Introduction

To make software more secure, we must find ways to prevent the class of exploits known as Return-Oriented Programming (ROP) attacks. ROPs are a relatively recent technique [Shacham 2007]. Nevertheless, in spite of its short history, ROP emerges today as one of the most effective ways to exploit vulnerable software [Lin et al. 2010, Roemer and Buchanan 2012]. This effectiveness stems from the fact that the available ROP prevention techniques do either suffer from false positives, false negatives, or high overhead. This paper presents RipRop, a tool to detect ROP exploits that is substantially different from all the techniques previously described. The key insight of this tool lies on the observation that public ROP-based exploits have a very high density of indirect branches. We demonstrate empirically that such a density is unlikely to be observed in actual programs. Thus, we have designed a binary instrumentation framework that keeps track of the frequency of indirect branches, and that fires an exception once this frequency exceeds a certain threshold. Even though simple, RipRop is effective. It recognizes all the exploits available in the "Exploits Database" [Anonymous 2014] that we have experimented with. Furthermore, we have not observed any false positive warning when applying our method on all the SPEC CPU 2006 [Henning 2006] programs. We reinforce our claims with two simple arguments. First, we show that although possible in some situations, circumventing our technique is hard enough to discourage attackers. Secondly, we explain that our approach has a low hardware footprint and does not affect the application's runtime.

2. Architecture

We have two versions of our tool: one for Windows, the other for Linux. It is built on top of Pin [Luk et al. 2005], a framework to instrument binary programs dynamically, i.e., at runtime. Thus, our prototype does not require us to recompile the programs that we analyze, and it handles seamlessly user and library code in the same way. RipRop is robust enough to detect ROP-based attacks in applications deployed in a production scenario. To meet such goal, RipRop gives us an *exact* view of the instructions that are processed by the hardware, including library code which is linked dynamically with the application.

Pin is a binary instrumentation framework maintained by Intel. It supports the architectures IA-32 and x86-64, and has versions that work in Android, Linux, OSX and Windows. Tools that use Pin are called *Pintools*. Our RipRop is a Pintool. We chose Pin to implement RipRop because it shows better performance than other binary instrumentation options publicly available [Bruening 2004, Nethercote and Seward 2007], and because it has a large community of users. Pin works in a just-in-time fashion: it intercepts an instruction before it is processed by the hardware; it then generates and runs new binary code, to handle that instruction; and finally, it ensures that, once the instruction is executed, the control flow will be given back to the resident pintool. Notice that, even though the monitored application, the resident pintool, and the Pin framework itself share the same address space, none of these processes share library code. In this way, Pin avoids undesired interactions between itself and the application that it instruments.

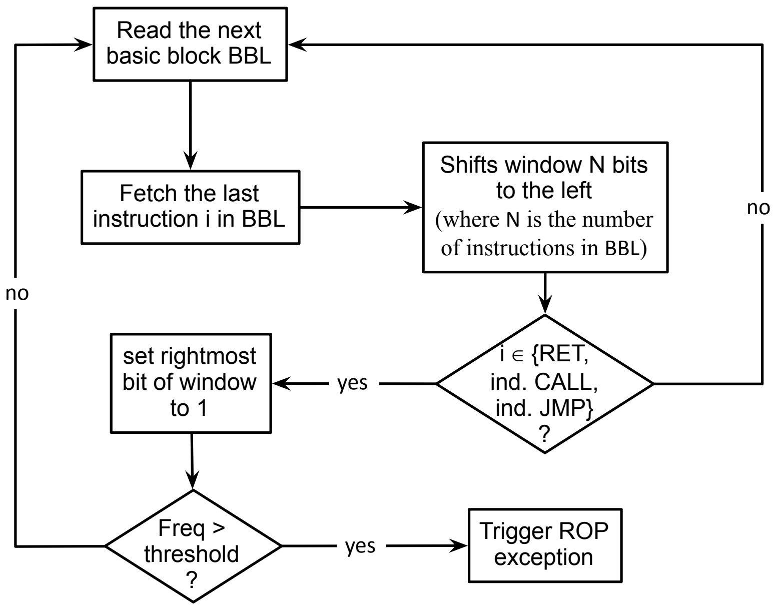
Read the next basic block BBL
Fetch the last instruction i in BBL
Shifts window N bits to the left (where N is the number of instructions in BBL)
no
no
i ∈ {RET, ind. CALL, ind. JMP} ?
yes
set rightmost bit of window to 1
Freq > threshold ?
yes
Trigger ROP exception

Figure 1. An overview of RipRop.

