Multilayer ROP Protection via Microarchitectural Units Available in Commodity Hardware

Mateus Tymburibá†
*Department of Computer Science (DECOM) CEFET-MG Belo Horizonte, Brazil mateustymbu@decom.cefetmg.br

Hugo Araújo de Sousa† and Fernando Magno Quintão Pereira†
†Department of Computer Science (DCC) UFMG Belo Horizonte, Brazil {mateustymbu, hugosousa, fernando}@dcc.ufmg.br

Abstract—This paper presents a multilayer protection approach to guard programs against Return-Oriented Programming (ROP) attacks. Upper layers validate most of a program’s control flow at a low computational cost; thus, not compromising runtime. Lower layers provide strong enforcement guarantees to handle more suspicious flows; thus, enhancing security. Our multilayer system combines techniques already described in the literature with verifications that we introduce in this paper. We argue that modern versions of x86 processors already provide the microarchitectural units necessary to implement our technique. We demonstrate the effectiveness of our multilayer protection on an extensive suite of benchmarks, which includes: SPEC CPU2006; the three most popular web browsers; 209 benchmarks distributed with LLVM and four well-known systems shown to be vulnerable to ROP exploits. Our experiments indicate that we can protect programs with almost no overhead in practice, alloying the good performance of lightweight security techniques with the high dependability of heavyweight approaches.

Index Terms—ROP, Architecture, Layers, RAS, LBR, CFI

I. INTRODUCTION

Return-Oriented Programming (ROP) is a form of software exploitation. It consists in chaining sequences of binary instructions of a program to force actions not originally intended by developers of that program [1]. Notorious malware such as Conficker [2], [3] and Duqu [4] use ROP. Additionally, the CVE database contains several examples of vulnerabilities recently exploited by ROP artifacts [5]–[7]. The damage that these exploits have caused has been extensively covered by the media; nevertheless, the research community tends to believe that current protection techniques can prevent most ROP attacks from succeeding [8], [9]. Even recent demonstrations of successful attacks are tailored to particular applications or protection measures, being unlikely to work in general [10]–[12]. And yet, ROP remains a threat. Performance, in this case, is to be blamed: effective protection techniques tend to slowdown programs beyond acceptable levels; hence, they are not used in practice.

The majority of anti-ROP protection techniques used in practice are based on Control Flow Integrity (CFI). Compilers such as clang or gcc implement different forms of CFI, but they all have in common the fact that they attempt to prevent unwanted program flows from happening. The first implementation of CFI incurred an overhead of 21% [13]. Said overhead has been since reduced, but it remains noticeable. As pointed by Burow et al. [8], effective CFI requires the validation of every type of indirect branch, including return instructions. Such verification has a cost. A quick survey over recent implementations of CFI reveals an overhead between 2 and 18%. This range includes SafeDispatch [14], T-VIP [15], VT/IFCC [16], vGuard [17], and VTint [18]. These approaches implement a version of CFI known as forward-only. More general techniques, that verify targets of return instructions, tend to be expensive because these instructions are more frequent than indirect calls and jmps. Techniques that guard return instructions, such as the original CFI [13], MoCFI [19], Lockdown [20], PathArmor [9] and PittyPat [21] present overheads ranging from 7 to 21%.

This observation: “effective CFI is computationally too expensive to be practical”, has been key for the demonstration of recent ROP-based exploits [22]–[24]. Additionally, two other factors contribute to make it difficult to design and implement a definitive CFI mechanism. First, the more constrained the implementation of CFI, the more false positives it raises, often rendering the execution of authentic programs an impractical task. Second, it has been demonstrated recently that even these more restrictive versions of CFI can be surpassed by determined attackers. Carlini et al. [23] have shown how to circumvent fully-precise CFI, and Veen et al. [24] have shown how to bypass context-sensitive CFI. Therefore, the design and implementation of a system that prevents ROP attacks from succeeding with efficiency and effectiveness remains an open problem.

Our Thesis. Efficiency and effectiveness do not need to coexist in a single protection technique. Rather, we can achieve them by combining existing approaches in layers. This view implies a change of perspective: instead of focusing on malicious execution streams as previous work does [9], [13]–[21], we first certify authentic streams. Upper layers cheaply certify most of the program flows and drastically reduce the number of tests executed in lower layers, which check eccentric codes via more expensive verifications. This novel multilayer system can be implemented using components that exist in current versions of the x86 architecture.

Our Contribution. As a proof of concept, we propose a
three-layered defense system to prevent code reuse attacks that divert return instructions. The first layer (Layer 1), explained in Section III-A, uses the hardware branch predictors to filter authentic execution flows. If the target of a return instruction is correctly predicted, then it is considered valid. This observation has already motivated previous work, which we revisit in Section V. Our Layer 1 filters out most of the valid program flows, because the branch predictor is very accurate, even when applied onto indirect branches, as we show in Section IV-C. Therefore, few branches slip from the first to the second layer, where more expensive checks happen. Layer 2, the subject of Section III-B, implements an original detection strategy that reduces the quantity of gadgets available to attackers. It extends Kiriansky et al.’s concept of call-preceded return address verification [25]. However, in addition to checking if a return address is preceded by a call instruction, we also verify if that call is valid. This new step was necessary: Carlini et al. [23] have already showed that Kiriansky’s method is not difficult to circumvent.

Our third layer, the subject of Section III-C, avoids false positives related to the verification of return addresses. The goal of this layer is to reduce the number of false alarms that would be raised by genuine control flows.

Our Results. We evaluate our ideas in Section IV. Our model imposes a small execution time overhead of 0.57% in a set of benchmarks composed by all 29 programs from SPEC CPU2006 [26] plus 209 benchmarks from the LLVM Test Suite [27]. The main reason for this reduced overhead is the high hit rate of branch prediction based on the Return Address Stack: 99%. The new check introduced in Layer 2 to validate call instructions that precede return addresses is also effective. It reduces by almost 35x the number of gadgets available to build an attack, when compared to a previous protection mechanism [25]. We analyzed the number of false positives that slip through our first two layers of protection. In this experiment, we enriched our benchmark suite with the three most popular desktop browsers and four vulnerable applications for which there are known ROP exploits. Only 3.14% of all return addresses are considered invalid in these benchmarks. We demonstrate that the cost to treat these false positives bears no impact in the execution time of programs.

