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

A program subject to a Return-Oriented Programming (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 techniques use universal thresholds: the density of indirect branches that characterizes an attack is the same for every application. This paper 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.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Compilers

General Terms Languages, Security, Experimentation

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

1. Introduction

A *Return-Oriented Programming (ROP)* attack is an exploitation technique that consists in reusing segments of instructions within the binary code of a compromised program to execute some arbitrary activity. If a binary program is large enough, then it is likely to contain bit sequences that encode instructions which, if chained together, can give an attacker the tools to carry out malicious actions. Hovav Shacham [30] has shown how to derive a Turing complete language from these sequences in a CISC machine, and Buchanan *et al.* [4] have generalized this method to RISC machines. In spite of its short history, Return-Oriented Programming emerges today as an effective software exploitation method, being present in a few highly engineered malware, such as Stuxnet and Duqu [5]. As a testimony of ROP's importance, the first three prizes awarded in the BlueHat'12 Software Security contest have gone to ROP prevention and detection strategies[1].

A ROP exploit is likely to produce a program behavior that is different than its normal execution [25]. One of the reasons behind this discrepancy is due to the fact that this type of malicious code consists of small sequences of instructions – henceforth called *gadgets* – ending with an indirect branch, e.g., a return, an indirect call or an indirect jump. The very nature of this attack technique forces these gadgets to be small. 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 [8–10, 17, 19, 24, 33].

Nevertheless, this detection approach is not foolproof, as independently demonstrated by Carlini *et al.* [6] and Goktas *et al.* [16]. These researchers have shown that it is possible to interpose long gadgets between small ones, thus masquerading the occurrence of the attack. In this paper we look deeper into techniques such as Carlini's and Goktas' and use compiler-related methods to design protection mechanisms that make it more difficult for them to succeed. With this

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

goal, we bring in three contributions, which we enumerate as follows:

- First, in Section 2.1 we demonstrate that a window of 32 instructions is enough to detect ROP attacks in all the 15 exploits publicly available in the *Exploit Database* [1] that we have been able to reproduce. This detection mechanism is more general than other frequency based techniques already described in the literature [8–10, 24].
- Second, in Section 2.2 we show that even this mechanism can be circumvented, depending on compilation choices while generating binaries. To this end, we rely on some default characteristics of the Gnu C Compiler (`gcc`). This evasion technique generalizes the strategies adopted by Carlini *et al.* [6] and Goktas *et al.* [16], which we have mentioned before.
- Finally, in Section 3 we describe a static analysis that estimates the peak density of indirect branches for a given program. By using this specific threshold, instead of a universal threshold, we show that it is possible to design detection mechanisms that raise the bar for exploits such as those that we introduce in Section 2.2.

As we show in Section 4, our new static analysis determines conservative, yet precise, approximations of peak density of indirect branches. To support this claim, we have analyzed instruction traces produced by the programs in SPEC CPU2006. Our results are tighter than a universal threshold. Hence, our specific detection thresholds make it much harder to produce an undetectable ROP attack in the exploits that we have been able to reproduce.

2. Universal ROP-Detection Thresholds

The goal of this section is three fold. First, we show that it is possible to use a window of fixed length of program instructions to identify current Return-Oriented Programming exploits. This approach generalizes previous frequency-based ROP detection techniques; hence, it is harder to evade than the techniques presently in use. Additionally, being able to use a fixed-size window to detect ROP attacks is an essential requirement of the algorithm that we later introduce in Section 3. Were we not dealing with a fixed-size window, then estimating peak densities of indirect branches would leave us with an undecidable problem. Second, we show that even a window-based detection mechanism can be circumvented by a determined attacker, due to the way that compilers generate code. Third, we demonstrate how specific detection thresholds can improve the security level of frequency-based protections. Before we dive into these three goals, we provide an intuition on why a ROP-exploited program tends to show a high density of indirect branches. To this end, Figure 1 shows the ROP attack used in Saif El-Sherei's tutorial [14].

There are several ways to trigger a ROP attack. One of the most common techniques is to exploit a buffer overflow. This vulnerability lets the attacker overwrite the return address of a function with a sequence of new return addresses that

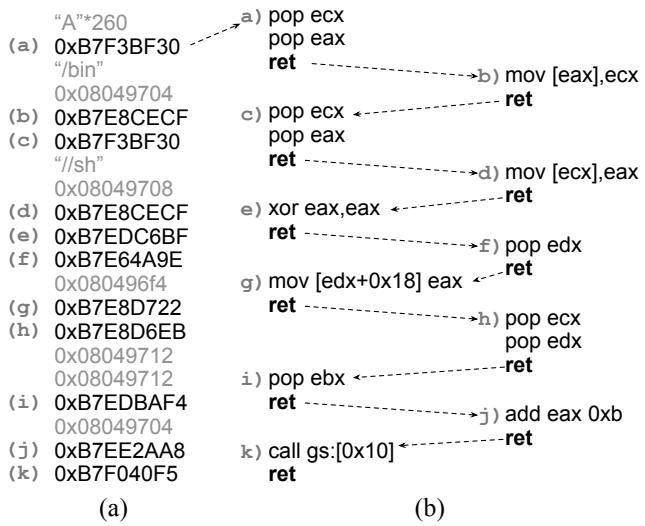

Figure 1. (a) Input string used in a buffer overflow attack that triggers a ROP exploit. Data used in the exploit is written in light grey, and return addresses are written in black. Letters in front of addresses indicate the gadget that will be invoked. (b) Gadgets that are chained during this attack.