Figure 1 provides an overview of RipRop. RipRop keeps track of a window of bits that represent the last W instructions observed during the execution of a program. W is parameterizable, and in Section 4 we experiment with different values of it. Whenever we read a basic block of size N we shift the window N bits to the left, zeroing the rightmost bits. After shifting the window, we analyze the instruction at the end of the basic block. If this instruction is a function terminator, then we set the rightmost bit of the window to one, otherwise we leave it as zero. Before moving on to read the next basic block, we check the frequency of function terminators. If the window contains a number of them that is higher than a threshold, then RipRop issues a warning that indicates a potential ROP attack. Like the size of the window, the threshold is also parameterizable.

To run RipRop, one must first download and install Pin. RipRop is then executed alongside with Pin, via command line, giving as argument the name of the application that shall be monitored. At 3:28s in the video[1], we show how to execute our tool to monitor a music player program.

As a drawback, the application under monitoring is slowered down: since each and every basic block must be processed by our Pintool before actual execution, RipRop adds an overhead to the original performance. Despite this, the running time of programs being monitored by RipRop is feasible, as we show in the video and present in the experimental results available in Section 4.

3. Core Insights

ROPs are built around sets of *gadgets*. A gadget is a small sequence of instructions that ends with an indirect branch (IB). In the x86 architecture, for instance, IBs are return instructions (RETs), indirect jumps or indirect call instructions. As an example of attack, Shacham *et al.* [Shacham et al. 2004, p.299] shows how to chain these gadgets to overcame a protection known as *Write*$\oplus$*Execute*, which marks memory addresses where data is stored as non-executable. A common trait among gadgets observed in practice is their size: they tend to be small. In general, ROP-based exploits use gadgets with 1-3 instructions only [Yuan et al. 2011]. When crafting an exploit, the attacker can only build gadgets out of sequences of instructions that are already part of the program code. Long sequences may create undesirable side effects, such as overwriting registers or the stack in unwanted ways. The contents of the stack need to be carefully prepared, so that a ROP-based attack can be successful. The probability that a store operation changes this content, thus breaking the progress of the attack, is non-negligible.

3.1. The Key Observation

As mentioned before, the gadgets tend to be formed by small sequences of instructions. Therefore, the *density* of IBs during the execution of a ROP-based exploit is likely to be high. We measure the density of IBs as the number of such instructions within a given *window*. In our context, a window is a sequence of instructions in the program's execution trace. If we consider, for instance, the last 32 instructions executed by a program under attack, then gadgets containing between 1 and 3 instructions would give us a density of at least 10 IBs in a 32-instruction window.

The table in Figure 2 shows numbers taken from 15 actual ROP-based attacks publicly available in the Exploit Database [Anonymous 2014]. We report the number of instructions that are part of the gadgets used in the exploit, and we report the number of IBs present in this sequence. This last value equals the number of gadgets themselves. These two quantities are readily available in the code of the exploits. In each one of these exploits, a window of 32 instructions would register, at its peak, a high number of IBs. We

[1]`https://youtu.be/EouKW1UUutU?t=3m28s`

Application	Instructions in Exploit	Function Terminators	Density
	Windows		
Wireshark*	49	24	0.49
DVD X Player*	62	28	0.45
Zinf Audio Player*	95	38	0.4
D. R. Audio Converter*	69	36	0.52
Firefox - use aft free*	51	21	0.41
Firefox - integer ovf*	36	18	0.5
PHP	53	21	0.40
AoA Audio Extractor	212	81	0.38
ASX to MP3 Conv	150	59	0.39
Free CD to MP3 Conv	47	19	0.40
	Linux		
ProFTPD Debian*	66	24	0.36
ProFTPD Ubuntu*	45	19	0.42
PHP	81	27	0.33
Wireshark	69	30	0.43
NetSupport	45	14	0.31