A Design Based on Microarchitectural Units. We demonstrate the viability of our ideas using a prototype implemented in software, via Pin [28]; however, an important part of this work is to show that all these layers could be built at the hardware level, via microarchitectural units already in place in modern versions of x86 processors. Such units include the branch prediction machinery, branch recording structures, and executable space protection mechanisms. To support this claim, the description of every protection layer contains a discussion on how it can be implemented in today’s hardware.

The change of perspective mentioned in our Thesis fosters a very modular design. It is conceptually possible to replace the second layer of our model with other techniques that validate branches, as long as the necessary architectural units are in place. One candidate that fits well our second layer is Intel’s Control-flow Enforcement Technology (CET) [29] – a technique that also thwarts Jump-Oriented [30], [31] (JOP) and Call-Oriented [23], [32] (COP) Programming attacks. The basic principle is the same: the high-accuracy of the branch predictor leaves less jmps and calls to be verified by CET. And, similar to the Return Address Stack, the call and jmp predictors cannot be tampered with, as they are based on the previous execution of the instruction. When a call or jmp is executed for the first time, the branch predictor always misses and, therefore, forces the validation of that instruction in Layer 2. To demonstrate modularity, in Section IV, we show empirically the benefits of implementing Layer 2 using CET.

II. Return-Oriented Programming

ROPs are built around chains of gadgets. A gadget is a sequence of instructions that ends with an indirect branch – usually a return instruction. ROPs are used after the exploitation of several kinds of vulnerabilities [33], such as buffer-overflows, integer-overflows and uses-after-free. Once attackers exploit such vulnerabilities, they can control the execution flow of the vulnerable application overwriting its function stack with addresses of several gadgets, which will execute in sequence. After one gadget finishes its execution, its final indirect branch sends the program flow to the next gadget whose address has been stacked by the attacker.

These gadgets can be chained to bypass typical OS-level protections, such as Write-Execute. Once it is disabled, attackers can execute code from memory pages marked as data. For instance, Windows, depending on its configuration, lets a process disable Data Execution Prevention (DEP) via the routine SetProcessDEPPolicy. To invoke this function, an attacker can use sequences of gadgets to prepare its arguments. Tutorials and examples of ROP-based attacks abound in the Internet. For instance, in September 17th of 2011, Blake et al. [34] demonstrated how to hijack control of My MP3 Player v3.0 by disabling the DEP in Windows XP SP3. This particular attack used 10 gadgets. Figure 1 shows a snapshot of the stack during the actual exploitation of My MP3 Player.

The greatest challenge that attackers face when performing a ROP-based exploitation is to find gadgets free of side effects to avoid spoiling values previously prepared by the attackers themselves. Nevertheless, there are tools that find gadgets out of the binary representation of programs. As an example, Q [35], Mayhem [36] and Mona [37] give an attacker the means to automatically discover gadgets in a binary file.

III. Layers of protection

This paper shows that it is possible to combine different anti-ROP mechanisms in layers, so that lower-level layers (costly but effective) are only used when cheap higher-level layers have already filtered most of the authentic indirect branch flows. Figure 2 shows an abstract view of the multilayer protection mechanism that we advocate. Our key idea is to
certify the validity of indirect branches – the cornerstone of modern ROP exploits – in layers. The targets of the vast majority of these branches will be certified in our first layer, described in Section III-A. The few targets that we cannot validate at this level will undergo a further verification step in our second layer, to be explained in Section III-B. A third layer will filter this stream of branches even further, using the methodology described in Section III-C.

Figure 1. Stack of My MP3 Player during ROP-based attack.

Figure 2. Abstract overview of our multilayer protection mechanism.

If we cannot verify the target of a branch into these three layers, then multiple actions are possible. The most immediate of them would be to interrupt the program. However, a better compromise is to move the program into a safe mode; safety, in this case, can be achieved via sandboxing at the operating system level. Section III-D discusses this possibility; however, we emphasize that the implementation of sandboxing, a well-researched theme, is not part of the scope of this paper. Our idea is to filter out most of the indirect branch targets at a low computational cost. For the few that remain, we can afford the heavy price of stronger enforcement. As we show in Section IV-B, the proportion of targets that escape from one sieve to the next is tiny.

A. First level: Branch predictors

The first layer of protection relies on the branch predictors to validate targets of indirect branches. Modern computer architectures try to predict the target of indirect branch instructions to avoid stalls in the pipeline. For instance, current implementations of x86 use the so called Return Address Stack (RAS) to predict the address targeted by returns. Whenever a function is invoked through a call instruction, the address immediately following this invocation point is pushed onto the RAS. When a return operation executes, the predictor attempts to guess its target as the address on top of the RAS. Correct predictions designate legitimate program flows; thus, these branch targets should not be further verified. This modus operandi is simple, yet very effective and costless.

The use that we make of the RAS resembles a technique known as Shadow Stack [38]. A shadow stack does exactly what the RAS does: matches the target of returns against the addresses following call instructions, in a stack-like fashion. However, shadow stacks were conceived before the RAS became part of current x86 processors; hence, they could not benefit from this hardware. We claim that the shadow stack can be implemented at zero-overhead, at the hardware level, by capitalizing on the RAS infrastructure. Although we do not treat ROP attacks based on the other types of indirect branches (indirect calls and jmps) in this work, we show that their predictors have characteristics of efficiency and effectiveness similar to those exhibited by the RAS. This makes them perfect candidates to compose the first layer of protection against the abuse of other indirect branches. In Section IV-B1, we show the hit rate of these predictors, in addition to the hit rate reached by the RAS.

Implementation: The implementation of this protection layer only requires one extra comparison at the level of the branch predictor. This check can be fully implemented in hardware with no impact on the execution time of programs, since it does not directly impact the instructions pipeline.

Prototype: In Section IV we use hardware performance counters available in current x86 architectures to evaluate the hit rate of branch predictors. The defense mechanisms proposed in this paper do not depend on the availability of hardware counters – we use such devices for evaluation only.

