Inference of Peak Density of Indirect Branches to Detect ROP Attacks

Mateus Tymburibá Rubens E. A. Moreira Fernando Magno Quintão Pereira

Department of Computer Science, UFMG, Brazil

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

Abstract

Return-Oriented Programming (ROP) is an exploitation technique widely known as the kernel of Stuxnet – and other highly engineered malwares –, developed and deployed as means of snooping into nuclear weapon facilities. A program subject to a ROP attack usually presents an execution trace with a high frequency of indirect branches. From this observation, several researchers have proposed to monitor the density of these instructions to detect ROP attacks. These strategies use universal thresholds: the density of indirect branches that characterizes an attack is the same for every application. This work shows that universal thresholds are easy to circumvent. As an alternative, we introduce an inter-procedural semi-context-sensitive static code analysis that estimates the maximum density of indirect branches possible for a program. This analysis determines detection thresholds for each application, thus making it more difficult for attackers to compromise programs via ROP. We have used an implementation of our technique in LLVM to find specific thresholds for the programs in SPEC CPU2006. By comparing these thresholds against actual execution traces of corresponding programs, we demonstrate the accuracy of our approach. Furthermore, our algorithm is practical: it finds an approximate solution to a theoretically undecidable problem, and handles programs with up to 700 thousand assembly instructions in 25 minutes.

Keywords Return Oriented Programming, Detection, Static Program Analysis, Security

1. Introduction

A *Return-Oriented Programming (ROP)* attack is a technique that consists in exploiting program security vulnerabilities to chain the execution of small sequences of binary code, namely *gadgets*. If a binary program is large enough, then it is likely to contain gadgets that, once chained, can give an attacker the tools to perform arbitrary activity in the host machine. Hovav Shacham [23] has shown how to derive a Turing complete language from gadgets in a CISC machine, and Buchanan *et al.* [2] have generalized this method to RISC machines. Despite its short history, ROP emerges today as an effective software exploit, being present in highly engineered malware, such as Stuxnet and Duqu [3]. The New York Times has extensively discussed the development and deployment of such ROP-based malware as a means of snooping into countries' nuclear weapon facilities[1]. Another testimony of ROP's importance is Microsoft's BlueHat'12 Software Security contest, that has awarded the first three prizes to ROP prevention and detection strategies[2].

A ROP exploit is likely to produce a program behavior that is different than its normal execution [18]. However, due to the very nature of this technique, the gadgets chained must be small: the attacker must pick gadgets with as few instructions as possible, to prevent undesirable side-effects from invalidating the attack. Thus, it is possible to identify ROP exploits by counting the number of indirect branches observed in a sequence of instructions fetched during the execution of a program. This approach is fairly studied in the literature: by tracking the density of indirect branches present in the instruction stream, different research groups have been able to provide ways to detect ROP-based attacks with various degrees of accuracy [6–8, 17].

Nevertheless, this detection approach is not foolproof: previous works [4, 12] demonstrate it is possible to interpose bigger gadgets between small ones, thus masquerading the attack. In this work, we look deeper into such evasion techniques and use compiler-related methods to design protection mechanisms that make it more difficult for ROP attacks to succeed. Our contributions are three fold: (i) we demonstrate a sliding window of 32 instructions to be enough to detect all ROP attacks from *Exploit Database* [1] that we have been able to reproduce; (ii) by generalizing the strategies adopted by Carlini *et al.* [4] and Goktas *et al.* [12], we show that even our mechanism may be circumvented with so-called *no-op gadgets*; (iii) and we describe a static analysis that estimates the peak density of indirect branches for a given program – raising the bar for successful attacks.

This report summarizes our original work [24], presented at ACM CGO, held in Barcelona, in 2016.

[1] `https://www.nytimes.com/topic/subject/cyberattacks-on-iran-stuxnet-and-flame`

[2] `http://www.microsoft.com/security/bluehatprize/`

2. Universal ROP-Detection Thresholds

There are several ways to trigger a ROP attack, being buffer overflow one of the most common techniques. This vulnerability lets the attacker overwrite the return address of a function with a sequence of new return addresses that invoke gadgets. Figure 1 shows the ROP attack used in Saif El-Sherei's tutorial [11]: it uses a particular string to open a shell session (`/bin/sh`) with the privileges of the exploited program. The exact contents of this string are immaterial to our explanation, except that it contains several addresses within the executable memory segment of the program under attack. Each address points to a sequence of instructions ending in an indirect branch (return operations, in the case of Figure 1). These sequences perform some simple task, and then invoke a return operation, which will move control to a new gadget, determined by the next address on the stack. The gadgets used are seen in Figure 1 (b), and have between two and three instructions, including the return operation. Larger gadgets may break the progress of the attack by overwriting previously manipulated memory slots or registers.