invoke gadgets. The attack in Figure 1 uses a particular string to open a shell section (`/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 one of these addresses point to a sequence of instructions that ends in an indirect branch (return operations in the case of Figure 1). These sequences, e.g., gadgets, 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 in this attack are seen in Figure 1 (b). They have between two and three instructions, including the return operation. Larger gadgets could break the progress of the attack by overwriting some memory slot or register used during the exploitation. As we show in Section 2.1, this small size is not a particularity of this example, but a characteristic of all documented ROP exploits that we have found.

2.1 In search of a universal threshold

One of the contributions of this work 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. To achieve this goal, we have produced execution traces of programs using Pin [22], a framework to instrument binary code at runtime. For this experiment, we chose 15 publicly available exploits in the Exploit Database [1]. Ten of these programs run on Windows; the other five run on Linux. We chose these 15 exploits because they were all the attacks that we could reproduce successfully; hence, confirming the results described in Exploit Database. Figure 2

Application	Instructions in Exploit	Indirect Branches	Ratio (Average Gadget Size)
	Windows		
Wireshark	49	24	2.04
DVD X Player	62	28	2.21
Zinf Audio Player	95	38	2.50
D. R. Audio Converter	69	36	1.92
Firefox - use aft free	51	21	2.43
Firefox - integer ovf	36	18	2.00
PHP	53	21	2.52
AoA Audio Extractor	212	81	2.62
ASX to MP3 Conv	150	59	2.54
Free CD to MP3 Conv	47	19	2.47
	Linux		
ProFTPD Debian	66	24	2.75
ProFTPD Ubuntu	45	19	2.37
PHP	81	27	3.00
Wireshark	69	30	2.30
NetSupport	45	14	3.21

Figure 2. Proportion of indirect branch instructions in cataloged exploits. Each gadget has, on average, 2.46 instructions. This results in a high frequency of indirect branches.

makes it explicit the "fingerprint" of these exploits: all of them show execution traces with a very high density of indirect branches. This is a consequence of having very small gadgets: only 2.46 instructions, on average. The smaller the gadgets, the higher the frequency of indirect branches.

This "fingerprint" is even more clear when we compare the traces of these exploits with the traces of legitimate executions of programs. Figure 3 shows an analysis of density of indirect branches in two sets of programs: the 10 ROP exploits that we have reproduced in Windows, and the programs available in the SPEC CPU2006 benchmark suite. All the applications have ben compiled with Visual Studio 7 in 32-bit mode to a x86 machine running Windows 7 Ultimate. Each dot in the charts show the maximum number of indirect branches observed on a sliding window of either 32, 64, 96 or 128 instructions. When running the SPEC CPU programs, we have used their reference input, which is the largest set of input data that the benchmark provides. We took about 3 days to run all the programs used to produce each of these charts – this data amounts to billions of instructions. Notice that we run each benchmark just once, as we are counting only discrete events, and there is no variation in our results.

These charts let us draw three conclusions: (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) at least for the benchmarks that we have tested, it is possible to determine a universal threshold that separates legitimate and illegal executions. In this particular experiment, all the publicly available exploits had more than 13 indirect branches per window