Figure 2. Sum of instructions in the gadgets used to craft a ROP-based attack, and number of cross-IBs among these instructions. We mark with * the exploits that use, in addition to RET instructions, also indirect jumps or indirect calls. We have run all the exploits in an x86 machine.

have observed a minimum of at least 11 IBs in each example that we have tested. On the other hand, during a legitimate execution of the applications, none of them has presented more than 9 IBs per window. This fact leads us to our key observation:

Observation 3.1 *In the 15 examples of exploits evaluated, there exists a well-defined separation between the density of IBs in the exploit and in the actual execution of the application.*

3.2. A Brief Discussion about False Positives

If we assume that we have a ROP-based attack in course whenever we observe a high-density of IBs, then false positives may happen if such density is achieved by the legitimate execution of code. We studied four situations in which we can expect a high density of IBs: (i) loops with very large bodies; (ii) invocation of short functions within loops having small bodies; (iii) bytecode interpreters and (iv) recursive function calls. We have designed and implemented a series of micro-benchmarks that exercises each one of these scenarios. Our toils let us conclude that even under these extreme situations we can expect a relatively low density of IBs. Further, using both `gcc` and MS Visual Studio 2013, we have not been able to produce a binary that yields a density of one IB per each two instructions.

3.3. A Brief Discussion about False Negatives

RipRop will produce a *false-negative* if an attacker can craft a sequence of gadgets that gives us a low density of IBs. We have tried to build such sequences on top of 10 exploits publicly available for Windows on x86. These efforts show that it is possible, although difficult, to circumvent our protection. To achieve such a purpose, we have trodden two different avenues: first, we have tried to simply increase the number of instructions after gadgets that we already knew for some of the publicly available exploits. These attempts were fruitless: the extra instructions break the exploits for the applications that we have analyzed. Second, we have tried to interpose *no-op gadgets* in between sequences of effective gadgets. A no-op gadget is a sequence of instructions ending with an IB, which does not cause side effects that could invalidate an exploit. In this case, we have been able to build effective exploits for two applications. Do these findings indicate that our defense is weak? We say: No! All the operative no-op gadgets belong into one of two categories: (i) static initialization sequences, and (ii) alignment blocks. In the former category we have long sequences of stores into fixed addresses, which are used to initialize static memory in C. In the latter we have no-op gadgets that correspond to code that the compiler inserts in-between functions to force an alignment of 16 bytes. We claim that it is possible to modify the compiler to remove these gadgets.

4. Experiments

Figure 3 shows the frequency of IBs for programs compiled with Visual Studio, running on Windows 7 in 32-bit mode. Each dot in the charts shows the maximum number of IBs observed on a sliding window of either 32, 64, 96 or 128 bits. When running the SPEC CPU programs, we have used their reference input, which is the largest set of input data that the benchmark provides. This data amounts to billions of instructions. From these charts, it is possible to draw three conclusions: (i) ROP-based exploits show higher peak density of IBs; (ii) larger window sizes tend to blur the distinction between legitimate and fraudulent executions; and (iii) at least for the benchmarks that we have tested, it is possible to determine a universal threshold that separates legitimate and illegal traces of instructions using a 32-instructions sliding window. We have observed similar behavior when running the exploits in the Linux system.

Universal vs Specific Thresholds. In our experiments, we have found a perfect distinction between legitimate and fraudulent executions in all the benchmarks that we have used, and in the two different operating systems that we have fiddled with. However, in Linux this threshold is made of only two instructions, a rather small gap. This small difference leads us to believe that, instead of searching for a universal threshold, a frequency-based mechanism to detect ROP attacks should consider thresholds that are specific to each application. It is possible to determine such thresholds either dynamically or statically. Dynamically, we can use an instrumentation framework to find out the peak frequency of IBs of each application. Statically, we can analyze the call graph of programs, to determine the shortest path between return instructions. We have already experimented with the first approach.