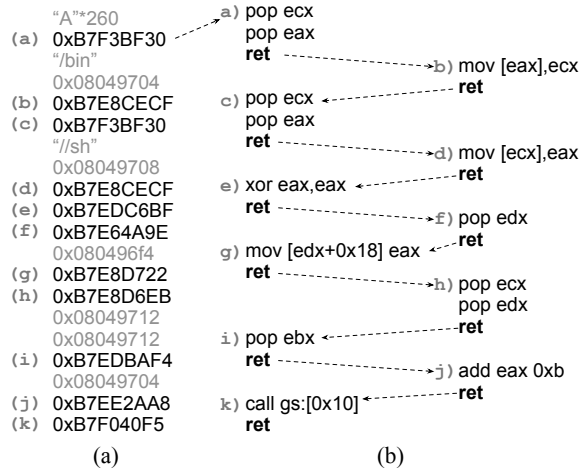

Figure 1. (a) Input string used in a buffer overflow ROP-attack. Data used in the exploit is written in light grey, and return addresses are written in black. Letters relate addresses to gadgets. (b) Gadgets that are chained during this attack.

2.1 In Search of a Universal Threshold

The small size of the gadgets is not a particularity of the example from Figure 1, but a characteristic of all documented ROP exploits we have found. One of our contributions is to show that the density of indirect branches within a fixed-size sliding window of instructions is an indicator strong enough to detect current ROP exploits. We have produced execution traces of programs using the instrumentation tool Pin [15], and have chosen 15 publicly available exploits from Exploit Database [1]. Ten of these programs run on Windows; the other five run on Linux. We have managed to successfully reproduce these exploits, hence, confirming the results described in Exploit Database. From our experiments, including with different sizes of sliding-window [24], we conclude that: (i) ROP-based exploits show high peak density of indirect branches; (ii) larger window sizes tend to blur the distinction between legitimate and fraudulent executions; and (iii) for the benchmarks we have tested, it is possible to determine a universal threshold.

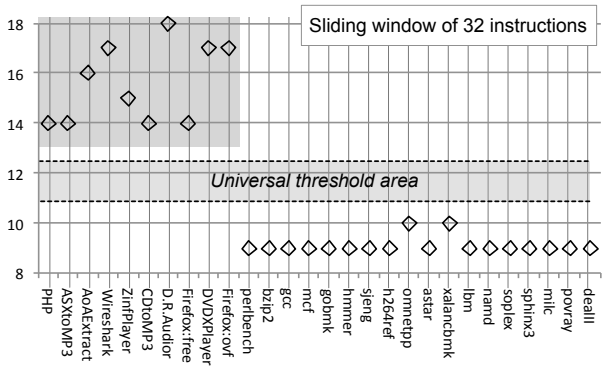

Figure 2. Maximum density of indirect branches in Windows 7, given different window sizes. The publicly available exploits are in the grey area.

2.2 Evading Universal Thresholds

By analyzing the data in Figure 2, we question the existance of gadgets large enough to lower the density below 10 indirect branches per group of 32 instructions. We have tried to build such sequences and our efforts show it is possible, although difficult, by use of *no-op* gadgets: a sequence of instructions ending with an indirect branch whose side-effects do not invalidate an exploit. We thus lower the density of indirect branches by interposing no-op gadgets between the valid sequences. However, it is possible to modify the compiler to remove such gadgets [24], thus rendering the counteraction ineffective.

2.3 Related Work

Memory and data protection [10, 19, 22], as well as control flow integrity [13] techniques, make it difficult for an attacker to take control of a program. Several works inspect the frequency of indirect branches in the instruction stream [6–8, 17], reporting exploits whenever the frequency surpasses a predefined threshold. However, as recently demonstrated [5], one may circumvent protection mechanisms by highjacking control from the program. Therefore, albeit we believe there is currently no general guard against ROPs, state-of-the-art defenses available today make it difficult for attackers to exploit software effectively.