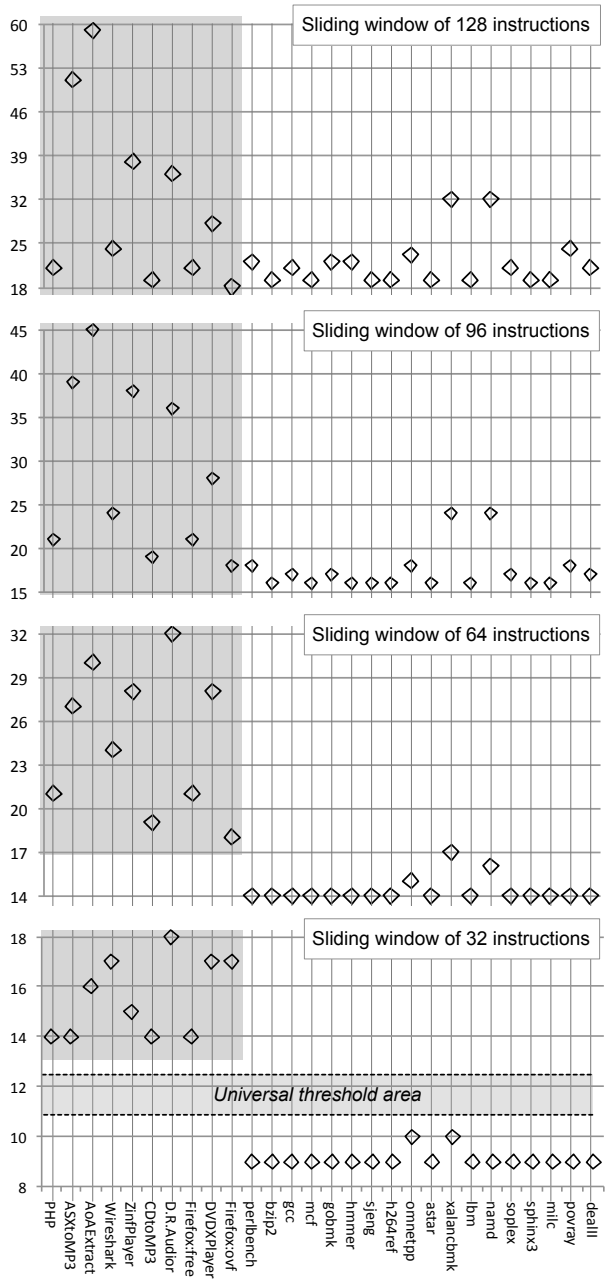

Figure 3. Maximum density of indirect branches in Windows 7, given different window sizes. The publicly available exploits are in the grey area.

of 32 instructions – at the point of maximum density. On the other hand, none of the SPEC programs showed more than 10 indirect branches per 32-instructions. We performed similar experiments on 64-bit mode Windows and on Linux (32 and 64-bit mode) with SPEC benchmarks and the 5 Linux ROP exploits listed in Figure 2. In this case, the benchmarks were compiled with `gcc` 4.6.3. All the results obtained closely resemble the behavior shown in Figure 3, which reinforces these conclusions.

2.2 Evading universal thresholds

The data seen in Figure 3 raises the following question: is it possible to find gadgets that are large enough to lower the density of indirect branches below 10 occurrences per group of 32 instructions? We have tried to build such sequences on top of the ten exploits used in Section 2.1. These efforts show that it is possible, although difficult, to find such gadgets. We define a *no-op* gadget as a sequence of instructions ending with an indirect branch, which neither changes the state of the registers or the stack used in the exploit, nor cause a memory violation (memory access, privileged instruction, etc). In other words, the instructions in a no-op gadget do not cause side effects that could invalidate an exploit. Our idea is to interpose no-op gadgets between the valid sequences, in order to lower the density of indirect branches observed in an exploit.

We have used the open-source tool Mona [13] to find no-op gadgets. Mona gave us, on average, 58,000 potential gadgets per each one of the ten vulnerable applications seen in Figure 3. Trying one by one all of these gadgets would be too time consuming; hence, we have defined four criteria to select good candidates. We choose gadgets that: (i) have more than five instructions; (ii) have no instructions that change the stack (`push`, `pop`, `add esp`, etc); (iii) have no instructions only allowed in privileged mode (`cli`, `ltr`, `smsw`, and 21 more); (iv) do not write on registers used in the exploit. Once we had a list of candidates, we sorted them according to two criteria: (a) increasing order on the number of indirect memory accesses; and (b) decreasing order on the total number of instructions. Criterion (a) is useful, because a large part of an application's address space is not accessable.

To check which gadgets work in practice, we decided to test all the candidates found in `Audio Extractor` and in `CD to MP3 Converter`. We chose these two applications because they have the lowest number of candidates (21 and 52, respectively). We did not test more applications because this process is very time consuming. Checking if a sequence of instructions can be interposed between the other gadgets of an original exploit is a manual process. To this end, we must adjust the return addresses used in the attack, and must check if the final effect of the exploit is still observable. We estimate that testing these 21 + 52 no-op candidates took us more than 70 man-hours of work.

In total, we found (3 + 6) functional no-op gadgets out of a universe of 21 + 52 candidates having between seven and ten instructions. The key conclusion of this experiment is that: *it is feasible to circumvent a window-based detection scheme using no-op gadgets*. All the no-op gadgets that worked 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 compilers insert to force an alignment of blocks of instructions. Most processors fetch instructions in aligned 16-byte or 32-byte blocks. Hence, it can be advantageous to align branch targets by 16 in order to minimize the number of 16-byte boundaries in the code. For instance, in `gcc` this alignment is controlled by the `-falign-labels` and `-falign-jumps` arguments, which are default in optimization levels `-O2` and `-O3`. The three valid no-op gadgets in `ASX to MP3 converter` were in category (ii). `CD to MP3 converter` had three no-ops in category (i), plus three more in category (ii).

Nevertheless, we claim that it is possible to modify the compiler to remove these gadgets. Elements in the first group, i.e., static initialization, can be removed by the simple expedient of interposing instructions that write to registers with stores into static memory. This process does not change the semantics of the program, because no register is in use when static data is being initialized. To eliminate the second category of no-op gadgets, it is possible to replace the padding code by no-ops of up to nine bytes, which would reduce the number of instructions in the innocuous gadgets considerably [18]. All the other no-op candidates failed to lead us to valid exploits, either because they write into general purpose registers or in memory addresses necessary for the exploit. The explanation for this failure is straightforward: it is hard to find no-op gadgets that fit anywhere in the ROP chain. The x86 architecture has very few registers, and usually gadgets with more than seven instructions break the progress of the attack by writing in some of these registers.