If we study again the chart in Figure 3, we see that only a few applications achieve an IB frequency that is close to the universal threshold. The table in Figure 4 makes this observation more explicit. We have grouped benchmarks according to their peak frequency. We notice that the peak frequency tends to be higher in the Windows OS; however, the frequency of the exploits is higher as well in this environment, as Figure 3 reveals. On the other hand, Linux, the system where we found the smallest gap between legal and illegal executions, has a very low peak (6/32 IBs per instruction) in almost half

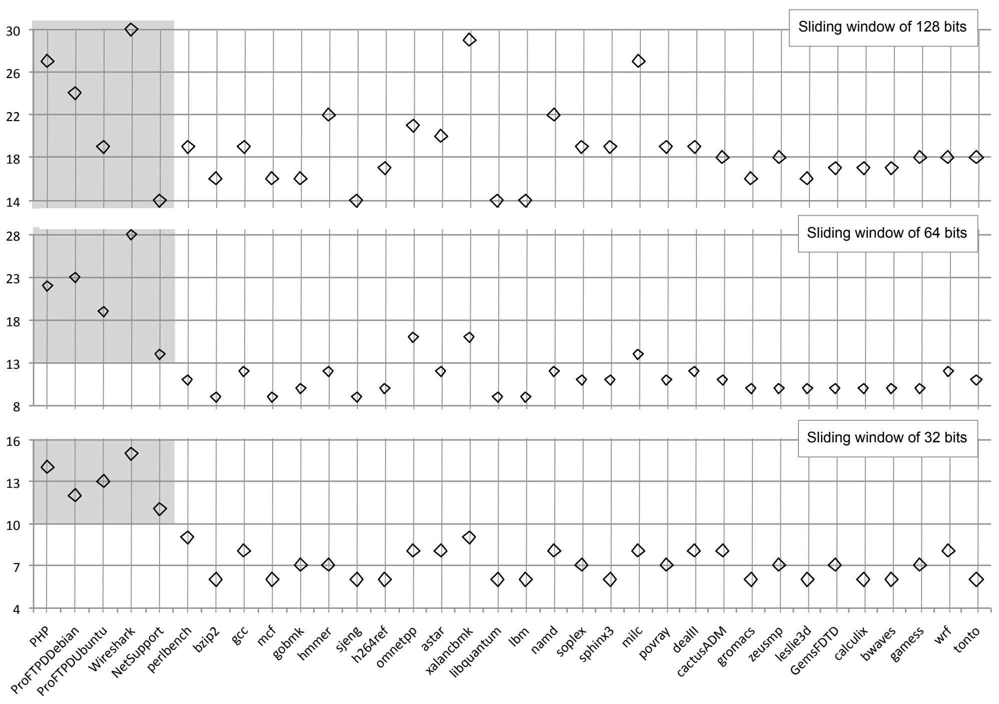

Figure 3. Maximum density of IBs in Windows 7, given different window sizes. The publicly available exploits are in the grey area.

the applications. The immediate conclusion that we draw from Figure 4 is that the most common frequency of IBs in our benchmarks is also the one that is the farthest away from the lowest peak observed in the actual exploits. Therefore, the use of a specific threshold for each application provides us with a more robust and accurate system.

Peak	Linux	Windows
10	0	11.1%
9	6.9%	88.9%
8	27.6%	0
7	24.1%	0
6	41.4%	0

Figure 4. Distribution of peak occurrences of IBs within a sliding window of 32 bits for SPEC CPU2006. The higher the peak, the higher the frequency of IBs, e.g., a peak of 10 instructions means that 10 IBs have been observed within a group of 32 successive opcodes.

Regarding the performance of applications monitored by RipRop, experiments with the SPEC CPU2006 suite of benchmarks show that our tool brings in an average overhead of 727%. This value is comparable to the overhead between 217% and 530% imposed by other protections that use dynamic code instrumentation mechanisms and unlike RipRop do not provide protection against all types of gadgets (terminated with returns, jumps or calls) [Chen et al. 2009, Davi et al. 2011]. Those overheads stem mainly from the use of dynamic instrumentation tools, which need to perform constant context switches with the instrumented application, disassembly codes and generate instructions, all while running the instrumented application. Nevertheless, various code optimizations aggregated to RipRop assure a competitive performance. When compared to DROP [Chen et al. 2009], which also uses Pin framework, RipRop imposes a lower overhead for the two SPEC benchmarks tested by the authors of DROP. In experiments with bzip2, DROP resulted in a computational cost of 1540%, considerably higher than the 809% recorded with RipRop. In tests with gcc DROP imposed an overhead of 960% while RipRop raised the CPU time 729%.