B. Second level: valid call

Not every return address that is mispredicted indicates a ROP attack. Mispredictions happen due to overflows in the
RAS buffer, an incident caused by very long sequences of function invocations, or context switches between processes that compete for the CPU. If the target of a return is mispredicted, then we resort to further validation.

Our next verification step, the second layer of our system, relies on a simple idea: we check if the target of a return instruction is preceded by a call instruction, and if the target of the call instruction that precedes it lays in executable memory. The first verification is the cornerstone of modern implementations of control flow integrity (CFI) [16]. The second, however, is a new idea. We had to add further verification on top of traditional CFI, to mitigate shortcomings of this technique, already well-known in the literature [32], [39]. Figure 3 provides an overview of all these interactions. In the rest of this section we describe our two checks, using Figure 3 to guide the discussion.

a) The call-preceded constraint: If the target of a return instruction is not an address that succeeds a call instruction, then it is possible that such flow has been artificially created by an attacker. This observation has already been extensively explored in the literature [25]. Thus, in the absence of this condition, we immediately move the application to the third verification layer of our system (Step 1 in Fig. 3), where we apply stronger checks to certify the legitimacy of the flow. However, even when this condition is satisfied, we might still be facing a ROP attack. As an example, Göktaş et al. have demonstrated how to build ROP exploits under the call-preceded constraint [32]. Therefore, to prevent exploits such as the one carried out by Göktaş et al., we augment this constraint with an extra check, which we claim as an original contribution of this paper: the executable target constraint.

b) The Executable Target (XT) Constraint: Gadgets whose first instruction is preceded by call operations might emerge by chance in a large binary program, due to unintended instructions. These are instructions formed by unaligned sequences of bits within the binary code. We can show that the vast majority of such gadgets are invalid through a simple expedient: we verify if the call instruction targets an executable memory segment (Step 2 in Fig. 3). Because the executable area in the address space of a program is limited to its “text” segment, the chance that a misaligned call instruction targets this segment is very small, especially in 64-bit architectures.

Example 3.1: The Linux version of Chrome 59.0.3071.115 (64-bits) has a large amount of code, resulting in an equally long (163 MB) text segment. Yet, the chance that a random address matches an executable address is only 1 in 1011.

The different types of call instructions available in the x86 range from 2 to 7 bytes in size. Thus, when executing a return instruction, we look for the 7 bytes that precede the instruction located in the return address stored at the top of the stack. We then decode the instruction formed by these bytes and check if it corresponds to a call. If it is a call, then we calculate the target address of that instruction. To obtain this address, we differentiate the types of call by their opcodes and the byte “ModR/M” in the instructions, as defined in the manual of the x86 instruction set [40]. Finally, we verify if the target address of the call instruction has the no-execute (NX) bit enabled. Notice that processors already check the NX flag in the page-structure entry of any fetched instruction. Therefore, the microarchitectural structures that already exist for this purpose can be reused by our XT Constraint Checker.

Example 3.2: Figure 4 shows a concrete example of checks performed by layer 2 in a hypothetical 16-bits architecture.

c) XT Enforcement on Indirect Calls: The verification of the target address of indirect calls is not trivial, because when a return operation executes, the address used by the call that gave origin to it might no longer be available in either registers or memory. To deal with this shortcoming, we resort to a hardware structure called Last Branch Record (LBR). This feature, available in Intel processors, lets the CPU log the “from” and “to” address of each branch into specific registers. These registers form a ring-buffer, which is continuously overwritten. Thus, it traces only the most recent branches. We only record data about call in the LBR. In addition, we use this hardware structure in its stack mode. Thus, whenever a call executes, its data is stacked in the
LBR. When executing a `return`, the most recently stacked data is removed from the LBR. The main difference between the LBR stack and RAS is that LBR enforces data savings on context switches. This allows the LBR to treat some cases not covered by the RAS. Different processor models have different LBR sizes: Netburst and Merom have buffers with 4 cells; Nehalem up to Haswell have buffers with 16; Skylake features 32 positions and the Atom only 8. When faced with an indirect `call` preceding a return address, we scan the last filled entry of LBR looking for a match (Step 3 in Fig. 3). Notice that only mispredicted return addresses undergo this verification, because correctly predicted instructions have already been filtered out in the first layer of our system.

Implementation: The XT constraint can be enforced in processors featuring Return Address Stack and Last Branch Record capabilities. For instance, current Intel/AMD machines feature an extension called execute Disable: XD in Intel, and NX (of no-execute) in AMD. This mechanism marks virtual pages with a bit denoting executable memory. This bit is consulted at zero overhead. The same is expected for a hardware comparison with the value stored at the top of LBR.

Prototype: In Section IV we simulate the implementation of this layer in software via a Pintool, i.e., a tool created with Pin. The Pintool checks if the targets of direct `call` instructions that precede return addresses lay in executable memory. It also simulates the LBR operation and matches the top of LBR with indirect `call` instructions that precede return addresses.

C. Third Level: Treatment of False Positives

It is possible that legitimate return addresses pass unfiltered through all our previous sieves because some authentic program flows contain `return` instructions targeting addresses not preceded by any `call` instruction. The typical situations where this event happens have been cataloged by previous work [41]–[43]: lazy binding of dynamically shared libraries and signal handling in Unix-based systems.

a) False Positives due to Lazy-Binding: Lazy binding optimizes the loading time of a program. It consists in delaying the resolution of the initial address of functions belonging to shared libraries until these functions are called for the first time. Thus, the dynamic linker does not waste time solving the address of functions that are never called. In Linux, for instance, false positives can occur because a return instruction is “improperly” used by the OS during lazy binding. On Linux systems, lazy binding is enforced by a combination of functions that are part of the `linux-ld.so` dynamic linker library. After resolving the address of the target function, the OS transfers the execution flow to the code of that function through a return instruction. False positives can occur because the execution flow is diverted to the first instruction of the requested function and there is no guarantee that a valid `call` exists immediately before that first instruction.

b) False Positives due to Signal Handling: In Unix-based systems, signals alert a process about exceptional events such as the decoding of illegal instructions, the execution of invalid arithmetic operations and unauthorized accesses of memory segments. Upon receiving a signal, the OS stacks the address of the current instruction of the paralyzed process and transfers the execution flow to the code responsible for signal handling, without executing a `call` instruction. When the signal handler returns, it removes the address stacked by the OS and transfers the execution back to the normal program flow through a `return` instruction. A process can be stopped by the OS at any instruction; hence, the address just removed can refer to anywhere. Therefore, the return address stacked by the OS may not be preceded by a valid `call` operation – a fact that might give us false positives.