2.3 Using specific thresholds to improve protections

One way to improve the level of protection offered by mechanisms based on control of the frequency of indirect branch instructions is to establish a specific threshold for each application to be protected. Because they are smaller than the universal threshold, specific thresholds force an attacker to use longer gadgets. However, as previously discussed, larger gadgets are more likely to corrupt the contents of the memory or any register used during the attack. Alternative attacks using no-op gadgets also become more difficult, since longer no-op gadgets are scarcer, as we show in Section 4. Although advantageous, specific thresholds are not yet adopted due to the lack of algorithms to estimate them. In Section 3 we describe a static analysis that solves this challenge.

Although unexplored so far, the possibility of using specific thresholds to enhance ROP detection mechanisms has been suggested by several researchers. As an example, the authors of schemes based on the control of indirect branches density stress the possibility of improvement through the use of specific thresholds [8–10, 19, 24]. For instance, *Jiang et al.* [19] claim that "the threshold must be carefully chosen in order to balance the false positive rate and the false negative rate" and *Chen et al.* [8] point out that "the thresholds can be chosen by the user". This opinion is also shared by authors who developed methods to overcome protections. Goktas *et al.* [16] devote an entire subsection to discuss counter-

measures based on per-application parameters. According to them, albeit challenging, one can explore the peculiarities of each application to establish specific values able to consistently detect attacks. Indeed, analysis of publicly available ROP exploits confirms the theory that specific thresholds can improve indirect branch frequency based protections. In Section 4, we will show that for the vulnerable applications analyzed, the use of specific thresholds reduces by 75% the availability of no-op gadgets.

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?*

To solve IPD, we could, in principle, interpret P, logging the maximum number of indirect branches observed at any point during this interpretation. However, P may not terminate, and we would never be sure to have found or not its peak density. In other words, IPD is undecidable in the general case, as a consequence of Rice's Theorem [27]. To see why, notice that we could ask if the program P ever reaches a return instruction at any point of its execution. "Reaching a return instruction" qualifies as an *interesting property* in Rice's sense. Given that the IPD is undecidable, we shall solve it via a conservative, partially context-sensitive, flow-sensitive static analysis. We describe this static analysis in the rest of this section.

3.1 δ-CFG: the supporting data-structure

For any instruction I in the program's control flow graph, our algorithm traverses every possible path of K instructions starting from I, counting up every indirect branch visited in this process. The worst case for the performance of this approach happens when each instruction may branch to each other, a situation that would give us searches as costly as $O(N \times N^K) = O(N^{(K+1)})$ starting at each of the N instructions that constitute the program. Even though we shall not reduce this asymptotic complexity, we will try to alleviate its constant factor by abstracting away every element of the CFG that has no influence in the solution of IPD. Therefore, we will not work on the program's control flow graph, but in a data-structure henceforth called δ-CFG.

The δ-CFG has three kinds of nodes: BBL, RET and FUN. Each one of these nodes represent a different kind of *basic block*, e.g., a sequence of instructions with only one entry-point, and only one exit-point. The difference between these blocks of instructions is the operation that ends it. Nodes BBL represent basic blocks that end with conditional

(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)

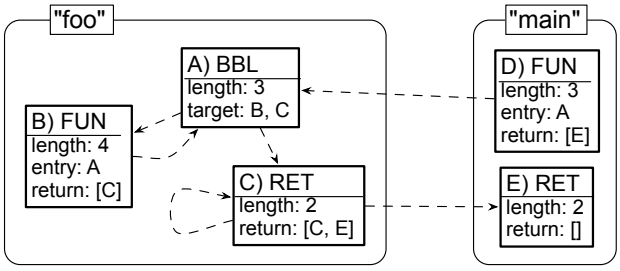

Figure 4. (a) Example program written in C. (b) Simplified x86 assembly of example program. (c) δ-CFG produced for the example program.

or unconditional direct branch instructions. For simplicity, we shall assume that each such block has two targets. Nodes RET represent blocks that end with return instructions. Each RET node that is part of a function f is associated with a list of instructions that succeed function calls to f. Nodes FUN represent blocks that end with function calls. These nodes are associated with an entry-point, and a return-point. The entry point of a node FUN that invokes a function g is the first basic block of g. The return point of a node FUN at program address ℓ is the instruction at label $\ell + 1$. Therefore, these nodes lets us represent the entire control flow graph of a program, considering direct and indirect branch instructions. 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 4 (a) shows an example of program written in C, which we shall use to explain how our algorithm works. A simplified assembly version of this program is given in Figure 4 (b). Figure 4 (c) shows the δ-CFG of the example program. In this section, we shall use this – rather artificial – program, to illustrate our technique to estimate peak density of indirect branches. The δ-CFG eliminates most of the instructions present on the original CFG, but it keeps the program flow. Thus, we can use this concise structure to find paths within the program code. This search is performed by a brute-force algorithm that we describe in the the rest of this section.

```
1  datatype δ-CFG = RET of int × list δ-CFG
2                 | BBL of int × δ-CFG × δ-CFG
3                 | FUN of int × δ-CFG × δ-CFG
4
5  fun max a b = if a > b then a else b
6
7  fun explore T [] (RET (n, return_addresses)) =
8      if T ≥ n
9      then
10       let
11       fun max_of_all_return_paths [] = 0
12         | max_of_all_return_paths (next_addr::addresses) =
13           let
14             val max_so_far = max_of_all_return_paths addresses
15             val dens_of_next = explore (T - n) [] next_addr
16           in
17             max max_so_far dens_of_next
18           end
19       in
20         1 + max_of_all_return_paths return_addresses
21       end
22     else 0
23   | explore T (top::stack) (RET n) =
24     if T ≥ n
25     then 1 + explore (T - n) stack top
26     else 0
27   | explore T stack (BBL (n, left, right)) =
28     if T ≥ n
29     then
30       let
31         val bestLeft = explore (T - n) stack left
32         val bestRight = explore (T - n) stack right
33         fun max a b = if a > b then a else b
34       in
35         max bestRight bestLeft
36       end
37     else 0
38   | explore T stack (FUN (n, entry, next)) =
39     if T ≥ n
40     then explore (T - n) (next::stack) entry
41     else 0
42
43 fun deduce P K = map (fn b => explore K [] b) P
```