2.4 Using Specific Thresholds to Improve Protections

Our static solution is not a competitor of previous frequency-based techniques, but a complement: as they rely on fixed thresholds to detect ROP exploits, we may improve their precision with tighter application specific thresholds. Because

they are smaller than the universal limit, specific thresholds force an attacker to use longer gadgets – which are more likely to invalidate the attack, as formerly discussed. Although unexplored so far, many researchers [6–8, 17] have suggested the possibility of using specific thresholds to enhance ROP detection mechanisms, and such approaches lack but the algorithms to produce the estimates. According to these researchers, albeit challenging, one may explore the peculiarities of each application to establish specific values able to consistently detect attacks, which we demonstrate, in Section 4, with analysis of publicly available ROP exploits.

3. Static Inference of Specific Thresholds

This section discusses the problem of inferring the peak density of indirect branches in a window of K instructions that can be observed during the execution of a program. Definition 3.1 states the problem of *Inference of Peak Density of Indirect Branches*, henceforth referred as IPD.

Definition 3.1 (IPD) *Given a program P, and two integers: K and R, is there an execution trace of P with no more than K instructions that contains R indirect branches?*

In principle, we could interpret P, logging the maximum number of indirect branches observed, but P may not terminate, and we would never be sure to have found its peak density. In other words, IPD is undecidable in the general case, since asking if the program P ever reaches a return instruction at any point of its execution qualifies as an *interesting property* in Rice's sense [20]. Given that the IPD is undecidable, we shall solve it via a conservative, partially context-sensitive, flow-sensitive static analysis.

3.1 δ-CFG: The Supporting Data-Structure

For any instruction I from P, our algorithm traverses every possible path of K instructions starting from I, counting up visited indirect branches. The worst case happens when each of the N instructions may branch to each other, $O(N^{(K+1)})$. However, we alleviate the algorithm's complexity by a constant factor, abstracting away CFG components irrelevant to solving IPD. Our simpler data-structure is called δ-CFG, and has three kinds of nodes: BBL, that let us represent blocks terminated by direct conditional or unconditional branches; RET, which are blocks terminated by return instructions; and FUN, terminated by function calls, thus linking functions' δ-CFGs and making our analysis inter-procedural. Each of these three kinds of nodes is associated with an integer n, which denotes the number of instructions present in that basic block. Figure 3 (a) a C program, and (b) its simplified assembly. Figure 3 (c) shows the δ-CFG of such program. The δ-CFG eliminates most of the instructions from the original CFG, but keeps the program flow. The search is then performed by a brute-force algorithm we describe in the the next section.

(a)

```
int x;

void foo() {
  if (x) {
    --x;
    foo();
  }
}

int main() {
  x = 10;
  foo();
}
```

(b)

```
A:  _foo:
      sub  esp 12
      cmp  dword ptr [x], 0
      je   C
B:    mov  eax, dword ptr [x]
      add  eax, -1
      mov  dword ptr [x], eax
      call _foo
C:    add  esp, 12
      ret

D:  _main:
      sub  esp, 12
      mov  dword ptr [x], 10
      call _foo
E:    add  esp, 12
      ret
```

(c)

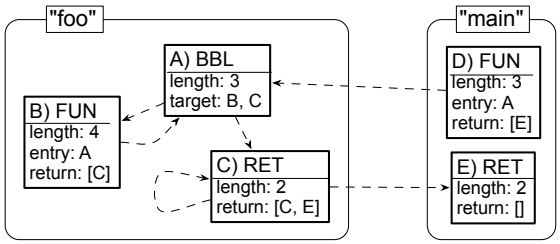

Figure 3. (a) Example program written in C. (b) Simplified x86 assembly of example program. (c) δ-CFG produced for the example program.

3.2 Algorithm to Estimate the Peak Density of Returns

Given a program P, the function deduce in Figure 4 estimates P's peak density of return instructions by visiting every path of size K starting at any node of P's δ-CFG. This SML/NJ's implementation has no omissions: Figure 4 is a running prototype of our algorithm. Pattern top::stack denotes a list with head top and tail stack. We use [] to represent empty lists. The construct fn x => e, at line 43, denotes an anonymous function with formal parameter x and body e. The datatype in lines 1-3 represents the δ-CFG. Each one of the three possible types of nodes is associated with an integer n, which represents the number of instructions in that node. Nodes of type BBL are also associated with two successors, left and right. Nodes FUN are also associated with an *entry point* and a *return point*. Finally, nodes RET are associated with a list of return points.