Detection of False Positives. The two categories of false positives discussed in this section can be avoided at the compiler level or at the hardware level. At the hardware level, false positives can be identified via extra checks, such as those proposed by Xia et al. in the CFIMon system [41]. CFIMon records performance samples using the Branch Trace Store (BTS) mechanism. The BTS makes it possible to monitor executed branches and correlate the traces with sets of valid targets obtained via static analysis. To avoid triggering false alarms, Xia et al.’s monitor is informed by the operating system whenever the signal handler is invoked.

At the compiler level, we can change specific `return` statements by a `pop rg; jmp rg` sequence. The return address `r` is dumped in register `rg`. Then the control flow is transferred to `r` via an indirect `jmp`. Zhang and Sekar propose a similar solution to circumvent the idiosyncrasy related to using a return instruction for lazy symbol resolution [43]. In our case, this modification must be done in the dynamic linker and in the signal handler. As Section IV-C will show, this replacement does not incur into any overhead of statistical significance on a modern x86 machine.

Implementation: The strategy used to handle false positives depends on the implementation of Layer 2. In Section IV we shall compare two implementations: one based on the validation techniques seen in Section III-B, and another based on Control-Flow Enforcement Technology (CET). The implementation discussed in Section III-B let us avoid false positives at the compilation level. In this case, no runtime check is required. On the other hand, the CET-like implementation of Layer 2 requires runtime validation at the level of Layer 3. In this case, we can reuse either the techniques employed in the CFIMon tool [41], or the rules discussed by Qiu et al. [44, Sec.V]. Both approaches require circuitry not yet in place in current x86 processors.

Prototype: In Section IV-C4 we shall implement the compiler-level enforcement of Layer 3 via manual replacement of instructions in assembly codes. To experiment with CET, we shall simulate it in Pin, following a description publicly available of the Intel implementation [29].

1If false positives are avoided at the compilation level, then our Layer 3 becomes inextinct at runtime. We preferred to leave it in our model, to emphasize that it exists to handle false positives produced at Layer 2.
D. Handling Code that Slips through the Third Layer

If the target of a return slips through our three sieves, then, with strong probability, the program flow might have been diverted by a malicious user. There are different ways to handle attacks. Solutions can go from simply terminating the program to sandboxing it. Sandboxes are implemented through emulated or virtualized runtime environments [45]. They often detect malicious behavior by monitoring system calls [46]. Previous reports indicate that the computational cost of emulating a program ranges from 2 to 40%, depending on implementation choices and the amount of system calls executed by the monitored application [47]. Nevertheless, as we demonstrate in the next section, very few programs under normal execution would pay this cost, in case sandboxing were the solution of choice to treat suspicious flows.

IV. Evaluation

In this Section, we investigate three research questions:

• RQ1: How effective is our method to reduce the number of gadgets available for an exploit?
• RQ2: What is the proportion of targets of indirect branches that are filtered at each protection layer?
• RQ3: What is the overhead of our approach?

We answer these questions empirically. In Section IV-A we use an assortment of dynamic analysis tools to count how many gadgets our system leaves available to an attacker. In Section IV-B we use Linux Perf [48] to count how many branches slip from Layer 1 to Layer 2, and Pin to measure leaks from Layer 2 to Layer 3. Finally, in Section IV-C we evaluate the overhead of our approach using an analytical argument supported with data from Pin. We also resort to manual interventions in assembly programs to evaluate the overhead of a compiler-based implementation of Layer 3. The tools used in the evaluation are publicly available on our web server (http://cuda.dcc.ufmg.br/multilayer-rop-protection/).

Benchmarks. We use several benchmark suites in this paper, because each set fits different purposes. To probe gadget reduction and false positives, we use 4 vulnerable applications for which we could reproduce 5 public ROP exploits (exploits 17634, 17672, 17974, 12189, 13834 at www.exploit-db.com) for which we could reproduce 5 public ROP exploits (false negatives) using gadgets that are not disclosed by manufacturers or could not be implemented in the simulators. For example, through preliminary experiments, we know that the hit rate of RAS is much higher than the hit rate of a conventional stack. Possibly, this is due to the adoption of optimization techniques like speculative RAS management [53]. This same difference occurs in the simulation of the memory system, as demonstrated in a comparison of hardware simulators [54].

A. RQ1 – Gadget Reduction: a false negatives assessment

In the context of this work, false negatives happen whenever we fail to detect a ROP-based attack. To assess the likelihood of false negatives, we follow Carlini et al.’s [23] methodology—they adopt the number of available gadgets as a metric of security. A good ROP defense mechanism leaves few gadgets still accessible to an attacker. In our case, attackers can build successful ROP exploits (false negatives) using gadgets that have any of the following properties:

1) the gadget does not trigger a branch misprediction;
2) the gadget is preceded by an indirect call instruction that is at the top of LBR;
3) the gadget is preceded by a direct call instruction that targets an executable memory region;
4) the gadget passes the false positive checks.

Finding gadgets with property (1) is impracticable because branch predictors record data about instructions already executed by the program. Thus, the first subverted branch instruction causes a misprediction, unless the attacker can reuse a program flow already executed. The impossibility of finding gadgets with property (1) is corroborated by previous studies [55]-[57]. It is also highly improbable to abuse RAS (property 1) or LBR (property 2) because for each gadget it would be necessary to force the execution of a call instruction positioned immediately before the return address. Even more: this call operation must be the most recent instruction of this type to be executed. This would only be possible in a scenario where useful gadgets for the attack, terminated with an indirect call, are sequentially placed in an executable address space so that an attacker can merge them with gadgets ending in returns. This event is equivalent to accidentally having attack code ready within the application itself. Finally, notice that our multi-layer approach does not introduce new gadgets in the process of eliminating those

Before reaching this decision, we considered two alternatives: using the gem5 hardware simulator [51], or doing instrumentation via LLVM [27]. The latter we quickly ruled out, because that approach would only let us analyze programs whose source code is available – libraries would remain terra incognita. Choosing Pin over gem5 was a harder decision: our initial implementation was based on the latter. However, its overhead is impractical for the experiments that we report. The most recent study that we know announced 50-500 KIPS [52]. Our experiments with SPEC CPU2006 alone already give us 48 Terabytes of instructions. Furthermore, we believe that a prototype implemented in a cycle-accurate simulator would present distorted results due to optimizations in hardware that are not disclosed by manufacturers or could not be implemented in the simulators. For example, through preliminary experiments, we know that the hit rate of RAS is much higher than the hit rate of a conventional stack. Possibly, this is due to the adoption of optimization techniques like speculative RAS management [53]. This same difference occurs in the simulation of the memory system, as demonstrated in a comparison of hardware simulators [54].