Figure 5. Brute-force inference of indirect branches using window of size K.

3.2 Algorithm to estimate the peak density of returns

Given a program P, the function deduce in Figure 5 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 implementation is written in SML/NJ's syntax, and it has no omissions: Figure 5 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 that has 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.

In line 43 of Figure 5, function deduce performs an exhaustive search bounded by the window size K starting from any node in the program. The function map applies explore onto every element of the list P, which represents the input program. Function explore receives a *trip budget* T, a *return stack* and a node of the δ-CFG. While explore's trip budget is

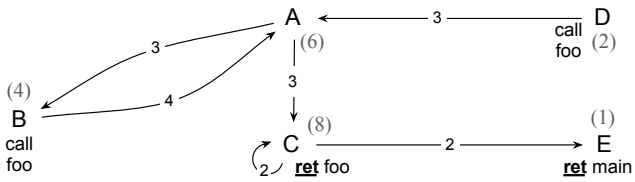

Figure 6. Intuitive view of the algorithm from Figure 5, when applied on the δ-CFG of Figure 4 (c). Numbers in parentheses above block letters indicate the highest frequency of RETs that our algorithm obtains starting the search from that block, considering a window of 16 instructions.

larger than 0, it will visit more nodes in the δ-CFG. Hence, the trip budget represents the part of the window that we must still fill up with instructions while searching for a peak density. 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 5. Upon reaching an indirect branch, e.g., a RET node, explore resumes its search at the node on the top of the stack, in case its trip budget is still positive, as seen in line 23-26 of Figure 5. Otherwise, explore continues the search at every return address that may succeed a call to the function that is being traversed, as seen in lines 7-22.

We shall use Figure 6 to provide an intuitive description of our algorithm. The figure shows another view of the δ-CFG seen in Figure 4 (c). The weights on the edges represent number of program instructions. We say that the *weight of a path* is the sum of all the weights on the edges that constitute this path. The IPD problem asks for a path weight K containing R indirect branches. Our algorithm starts a graph traversal at each node of the δ-CFG, stoping each search upon finding a path of weight K. During this search, every edge can be traversed; the only special cases are the RET nodes. Upon visiting a FUN node that calls a function f, we store f and its return address onto the stack. When leaving a RET node that returns from a function f, we check if f is on the top of the stack. If that is the case, then we can only continue the search at the return point of the last call to f that we have seen, which is naturally onto the stack. In this example, the peak density that our algorithm finds for a window of 16 instructions is 8, which corresponds to eight visits to the edge $C \rightarrow C$ in Figure 6. This scenario happens when `foo` (Fig. 4) returns from eight or more recursive calls.

Sources of Imprecision. Our algorithm works on machine code; however, inferring the exact control flow graph of a binary program is not always possible due to indirect jumps [31]. We have implemented our algorithm on the x86 back-end available in the LLVM compiler. LLVM uses an intermediate representation that makes explicit the control flow of programs. We use this information, still available during the translation from LLVM's representation to x86's binaries, to build the δ-CFG. One source of imprecision left is due to functions called through pointers. We assume

that any function whose address has been taken can be the target of an indirect call. Other heuristics, such as Bacon's Rapid Type Analysis [2] could be used to refine our δ-CFG; however, we have not experimented with such approach yet.

Our algorithm incorporates some context sensitivity by propagating a stack of return points during the traversal of the δ-CFG. Whenever this traversal meets a FUN node, we push the address immediately after this node onto the stack. Upon finding a RET node, we continue our traversal at the address that is on the top of the stack. Notice that our algorithm is not fully context-sensitive, because we assume that a stack-less RET, within a function f, may divert execution to any program point that could succeed a call to f.

One last source of imprecision is the fact that our actual implementation of deduce works at the intermediate representation of the LLVM compilation infra-structure. Therefore, we can only analyze code that we can compile. However, this code often calls library functions, whose source code is not available. To handle these functions, we must assume a peak density for them. In Section 4 we have experimented with different assumptions. By assuming that each unknown function gives us no more than one return per five instructions we have observed no false positives.

