Rest in Protection
A Kernel-level Approach to Mitigate RIP Tampering

Vincent Haupert and Tilo Müller
Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU),
Martensstraße 3, Erlangen, Germany
{vincent.haupert, tilo.mueller}@cs.fau.de

Keywords: RIP Protection, ROP Prevention, Instruction-level Monitoring, Linux Kernel.

Abstract: We present RIProtection (*Rest In Protection*), a novel Linux kernel-based approach that mitigates the tampering of return instruction pointers. RIProtection uses single stepping on branches for instruction-level monitoring to guarantee the integrity of the `ret`-based control-flow of user-mode programs. Our modular design of RIProtection allows an easy adoption of several security approaches relying on instruction-level monitoring. For this paper, we implemented two exclusive approaches to protect RIPs: XOR-based encryption as well as a shadow stack. Both approaches provide reliable protection of RIPs, while the shadow stack additionally prevents return-oriented programming and withstands information leakages of the user-mode stack. While the performance of RIProtection is a severe drawback, its compatibility with regard to hardware and software requirements is outstanding because it supports virtually all 64-bit programs without recompilation or binary rewriting.

1 INTRODUCTION

Numerous exploit techniques for memory-safety vulnerabilities are known and have been widely applied for years. Exploiting these mistakes was easy from the beginning but many of the approaches available today limit or even prevent the damage. The most widely applied safeguards in use at present are *Address Space Layout Randomization* (ASLR), which randomizes the base address of segments, the *No eXecute* (NX) bit, which marks a page as either executable or not, and stack *canaries*, which place a unique random value in front of the RIP. All three countermeasures significantly increase the effort an attacker needs to undertake, especially if ASLR, NX and canaries are combined.

When it comes to creating exploits that subvert a great number of countermeasures, however, return-oriented programming (ROP) (Shacham, 2007) has become the method of choice. ROP can circumvent NX because it uses chained addresses (so-called *gadgets*) of segments that do not have the NX bit set. All these gadgets eventually execute a `ret`. ASLR can be subverted by the prior exploitation of an information leakage vulnerability (Sotirov and Dowd, 2008). If the latter is the case, canaries can be trivially disabled, too (Bulba and Kil3r, 2000). Especially ROP distorts the original intent of a `ret` because it does not read an address that was previously written by its corresponding `call`.

1.1 Contributions

Our work yields the following contributions:

(a) We propose RIProtection, a novel kernel-based protection mechanism that uses single steps on branches to synchronously monitor any branch instruction. RIProtection currently only monitors `call` and `ret` instructions but is able to monitor any branch. Our approach is developed to run on commodity hardware and facilitates the branch trap flag and the last branch record of the Intel x86 CPU. Both features have been available for years now and, therefore, allow high compatibility.

(b) Different types of security models are supported on a per-process basis. This work presents two approaches: One relying on the XOR encryption of return addresses, the other using a shadow stack to provide enhanced security.

(c) Our security evaluation shows that RIProtection can safely prevent RIP tampering as well as ROP and information leakage attacks in the shadow stack operation mode. Furthermore, our empiric evaluation of the shadow stack security module proves that all `ret`-based attacks are prevented

Haupert, V. and Müller, T.
Rest in Protection - A Kernel-level Approach to Mitigate RIP Tampering.
DOI: 10.5220/0006083800250037
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 25-37
ISBN: 978-989-758-209-7