A note on the choice of tools. We perform all the experiments that require interventions in hardware via simulation in Pin.
already known—a shortcoming present in techniques that shrink source code, as pointed by Brown and Pande [58].

In principle, a user can combine ROP with an attack similar to Spectre against RAS to change the execution flow of a process from another user. However, even if a malicious user succeeds in polluting RAS with fake return addresses, the context switch between the malicious program and the vulnerable process would need to happen exactly between the original call and the vulnerable return. The inability of an attacker to interfere in those switches yields this scenario improbable. Finally, item (4) filters the cases enumerated in Section III-C, which otherwise would be misclassified as attacks. Using gadgets with those characteristics is virtually impossible, as they involve Operating System code. Thus, we shall count how many gadgets meet property (3) by chance (i.e., direct calls whose random target is executable).

**Measurement methodology.** To count gadgets we follow a methodology proposed by Bowne [59]. Firstly, we pause the program of interest at a particular execution point (right before hijacking the control flow, for instance) with help of Immunity Debugger. Secondly, we use our own crafted version of a Mona plug-in to scan the whole process memory and list all available call instructions. This list lets us identify which gadgets are preceded by a call operation, even when this instruction is the result of an unaligned memory access. Finally, we use Pin to simulate the call validation mechanism of Section III-B. Pin lets us inspect each instruction upon its execution. We use it to verify (i) that the destination address of a call that precedes a candidate gadget is executable and (ii) to simulate the presence of the LBR.

**Analysis of Results.** Figure 5 shows the results of this experiment on seven large applications—four of which are vulnerable to known ROP exploits. The number of gadgets left by the executable target constraint (XT) is 0.06% of the total number of gadgets in the binary. To give the reader some perspective on this number, this constraint leaves 34.94 times less gadgets than the call-preceded constraint when applied onto the same suite. The call-preceded constraint is central to many previous defense mechanisms [25], [41], [43], [60]–[62].

In practice, the reduction that our new constraint achieves makes it very difficult to build effective ROP attacks. We support this statement with a statistical argument first used by Schwartz et al. [35]. Their tool, Q, usually requires binaries with at least 100KB to be able to execute calls to any function in libc. Similarly, it requires binaries of at least 20KB to invoke linked functions and update arbitrary locations. In other words, the number of gadgets expected to be found within 20KB of code is enough to carry out an attack. The executable target constraint reduces the density of gadgets in the binary. Pragmatically, this is equivalent to reducing the size of the binary available to the attacker. Figure 5 shows the size of the binary representation of each application. The last column of Figure 5 shows the “apparent” size of those binaries, in terms of number of gadgets, that Q would see. Notice that this new size is given in KBs, not in MBs. The largest apparent size, found in Internet Explorer 8, is over 5x smaller than the minimum size that Q requires to perform a successful exploit.

**B. RQ2 – Filtering statistics**

The multilayer approach that we advocate in this paper is interesting if a large volume of indirect branches is filtered across successive layers. In this section, we show that the flow of indirect branches to be validated is drastically reduced by going through Layers 1 and 2 of our model. We test two different implementations of Layer 2: in Section IV-B2 we use the executable target constraint (XT), and in Section IV-B3 we use Intel’s Control-flow Enforcement Technology (CET).

1) From Layer 1 to Layer 2: Indirect branch prediction: The Layer 1 of our model, described in Section III-A, uses branch prediction to validate program flows. The greater the number of indirect branches that can be correctly predicted, the smaller the number of cases that reach Layer 2. In this section we shall count this hit rate.

**Measurement methodology.** To analyze the effectiveness of modern indirect branch predictors, we count the actual number of prediction misses. To this end, we use Hardware Performance Counters (HPCs) available in current x86 processors. In the experiments, we calculate the percentage of indirect branches correctly predicted by monitoring the number of misses and the total amount of branches executed. x86 implements different prediction strategies for different categories of branches. Thus, we had to capture the performance counters for each type of indirect branch. We count the taken speculative and retired returns, indirect calls, and indirect jmps. We also measure the number of mispredictions for the same instructions. To produce summarized results, all the average values presented were weighted on the number of instructions executed by the benchmarks: programs with more instructions contribute proportionally more towards the final average.

**Analysis of Results.** Figure 6 reports the accuracy of the branch predictor for the programs in the SPEC CPU2006 collection, and in the LLVM test suite. We have grouped the results for the LLVM test suite into a single bar, called LLVM. This procedure will be also used in the next charts that we present in this paper. The benchmark that executed the lowest number of instructions was gobmk, with 6.8E+10

<table>
<thead>
<tr>
<th>Application</th>
<th>#Gadgets</th>
<th>CPG</th>
<th>VCPG</th>
<th>BS (MB)</th>
<th>AS (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP 6.0</td>
<td>269,318</td>
<td>2.77%</td>
<td>0.04%</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>Free CD to MP3 Converter 3.1</td>
<td>346,415</td>
<td>2.63%</td>
<td>0.09%</td>
<td>1.08</td>
<td>0.99</td>
</tr>
<tr>
<td>Internet Explorer 8.0.6001.18702</td>
<td>624,638</td>
<td>2.64%</td>
<td>0.14%</td>
<td>2.45</td>
<td>3.59</td>
</tr>
<tr>
<td>Mozilla Firefox 3.6.16</td>
<td>1,444,959</td>
<td>2.61%</td>
<td>0.10%</td>
<td>0.87</td>
<td>0.88</td>
</tr>
<tr>
<td>Mozilla Firefox Quantum 62.0.3</td>
<td>1,779,989</td>
<td>2.22%</td>
<td>0.05%</td>
<td>1.31</td>
<td>0.73</td>
</tr>
<tr>
<td>Google Chrome 69.0.3497.100</td>
<td>3,098,708</td>
<td>2.08%</td>
<td>0.02%</td>
<td>0.39</td>
<td>0.09</td>
</tr>
<tr>
<td>Internet Explorer 11.64.16299.0</td>
<td>3,332,556</td>
<td>1.05%</td>
<td>0.04%</td>
<td>0.79</td>
<td>0.29</td>
</tr>
</tbody>
</table>