A word on Asymptotic Complexity. The δ-CFG can be constructed in time linear on the size of a program's control flow graph. Function deduce has higher complexity. As stated in the beginning of this section, 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 [2]. Furthermore, multi-target branches are scarce and most of the branches that have 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.

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 lead to less available gadgets

ROP attacks require a fair number of available gadgets in the executable address space of the program under attack [28], 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. The most common technique is to ensure that return operations can only divert execution to addresses that follow call instructions [3, 9, 11, 15, 20, 24, 32]. Thus, the amount of gadgets that an attacker can use to build an attack is restricted to only those preceded by a CALL. Another strategy consists in replacing instructions that incidentally contain the code of indirect branches within their binary representation with alternative instructions with the same effect. G-Free [23] and Return-less Kernels [21] do it at compile time. 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. This happens because an specific threshold is strictly lower than an universal one. However, our experiments show that larger gadgets are rare.

If an attack uses gadgets of size 2, which is the average found in the publicly available exploits seen in Figure 2, then we would need no-op gadgets of size 4 to conceal an attack from a universal detection threshold. Figure 7 (a) illustrates this issue. In the figure, we interpose no-op gadgets of size 4 among effective gadgets of size 2; hence, obtaining a final density of 11 indirect jumps per 32 instructions. In contrast, if we use specific detection thresholds, then gadgets of size four are not enough to disguise an attack. For most of the benchmarks (16 out of 18), which have a specific threshold of 8, Figure 7 (b) shows that a no-op gadget must have a length greater than or equal to 6. Even for the two benchmarks whose specific threshold is 10 (*omnetpp* and *xalancbmk*), the minimum size of a no-op gadget is 5.

The need to use larger no-op gadgets implies stricter restrictions because the availability of no-op gadgets decreases as the length of these sequences grows. For sizes of no-op gadgets ranging from 3 to 25, Figure 8 shows the amount of these no-op gadgets found in the 9 vulnerable applications studied in Windows (note that Firefox has two different exploits). We use the four criteria of Section 2.2 to find no-op gadgets, but we have not tested their effectivity by crafting exploits with all of them. It was found that the number of no-op gadgets of size 4, able to circumvent a universal threshold of 12, is 75% greater than the amount of no-op gadgets of size 6, useful in overcoming a specific threshold of 8. Therefore, the use of specific thresholds drastically reduces the amount of artifacts available for building actual ROP attacks.

4.2 On the Accuracy of our Estimates

Aiming to measure the precision of our algorithm, we compared our estimates of maximum density for the SPEC CPU2006 suite of benchmarks with those observed when running these applications. The execution traces were monitored using the dynamic binary instrumentation framework Pin [22]. We have used the reference input of each program. Figure 9 shows that the peak values of the frequency of return instructions indicated by our algorithm are equal to or higher than the values recorded dynamically. This means that our algorithm is more conservative than the dynamic analysis, as it explores all possible program execution paths. In contrast, the control flow paths traversed during dynamic

[2] Source: http://www.strchr.com/x86_machine_code_statistics

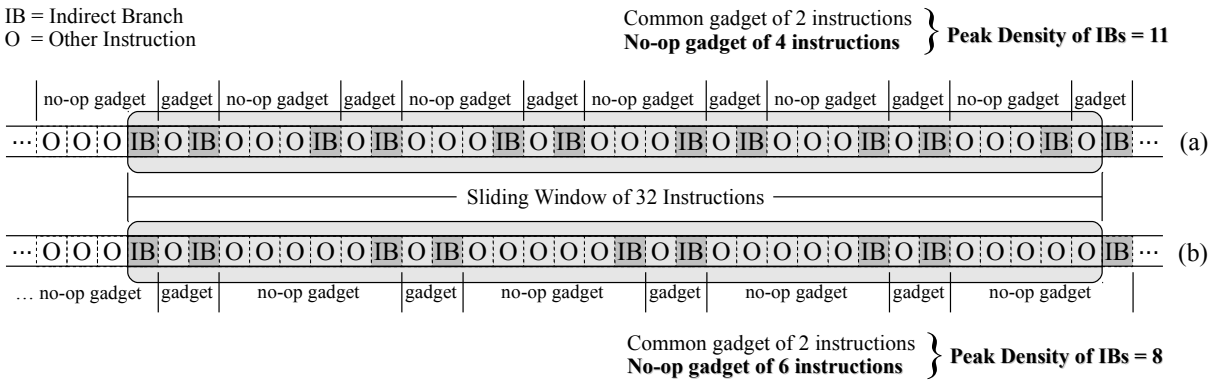

Figure 7. Minimal size of a no-op gadget needed to bypass universal (a) and specific (b) thresholds in a window of 32 instructions. Specific thresholds require longer gadgets, which are harder to find, as shown in Figure 8.

analysis depends on the input values sent to the application. As is often difficult to establish an input set able to exhaust all possible execution paths of a program, our static analysis exceeds dynamic monitoring in 64% of the benchmarks. In other cases, the two analyses presented the same accuracy.