The function map applies explore onto every element of the list P (the input program). Function explore receives a *trip budget* T, a *return stack* and a node of the δ-CFG, and traverses it recursively until the budget is zeroed. If explore visits a FUN node, then it will store this node's return point onto the stack, as seen in line 40 of Figure 4. Upon reaching an indirect branch, e.g., a RET node, explore either resumes its search at the node on the top of the stack, (lines 23-26), or continues at every return address that may succeed a call to the function being traversed (lines 7-22). An intutive view of the algorithm is given in Figure 5, which shows the maximum densities from each node in the δ-CFG of the C program from Figure 3.

```
datatype δ-CFG = RET of int × list δ-CFG
               | BBL of int × δ-CFG × δ-CFG
               | FUN of int × δ-CFG × δ-CFG

fun max a b = if a > b then a else b

fun explore T [] (RET (n, return_addresses)) =
    if T ≥ n
    then
      let
      fun max_of_all_return_paths [] = 0
        | max_of_all_return_paths (next_addr::addresses) =
          let
            val max_so_far = max_of_all_return_paths addresses
            val dens_of_next = explore (T - n) [] next_addr
          in
            max max_so_far dens_of_next
          end
      in
        1 + max_of_all_return_paths return_addresses
      end
    else 0
  | explore T (top::stack) (RET n) =
    if T ≥ n
    then 1 + explore (T - n) stack top
    else 0
  | explore T stack (BBL (n, left, right)) =
    if T ≥ n
    then
      let
        val bestLeft = explore (T - n) stack left
        val bestRight = explore (T - n) stack right
        fun max a b = if a > b then a else b
      in
        max bestRight bestLeft
      end
    else 0
  | explore T stack (FUN (n, entry, next)) =
    if T ≥ n
    then explore (T - n) (next::stack) entry
    else 0

fun deduce P K = map (fn b => explore K [] b) P
```

Figure 4. Brute-force inference of indirect branches using window of size K.

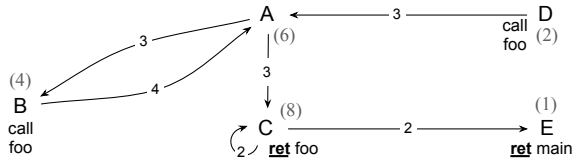

Figure 5. Algorithm from Figure 4 applied on the δ-CFG of Figure 3 (c). Numbers in parentheses in each block indicate the highest frequency of RETs estimated from that block (16-inst. window).

Our algorithm has a worst case complexity of $O(N^{(K+1)})$, where N is the number of instructions in the program, and K is the size of the detection window. However, most of the instructions in a typical program are actually sequential – branches correspond to less than 15% of the total [3]. Furthermore, most of the branches with more than one target have exactly two targets (conditional branches). These observations reduce the complexity of our algorithm to a $O(N \times 2^K)$ when all the instructions are 2-target branches – still a conservative assumption.

[3] Source: `http://www.strchr.com/x86_machine_code_statistics`

4. Experiments

This section validates the following statements: (i) specific thresholds are better than universal thresholds to prevent ROP attacks (Section 4.1); (ii) we find good approximations to peak density of indirect branches (Section 4.2); (iii) our algorithm scales well to large program sizes (Section 4.3).

4.1 Specific Thresholds Reduce Available Gadgets

ROP attacks require a fair number of available gadgets in the executable address space of the program under attack [21], otherwise it may not be possible to establish a chain of gadgets able to consolidate the exploit. Several strategies to combat ROP attacks try to limit the number of available gadgets, commonly by ensuring return operations to only divert execution to addresses that follow call instructions [7, 9, 13, 17, 25], or by replacing instructions that incidentally contain the code of indirect branches within their binary representation with alternative instructions with the same effect [14, 16]. Our specific thresholds also try to limit the amount of gadgets available to attackers: to camouflage an exploit, attackers will need gadgets greater than those capable of simulating a universal threshold, as an specific threshold is strictly lower than an universal one. Our results show that larger no-op gadgets are rare, thus raising the bar for successful ROP attacks to take place: nearly 60% of no-op gadgets have size 3; $\tilde{1}5\%$ are of size 4; $\tilde{5}\%$ of size 5; and the remaining distributed from 6 to 25. There are 75% less potential no-op gadgets of size 6 (used to bypass the average specific threshold) than 4 (used to bypass the universal threshold).

4.2 On the Accuracy of our Estimates

We compared our estimates of maximum density for SPEC-CPU2006 benchmarks with estimates retrieved using Pin [15]. Figure 6 shows that the peak densities indicated by our algorithm are equal to or higher than the values recorded dynamically, i.e., our algorithm is conservative, as it explores all possible execution paths – often difficult to achieve with non-exhaustive input set. Our static analysis exceeds dynamic monitoring in 64% of the benchmarks, and has the same accuracy in the remaining cases. The discrepancy between static and dynamic estimates for *gcc* and *omnetpp* are due to tail recursive functions – not triggered during dynamic analysis. Figure 6 also illustrates the impact of the maximum density assumed for library functions (whose code is not known). For these functions, we assumed that they have at most 1 return every 4, 5 or 6 instructions executed and observed that a maximum density of 1/5 ensures that no false positives occur, i.e., static estimates are is greater than or equal to dynamically monitored values.