Figure 5. Gadget Reduction on four applications vulnerable to known ROP exploits (top) and on the three most popular desktop browsers in November 2018 (bottom) [49]. CPG: call-preceded gadgets (as a percentage of #Gadgets). VCPG: valid call-preceded gadgets. Averages are CPG = 1.94% and VCPG = 0.06%. BS binary size, in MBs, and AS apparent size, in KBs, after gadget reduction.
operations. The largest number of instructions was executed by calculix: 6.1E+12. In total, we observed 4.8E+13 instructions when evaluating SPEC CPU2006. The dynamic predictor succeeds in 97.72% of the indirect branches executed. This number is the average of all the averages, weighted by the number of instructions in each benchmark. In the specific case of return addresses, the return address stack (RAS) achieved the accuracy of 99.03%. The prediction of targets of indirect jmps and calls achieved respectively a hit rate of 92.81% and 89.32%. These values corroborate the hypothesis that dynamic predictors of indirect branches are an effective way to filter out the cases to be monitored. If we consider all the instructions executed, on average one in 100,000 requires action of Layer 2. This is the main reason for the low overhead imposed by our model, as we shall discuss in Section IV-C.

2) From Layer 2 to Layer 3 - Return Address Validation: Our Layer 2 applies the Executable Target Constraint described in Section III-B to validate returns that are mispredicted. This layer is effective as long as most of the return addresses are preceded by valid call instructions. In this section we quantify them.

Measurement methodology. We use the prototype described in Section III-B to simulate the Executable Target Constraint. We also simulate the Instruction Translation Lookaside Buffer (iTLB). The idea is to check the execution permission of the target of a direct call instruction in the same frequency that a page walk actually happens on a real processor. Using this strategy, we establish an upper bound that better approximates the execution time of a solution implemented in hardware. We have used hardware performance counters to estimate the iTLB miss rate observed when executing SPEC CPU2006 benchmarks. We use this estimate, plus a counter in the Pin-tool, to mimic the hardware behavior adjusting the frequency that permission checks should actually execute. Our machine’s iTLB is parameterized in the following way: L1 instruction TLB for 2M/4M pages is fully associative with 8 entries; L1 instruction TLB for 4K pages is 4-way associative with 64 entries; and L2 TLB (instructions and data) for 4K pages is 4-way associative with 512 entries.

Analysis of Results. Using CET’s shadow stack, we observed that 1.44% of the branches slip from Layer 2 to Layer 3. The target execution constraint, as seen in Figure 7, gives 3.44%. As recently pointed by Qiu et al. [44, Sec.V.C], a CET-like shadow stack has more cases that yield false positives than those we have enumerated for the executable target constraint. Nevertheless, in our experiments, CET has been able to validate more return instructions than XT. We speculate that this result is due to the fact that Pin does not instrument privileged code and some of the known cases of false positives are related to system calls. This said, our results indicate that both strategies produce a comparably low false positive rate. On the other hand, a shadow stack is more costly, from an implementation perspective, as we discuss next.

<table>
<thead>
<tr>
<th>return addresses preceded by Valid CALLs</th>
<th>Direct</th>
<th>69.84%</th>
<th>96.66%</th>
</tr>
</thead>
<tbody>
<tr>
<td>indirect</td>
<td>27.01%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return addresses preceded by Invalid CALLs</td>
<td>Direct</td>
<td>0.17%</td>
<td></td>
</tr>
<tr>
<td>indirect</td>
<td>1.54%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return addresses not preceded by CALLs</td>
<td>1.44%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. Validations that slip from Layer 2 to Layer 3. 96.86% of return addresses are validated at Layer 2; 3.14% will slip to Layer 3.
A shadow stack has three shortcomings, which do not exist in the executable target constraint. First, XT has lower impact on the instruction pipeline since validations happen in parallel with instruction processing. To manipulate the shadow stack, on the other hand, CET adds work onto call and return instructions. It also adds work onto store operations. Such instructions must be verified to prevent them from tampering with the shadow stack, as pointed out by Sinha et al. [63]. Second, the executable target constraint only accesses memory when executing return instructions; and only due to iTLB misses, to check if the destination of a call is executable. In our experiments with SPEC CPU2006 benchmarks, the iTLB miss rate was only 0.005%. In CET, the shadow stack is implemented in memory to prevent overflows. Therefore, each call and return operation will require a memory access. Finally, our schema does not require the creation of mechanisms to protect storage structures, such as the RAS or the LBR, because they already exist in hardware and are inaccessible to the programmer. In the case of CET, since the shadow stack is located in memory, it is necessary to create and manage a protection flag against unauthorized writing. This said, we emphasize that our multilayer approach is equally effective to reduce the overhead imposed on either CET or XT. This overhead is the subject of the next section.

C. RQ3 – Overhead

Our first layer of protection can be implemented at zero-overhead, because it relies on hardware already in place in modern computer architectures. Therefore, in this section we focus on the overhead of Layers 2 and 3. Like in Section IV-B, we analyze two implementations of Layer 2, based on call-validation and control-flow enforcement technology.