The discrepancy between the values observed through static and dynamic analysis to *gcc* and *omnetpp* is due to tail recursive functions, which were not repeatedly activated by the reference input used in the dynamic analysis. Tail recursion results in a peak density of half the window size (K/2); hence, it gives us a density of 16/32 indirect branches. This corresponds to a repetition of the sequence of instructions *ADD ESP, #value* $\rightarrow$ *RET* that usually appears in function epilogues. The ADD is responsible for unstacking the function frame, while the RET returns the execution flow to the caller function.

Figure 9 also illustrates the impact of the maximum density assumed for the 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. These values were chosen since the average of the values obtained for the assessed functions is 1 return statement every 5 instructions. It was observed that this choice had no impact in 43% of the cases. For the remaining benchmarks, a maximum density of 1/5 would ensure that there are no false positives. That is, the estimated specific threshold is greater than or equal to the maximum density monitored dynamically.

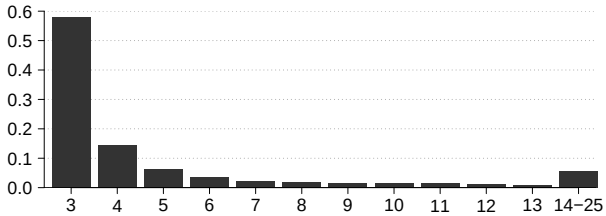

Figure 8. Percentage of potential no-op gadgets of sizes ranging from 3 to 25. There are 75% less potential no-op gadgets of size 6 (used to bypass the specific threshold) than 4 (used to bypass the universal threshold).

4.3 Performance of the inference algorithm

We also measured the performance of our algorithm to infer detection thresholds. Figure 10 shows the execution time of the benchmarks listed in Figure 9 as a function of the number of assembly instructions contained in each program. This experiment uses a window size of 32 instructions. The line in Figure 10, which approximates the points by a coefficient of determination of 0.91491 on a logarithmic scale, indicates that, given a fixed window, our algorithm is linear on the program size. The largest SPEC program, *xalancbmk*, with approximately 700 thousand instructions, was analyzed in

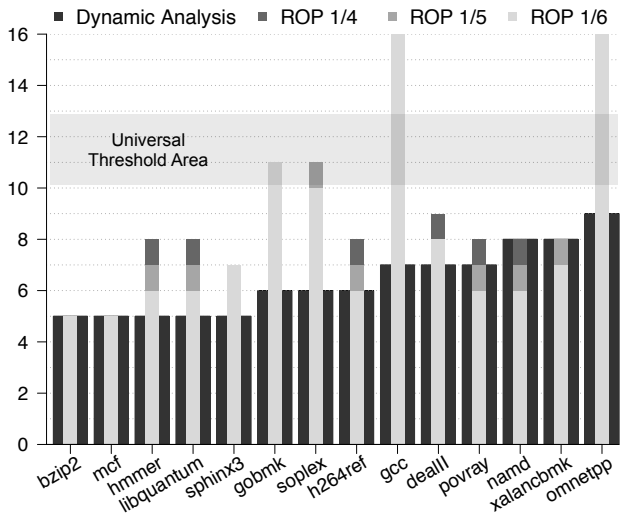

Figure 9. Specific threshold of return instructions for programs in the SPEC CPU2006 benchmark suite. Grey bars show specific thresholds found with the algorithm of Section 3, and black bars show values found dynamically. $1/4$ is the density that we have assumed for library functions, which we cannot analyze statically.

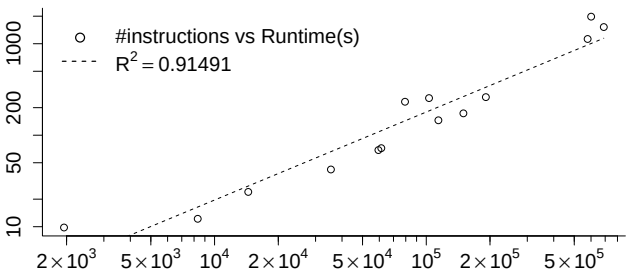

Figure 10. Program size vs runtime for K = 32. Points in the X axis are benchmarks from Figure 9, ordered by number of instructions. This figure indicates that even though our algorithm has a theoretical exponential complexity, its execution time has a linear behavior and is practically feasible.

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.

Runtime vs window size. We have run the benchmarks with different window sizes: $K = 2, 4, 8, 16, 32$. Despite the costly theoretical complexity of our algorithm, discussed in section 3.2, the exponential behavior has not surfaced in practice. Our runtime depends on the number of instructions in basic blocks. If a basic block contains more than K instructions, we will not touch its branch when starting the search at its first instructions. More than 50% of our basic blocks contain more than 5 instructions. Runtimes observed are roughly constant up to $K = 16$. With $K = 32$, the only exception observed was GCC, whose analysis time jumped from 613s with $K = 16$ to 1910s with $K = 32$.