4.3 Performance of the Inference Algorithm

We also measured the execution time of our alogorithm as a function of the number of assembly instructions contained in

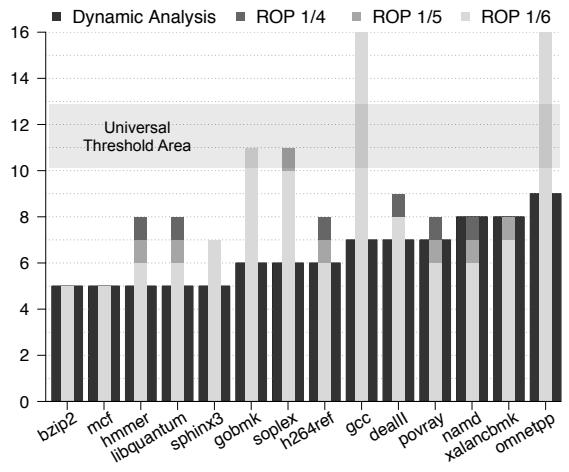

Figure 6. Specific threshold of return instructions for programs in the SPEC CPU2006 benchmark suite. Grey bars show specific thresholds found with the algorithm of Section 3, and black bars show values found dynamically. $1/4$ is the density that we have assumed for library functions, which we cannot analyze statically.

each program used as benchmark. Using a window size of 32 instructions, our runtime scales linearly on the program size (by a coefficient of determination of 0.91491). The largest SPEC program, *xalancbmk*, with approximately 700 thousand instructions, was analyzed in 25 minutes. On average, each benchmark has approximately 200,000 instructions and was analyzed in 7 minutes. Therefore, we believe that our solution can be used in practice to estimate specific thresholds for general applications, despite its exponential complexity.

5. Conclusion

Our new static analysis determines conservative, yet precise approximations of peak density of indirect branches. To support this claim, we have analyzed instruction traces from programs in SPECCPU2006. Our results are tighter than a universal threshold, making it much harder to produce an undetectable ROP attack in the exploits that we have been able to reproduce. Our method is practical, scaling up to programs with approximately 700 thousand assembly instructions, and decreases effectively the number of gadgets available for an exploit. Recent history has shown no foolproof method to prevent ROP-based exploits, but we believe specific thresholds raise the bar for successful attacks.

References

[1] Anonymous. Exploit-DB, 2014. `www.exploit-db.com/`.

[2] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS*, pages 27–38. ACM, 2008.

[3] J. Callas. Smelling a RAT on duqu, 2011. On-line.

[4] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. USENIX, 2014.

[5] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. USENIX, 2015.

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

[7] Y. Cheng, Z. Zhou, and M. Yu. ROPecker: A generic and practical approach for defending against ROP attacks. *NDSS*, 2014.

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

[9] J. Demott. /ROP - BlueHat Prize Submission, 2012. URL `http://www.vdalabs.com/tools/DeMott_BlueHat_Submission.pdf`.

[10] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. ACM PLDI, 2006.

[11] S. El-Sherei. Return oriented programming (rop ftw), 2013. URL `www.elsherei.com--whitepaper`.

[12] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. USENIX, 2014.

[13] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. USENIX, 2002.

[14] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. ACM, 2010.

[15] C.-K. e. a. Luk. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*. ACM, 2005.

[16] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadgetless binaries. ACM, 2010.

[17] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Security Symposium*, pages 447–462. USENIX, 2013.

[18] D. Pfaff, S. Hack, and C. Hammer. Learning how to Prevent Return-Oriented Programming Efficiently. LNCS, 2015.

[19] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*. IEEE, 2006.

[20] H. G. Rice. Classes of recursively enumerable sets and their decision problems. 1953.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *SEC*, pages 25–25. USENIX, 2011.

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. USENIX, 2012.

[23] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls. In *CCS*. ACM, 2007.

[24] M. Tymburibá, R. Moreira, and F. Pereira. Inference of peak density of indirect branches to detect rop attacks. In *CGO*. ACM, 2016.

[25] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *DSN*, pages 1–12. IEEE, 2012.