1) The Overhead of Layer 2 Without Layer 1 - Executable Target Constraint: Return address validation is performed by the combination of two constraints, as mentioned in Section III-B: the call-preceded and the executable target. In this section we shall measure the impact of enforcing them. Measurement Methodology. To estimate the overhead of the second layer, we will proceed in two steps. First, we shall approximate the cost of implementing the checks of Section III-B for every return instruction processed during the execution of a program. Then, we shall reduce this cost proportionally to the quantity of such instructions that slip from the first to the second layer of our approach. As benchmarks, we use SPEC and the LLVM test suite. Our prototype is implemented in Pin, which already imposes a heavy overhead on the code that it emulates. Thus, to estimate the overhead of Layer 2 when running at the hardware level, we need to discount Pin’s runtime cost. To carry out this discount, we also run our applications through Pin without the verifications used to implement Layer 2. Let $ExTime_{pruned}$ be our baseline time. This is the runtime of the simplified Pintool. Let $ExTime_{complete}$ be the runtime of the Pintool with the implementation of Layer 2. The estimated overhead is the difference $ExTime_{complete} - ExTime_{pruned}$.

Analysis of Results. Figure 8 shows the result of this analysis. The numbers in Figure 8 show a weighted average upper bound of 50.64% on the overhead that we can expect. We are emulating in software all the verifications that, in production, would be implemented in hardware. For instance, our Pintool produces sequences with several operations to check if the instruction that precedes the target of a return is a call. Even more instructions are issued by the Pintool to verify if the call targets an executable memory area because our prototype needs to track, through data-structures provided by Pin, which memory sections of a process image are executable. In hardware, these verifications would happen in parallel with the instruction pipeline; hence, we speculate that it would be orders of magnitude lower. Nevertheless, as we shall see in Section IV-C4, this overhead shall fade away due to Layer 1 – this is the beauty of the multilayer approach.

2) The Overhead of Layer 2 Without Layer 1 - Control-Flow Enforcement Technology: We also measured the overhead of Layer 2 considering a CET-like shadow stack in-place. This new experiment uses the infrastructure seen in Section IV-B3. Measurement Methodology. Like in Section IV-C1, we also discount Pin’s runtime structure to measure overhead. To this end, we created a pruned version of the Pintool. This baseline has all the code of the final prototype: shadow stack manipulation, counters operation, etc. However, it does not perform the comparison between the return address and the address in the top of the shadow stack. We subtract its execution time from the value observed for the complete Pintool.

Analysis of Results. Results are presented in Figure 9. Our shadow stack implementation slowed down the benchmarks in 21.65% on average. We cannot establish a rigid comparison between the overhead observed in the shadow stack and in our Executable Target Constraint, because software implementations in Pin are considerably different than hardware implementations. However, these values indicate that both strategies present a comparable cost. When applied to a tiny portion of returns, as shown in Section IV-B1, both strategies will cause an insignificant overhead, as we show in the next section.

3) The Overhead of Layer 2 and Layer 1: The overhead presented thus far, be it through call validation, be it through Control-flow Enforcement Technology is high. However, only
mispredicted indirect branches need to be verified at Layer 2. In this section we estimate this new overhead, to conclude that Layer 1 brings a fundamental improvement onto any of these two validation mechanisms.

**Measurement Methodology.** We use the following formula to compute the overhead for each application/benchmark:

$$Overhead_{L1+L2} = RET_{mp} \times RET_{vo}$$

$RET_{mp}$ is the number of return addresses that are mispredicted at Layer 1; hence, falling into Layer 2. $RET_{vo}$ is the verification overhead per return instruction. This metric is calculated as follows:

$$RET_{vo} = \frac{ExTime_{complete} - ExTime_{pruned}}{RET_N}$$

$ExTime_{complete}$ is the execution time taken by an application monitored by the complete Pintool, as seen in Section IV-C1. This Pintool has every data structure used in the implementation of Layer 2 (LBR structure, verification of the instruction that precedes a return address, verification of a LBR match for indirect calls, calculation of targets for direct calls, verification of permissions of targets of direct calls and manipulation of counters). $ExTime_{pruned}$ is the execution time taken by an application monitored by a Pintool that does not have the verification code. This Pintool has: LBR structure and manipulation of counters. Finally, $RET_N$ is the number of return operations executed by each application.

**Analysis of Results.** Figure 10 presents the results for the multilayer system based on the Executable Target Constraint. Most of the benchmarks present an overhead close to 0%. The weighted average overhead is 0.57%.

The cumulative overhead of a CET-like shadow stack is even lower, due to the lower overhead of our shadow stack implementation (Section IV-C2). Figure 11 shows this result. The weighted average in this case is just 0.02%. The numbers outlined in Figure 10 and 11 reinforce our thesis that a multilayer system allows the combination of costly anti-ROP protections. Notice that our overhead results are an upper bound limit: they are based in software simulations of hardware solutions. Therefore, we can expect much lower values in a hardware implementation.

4) The **Overhead of Layer 3**: As mentioned in Section III-C, false positives can be avoided either at the hardware level, or at the compiler level. If Layer 2 is implemented via the call validation strategy of Section III-B, then a solution at the compilation level is possible. The CET-based implementation of Layer 2 requires hardware interventions. At the end of this section, we discuss the overhead of a hardware-based implementation of Layer 3.

**Measurement methodology.** As discussed in Section III-C, a compiler-based solution to eliminate false positives consists in replacing the return instruction by a sequence of pop reg and jmp reg. This replacement is necessary only for functions that might be invoked by means other than the execution of a call instruction. To estimate the overhead of this replacement, we have performed it manually on every function of the twelve programs of the Shootout collection, available in the LLVM test suite\(^2\). We had to perform 52 replacements. Tail call optimization was not allowed. Figure 12 shows an example of code replacement.

**Analysis of Results.** Figure 13 shows the result of this experiment. All the benchmarks run for more than one second; Hash (hsh) runs for 35 seconds — the longest runtime. Every benchmark performs thousands of function calls. Each box

---

\(^2\)Shootout contains the smallest benchmarks in the LLVM test suite that are not unit tests. We chose small benchmarks because we had to perform the instruction replacement manually, directly in the assembly representation. The sole difficulty in this task is finding a free register reg.
shows the runtime of the modified program divided by the runtime of the original program. The \( t \)-test applied on the two populations has never given us a \( p \)-score lower than 0.1. Thus, it is statistically difficult to distinguish original and modified binaries by just observing their runtimes.

**Figure 13.** Execution overhead of replacing `return` with the `pop` and `jmp` sequence in programs of the LLVM Shootout suite. Each box contains eight executions. The higher the average, the higher the overhead.