5. Related Work

There are strategies to identify ROP-based attacks that are similar to ours. The closest works are due to Chen *et al.* [Chen et al. 2009], Davi *et al.* [Davi et al. 2009], Pappas *et al.* [Pappas et al. 2013], and - more recently - Cheng *et al.* [Cheng et al. 2014]. These researchers propose to monitor the sequence of instructions that a program produces during its execution, flagging as exploits instruction streams that show a high frequency of RET operations. For instance, *Chen et al.* [Chen et al. 2009] fire warnings if they observe sequences of three RET operations and if each is separated from the others by five or less instructions. This approach yields more false positives than our sliding window: Chen *et al.* have reported one false alarm in a smaller corpus of benchmarks than the one we have used in this paper. Davi *et al.* have not implemented their approach, but discuss four situations that lead to false positives.

Pappas *et al* [Pappas et al. 2013] and Cheng *et al.* [Cheng et al. 2014] use the Last Branch Record (LBR) registers to detect chains of gadgets. LBR refers to a collection of register pairs that store the source and destination addresses of recently executed branches. They are present in Intel Core 2, Intel Xeon and Intel Atom. ARM has similar capabilities in some processors. We propose a different approach, which consists of a sliding window. We believe that our strategy may be impemented in hardware easier and faster than using the LBR approach. Carline *et al.* [Carlini and Wagner 2014] and Goktas *et al.* [Göktas et al. 2014] propose ways to bypass the LBR-based defense. However, we do not believe that these strategies work easily against our protection mechanism. For attackers to succeed, they must insert no-ops between gadgets. Just adding them at the end, or after 10 gadgets, like Carline does against the defense proposed by Pappas *et al.* and Cheng *et al.* is not enough to circumvent our approach.

6. Conclusion

This paper has presented RipRop, a novel technique to detect Return-Oriented Pogramming attacks: we check the density of indirect branches in the stream of executed instructions. A sliding window implemented over Pin lets us keep track of this information efficiently. Our approach does not make ROP exploits impossible to be implemented. The recent history of construction and defeat of ROP prevention mechanisms has shown us that this goal would be rather ambitious. Instead, we propose a cheap, original and efficient method to make ROP attacks substantially more difficult to craft.

References

[Anonymous 2014] Anonymous (2014). Exploit-DB. `www.exploit-db.com/`.

[Bruening 2004] Bruening, D. L. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology.

[Carlini and Wagner 2014] Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *Security Symposium*, pages 385–399. USENIX.

[Chen et al. 2009] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In *ISS*, pages 163–177. IEEE.

[Cheng et al. 2014] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). Ropecker: A generic and practical approach for defending against ROP attacks. In *NDSS*. Internet Society.

[Davi et al. 2011] Davi, L., Dmitrienko, A., and Egele, M. (2011). ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, pages 1–21.

[Davi et al. 2009] Davi, L., Sadeghi, A., and Winandy, M. (2009). Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *WSTC*, pages 49–54.

[Göktas et al. 2014] Göktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Security Symposium*, pages 417–432. USENIX.

[Henning 2006] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

[Lin et al. 2010] Lin, Z., Zhang, X., and Xu, D. (2010). Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *DSN*, pages 281–290. IEEE.

[Luk et al. 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM.

[Nethercote and Seward 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM.

[Pappas et al. 2013] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *SEC*, pages 447–462. USENIX.

[Roemer and Buchanan 2012] Roemer, R. and Buchanan, E. (2012). Return-oriented programming: Systems, languages and applications. *Trans. Inf. Syst. Secur.*, V.

[Shacham 2007] Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561. ACM.

[Shacham et al. 2004] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298–307. ACM.

[Yuan et al. 2011] Yuan, L., Xing, W., Chen, H., and Zang, B. (2011). Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *APSys*, pages 6:1–6:5. ACM.