5. Related Work

Since the first descriptions of ROP-based attacks, researchers have designed several different techniques to prevent or to detect such exploits. Memory protection mechanisms, such as Serebryany *et al.*'s AddressSanitizer [29] and Dhurjati *et al.*'s SAFECode [12], or data protection techniques such as LIFT [26], make it difficult for an attacker to corrupt memory to take control of a program. Furthermore, control flow integrity techniques, such as Kiriansky *et al.*'s notion of program shepherding [20], raise considerably the bar for a successful attack. However, as Carlini *et al.* have recently demonstrated [7], determined attackers can circumvent even protection mechanisms as strong as fully-precise static control flow integrity, as long as they can highjack control from the program. Therefore, we believe that currently there is no general and definitive technique to secure programs against ROPs. Nevertheless, the state-of-the-art defenses available today make it very difficult for attackers to exploit software effectively. In the rest of this section we discuss some of these protection techniques, focusing on those that rely on frequency monitoring to detect ROPs.

There are several previous work that propose ways to detect ROP attacks by inspecting the frequency of indirect branches in the instruction stream [8–10, 17, 19, 24, 33]. 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.* [8] fire warnings if they observe sequences of three return operations separated from each other by five or less instructions. Pappas *et al* [24] and Cheng *et al.* [9] use the Last Branch Record (LBR) registers to detect chains of gadgets. Carline *et al.* [6] and Goktas *et al.* [16] propose ways to bypass the LBR-based defense. However, we believe that an approach based on a sliding window would be harder to circumvent. For attackers to succeed against a sliding window, they must insert no-ops between gadgets. Just adding them at the end, or after 10 gadgets, like Carline does against the defenses proposed by Pappas *et al.* and Cheng *et al.* is not enough.

In a different direction, Yuan *et al.* [33] have proposed to count the frequency of instructions that are close to return operations. Instructions are assigned an amount of "suspicion" according to their proximity of a return instruction. The closer, the more suspicious it is. Once a given suspicion threshold is achieved, a warning is issued. Finally, there are approaches to detect ROP-based attacks that count the density of return instructions in the stack [17, 19]. Code that invokes gadgets usually only allocates space on the stack for return addresses. That is to say: the activation record of gadgets do not contain space for local variables, parameters, links to nesting functions, and other data that is usually present in the activation record of legitimate functions. Thus, in face of a ROP attack, the stack is likely to contain many words holding program addresses.

The work that we present in this paper is not a competitor of all these precious techniques. Instead, it complements them. They all rely on fixed thresholds to detect ROP exploits. By giving them the means to estimate the peak density of indirect branches, we help them to be more precise. Additionally, by showing that a sliding window provides an effective way to recognize ROP attacks, we open new avenues for the design and implementation of protection mechanisms that are simpler and more reliable than the current state-of-the-art detection approaches.

6. Conclusion

This paper has presented a technique that supports more precise detection of Return-Oriented Programming attacks. One of the most common detection mechanism is based on the monitoring of the frequency of indirect branches in the instruction stream of programs. We give this methods a way to compare this frequency against density thresholds that are specific per application. We have showed that our method is practical, scaling up to programs with more than 600 thousand assembly instructions. It also decreases effectively the

number of gadgets that are available for an exploit. The recent history has shown that, thus far, there is not foolproof method to prevent ROP-based exploits. Nevertheless, we believe that our specific detection thresholds raise significantly the bar for the successful execution of an attack.

Acknowledgment: This project is supported by the Brazilian Ministry of Science and Technology through CNPq, the Intel Corporation (the ISRA eCoSoC project) and FAPEMIG.

References

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

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341. ACM, 1996.

[3] P. Bania. Security mitigations for return-oriented programming attacks. Technical report, Kryptos Logic Research, 2010. URL https://kryptoslogic.com/download/ROP_Whitepaper.pdf.

[4] 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.

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

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

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Security Symposium*, pages 161–176. USENIX, 2015.

[8] 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.

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

[10] 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.

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

[12] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI*, pages 144–157. ACM, 2006.

[13] P. V. Eeckhoutte. Analyzing heap objects with mona.py, 2014. URL https://www.corelan.be/.

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

[15] I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks, 2012. URL https://ropguard.googlecode.com/svn/trunk/doc/ropguard.pdf.

[16] 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. In *Security Symposium*, pages 417–432. USENIX, 2014.

[17] Y. H. Han, D. S. Park, W. Jia, and S. S. Yeo. Detecting return oriented programming by examining positions of saved return addresses. In *LNEE*, pages 3:1–3:18. Springer, 2013.

[18] Intel. Ia-32 architecture software developers manual, 2006. Volume 2B: Instruction Set Reference, N-Z.

[19] J. Jiang, X. Jia, D. Feng, S. Zhang, and P. Liu. HyperCrop: a hypervisor-based countermeasure for return oriented programming. In *LNCS*, pages 46–62. Springer, 2011.

[20] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Security Symposium*, pages 191–206. USENIX, 2002.

[21] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *EuroSys*, pages 195–208. ACM, 2010.

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

[23] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, pages 49–58. ACM, 2010.

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

[25] D. Pfaff, S. Hack, and C. Hammer. Learning how to Prevent Return-Oriented Programming Efficiently. In F. Piessens, J. Caballero, and N. Bielova, editors, *ESSoS*, volume 8978 of *LNCS*, pages 68–85. Springer, 2015.

[26] 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*, pages 135–148. IEEE, 2006.

[27] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

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

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *ATC*, pages 28–28. USENIX, 2012.

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

[31] H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA*, pages 23–30. IEEE, 2000.

[32] 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.

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