**Hardware** It is not straightforward to estimate the overhead of false positive detection, at the hardware level, when used in tandem with our system. Considering the filtering statistics of Layers 1 and 2 combined (Section IV-B), we can expect that the third layer will be activated for less than 0.03% of the indirect branches in a program. Thus, verification à la CFIMon does not incur a cost that we can measure reliably. Nevertheless, we can rely on previous work to put an upper bound on the expected overhead of a hardware implementation of Layer 3. CFIMon authors claim an overhead of about 6.1% to execute all their solution and point out that 86% of this cost is due only to the code for recording executed branches, carried out through their “pure BTS” implementation. Therefore, the rest of CFIMon, including the tracking code and the treatment of false positives, accounts for an overhead of no more than 0.9%. Notice that every branch target is sanitized in CFIMon. In our case, this overhead will be imposed upon a negligible number of indirect branches, as we explained in Section IV-B. Therefore, we expect a much lower cost. More importantly, this verification does not impose any overhead upon the branches that do not reach the third layer.

**Figure 12.** To carry out the experiment described in this section, we compile a program such as (a) into its assembly version (b). Then, we replace, manually, occurrences of `rec` with the two-instruction sequence seen in (c).

(a) void rec(int i) (b) rec_ret: (c) rec_pop_jmp:

```c
{ 
  if(i>0) {
    rec(i-1);
  }
  return;
}
rec_ret:
.LFB0:
  cmp [esp+4], 0
  .LFB0:
  cmp [esp+4], 0
  jle .L4
  jle .L4
  mov eax, [esp+4]
  mov eax, [esp+4]
  sub eax, 1
  sub eax, 1
  push eax
  push eax
  call rec_ret
  add esp, 4
  add esp, 4
  .L4:
  .L4:
  ret
  pop edx
  jmp edx
```

V. RELATED WORK

a) On the techniques reused in our multi-layer system:

The idea of combining different defense mechanisms in layers, so that low-overhead protections are used to gradually filter safe control-flows, is an original contribution of this paper. Nevertheless, except for the verification of executable targets, discussed in Section III-B, our defense layers reuse techniques presented in previous work. Our rational when choosing which technique to apply on each layer was based on two tenets: (i) upper layers – which are applied first – should have lower computational overhead; and (ii) work done in upper layers should not be redone in lower layers.

Our uppermost layer, which certifies targets of indirect branches via the branch predictor, has already been described as early as 2007, by Shi and Lee [57], right after Hovav Shacham [1] introduced the concept of Return-Oriented Programming. More recent works have revisited this concept in an attempt to block ROP attacks [64]–[66]. Other proposals use the number of mispredictions of indirect branches as an attack indication [67], [68]. We adopt the inverse perspective: correct predictions of indirect branches indicate authentic executions. Execution flows that we cannot validate using branch prediction flow into lower layers of our system. The previous literature doesn’t talk about this multi-layer approach.

The idea of checking if the address before the target of a `return` is a `call` instruction has been independently described by previous works. Carlini et al. [23] provide a comprehensive overview of the related literature. The fact that Carlini et al. have been able to circumvent this kind of defense led us to conceive the Executable Target Constraint that we described in Section III-B. We designed this test to be easily implemented, once the call-validation mechanism is in place.

The technique that we use to filter out false positives, discussed in Section III-C was first suggested by Zhang and Sekar [43]. We adopted the same strategy defined in that work because it is simple and computationally cheap. However, were it used without our upper layers, then it would be too costly to be practical. Finally, sandboxing, one of the alternatives described in Section III-D to protect an application, when all the upper layers fail to certify an indirect branch, is well-known in the systems community.

b) How our overhead compares to previous works:

The ultimate goal of this paper is to provide protection against ROP attacks at a low computational cost. The literature contains much previous work with similar purpose. For a comparison, we mention the overhead of eight recent systems [9], [19], [21], [69]–[73] that enforce some sort of Control Flow Integrity policy. We report overheads instead of reproducing them ourselves because these tools have existed as research artifacts only and they aren’t easily available today. Some of these systems were only tested via software-based prototypes, although they are meant to be implemented in hardware; hence, some numbers that we shall mention are estimates from their authors. SCRAP’s authors report the lowest overhead [71], between 1% and 2%; however, they use
a strategy based on controlling the frequency of small gadgets, which has already been overcome in subsequent work [74–76]. Kayaalp et al. report a 2% overhead for BR [70] – same value reported by Lucas et al. for HAFIX [69] and for Zipper Stack [72]. The same group had, before, proposed MoCFL, with an overhead of 7% [19]. Veen et al. report an overhead of 8.4% for PathArmor [9]. PittyPat’s original presentation reports an overhead of 12.73% [21]. Finally, Lockdown’s authors have measured a runtime slowdown of 19% [73]. Most of these techniques require some support from the hardware. For instance, HAFIX requires changes in the target architecture’s instruction set. Software-only approaches, such as PittyPat, tend to present higher overhead, in addition of requiring recompilation of the code that must be protected.

VI. CONCLUSION

This paper has presented a multilayer approach to hinder Return-Oriented Programming attacks. Each layer of the proposed system validates targets of indirect branches; hence, proving that they belong to legitimate program flows. Our key insight is to combine layers, so that layer \( L_i \) runs at a lower computational cost than \( L_{i+1} \). A branch certified at \( L_i \) does not need to be checked at \( L_{i+1} \). We apply stronger enforcement guarantees only onto cases that are hard to verify, because \( L_i \), by construction, receives more branches than \( L_{i+1} \).

Recent developments showed that there exists no system that is able to stop any ROP attack [23]. Ours is not an exception to this rule: it still faces false positives and false negatives. False positives happen if we flag a legitimate branch target as unsafe. The way in which the operating system treats exceptions to this rule: it still faces false positives and false negatives happen if we allow an attacker to bend the validation constraint leaves a very small number of gadgets available for the construction of an attack. Consequently, state-of-the-art exploits, such as those carried out by Carlini et al. [23] are not possible. The construction of attacks that circumvent our protection is, therefore, an open problem.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Marcus Botacin for many suggestions that greatly improved this paper. This work has been made possible by the financial support of CNPq, CAPES, FAPEMIG, Intel, and CEFET-MG, which granted Mateus Tymburibá his sabbatical.

REFERENCES