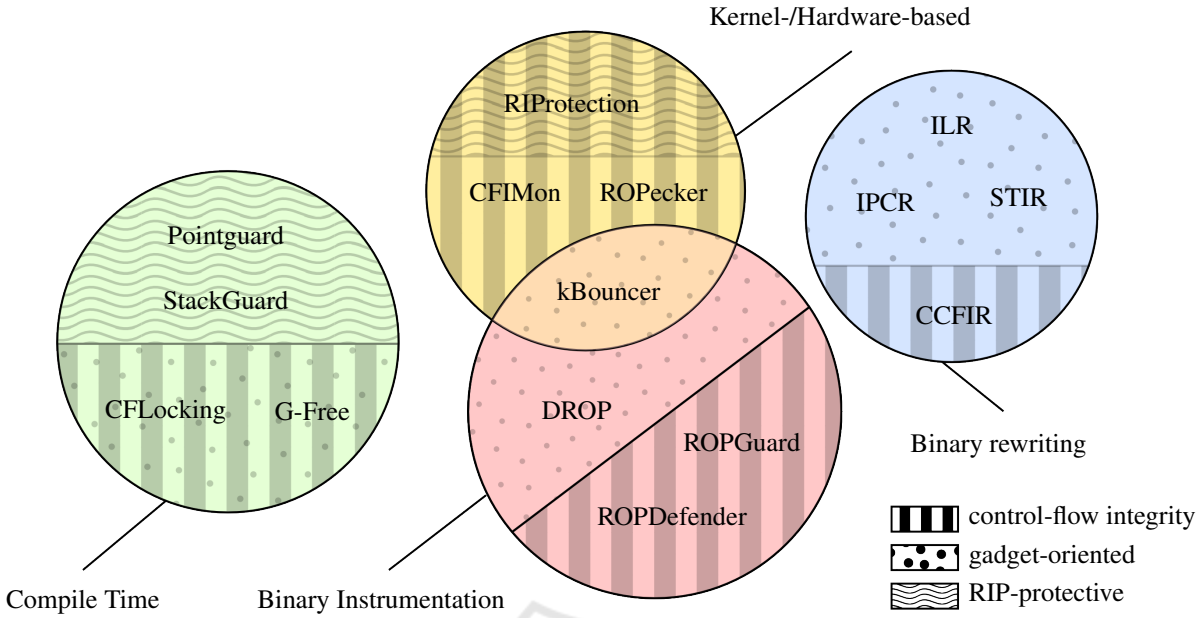

Figure 1: The Landscape of Related Work: Each circle represents the technical approach that a defense mechanism uses. While the compiler, binary rewriting and instrumentation are rather precise and self-explanatory, the kernel- and hardware-aided category is more generalized and sums up techniques that try to facilitate hardware features. Additionally, the circles are subdivided to illustrate the concept that an approach is using. They might aim to assert control-flow integrity, concentrate to eliminate gadgets, focus on protecting the RIP or use multiple concepts.

and an effective gain in security is achieved. Additionally, we evaluated performance and compatibility. While run-time performance is a big issue, RIProtection has high compatibility with any x86_64 legacy binary without requiring any side information, source code, binary rewriting or user-level binary instrumentation.

1.2 Outline

The outline of this paper is as follows. Section 2 covers related work on exploit mitigation that uses compile-time approaches, binary rewriting, binary instrumentation or a kernel-based approach. Section 3 describes the design and implementation of RIProtection. Section 4 is dedicated to the evaluation of RIProtection in terms of security, performance and compatibility. Finally, Section 5 provides a conclusion and possibilities for future work.

2 LANDSCAPE OF RELATED WORK

All defense techniques presented in this section can be categorized to some extent to use compiler-based solutions, binary rewriting, instrumentation or a hardware-aided kernel-level approach as a technical strategy. In addition to these techniques, they have a concept that may enforce control-flow integrity, has the goal of making ROP gadgets unavailable to an attacker or focuses on mitigation of the RIP. In contrast to all related work, RIProtection operates only in kernel-mode without modification of hardware or user mode software. 1 illustrates the landscape of the different technical approaches described in the following sections.

2.1 Compile-time Approaches

This section is dedicated to compiler-level extensions that aim to improve the security of programs by statically adding security logic to the source code. Many works use this approach as it is rather convenient to add checks for improved security. However, a natural feature of all these approaches is that they need the source code to be available.

StackGuard (Cowan et al., 1998) is a patch for the GCC compiler that is available using the `-fstack-protector` option. It inserts a random cookie (also called canary) in front of the RIP that is checked during a function's epilogue. StackGuard is a potent mitigation technique that significantly increases the effort that an attacker needs to take and introduces only a slight performance overhead. Espe-

cially the fact that only one secret is used for the entire process, however, renders canaries vulnerable to information leakage attacks (Bulba and Kil3r, 2000).

Pointguard (Cowan et al., 2003) is yet another patch for the GCC that encrypts any pointers and decrypts them only if they are about to get loaded into a CPU register. It uses XORencryption with a key that is acquired at program start using `/dev/urandom`. In general, Pointguard can provide security not only for the RIP but for every pointer in the program. Yet, it uses only one secret for the entire process and, consequently, becomes especially vulnerable against information leakage attacks.

An approach that wants to eliminate useful gadgets is G-Free (Onarlioglu et al., 2010). To achieve this, two techniques are used. First, any unintended code sequences are transformed that could be read as `ret` or `jmp`. To prevent the usage of these instructions created by the compiler, G-Free additionally enforces control-flow integrity by ensuring that a function is called only from its entry point. The fact that G-Free first removes unaligned gadgets and later adds code that is required for its approach may introduce new gadgets (Cheng et al., 2014). Furthermore, the policy to only allow functions to be called from their entry point cannot always prevent ROP attacks (Göktaş et al., 2014).

Another compiler extension that uses control-flow integrity in conjunction with the elimination of unintended gadgets is CFLocking (Bletsch et al., 2011). First, non-intended gadgets are eliminated. To account for the name of the tool, a lock that is immediately unlocked when the control flow is transferred to a legal target in the control-flow graph is introduced. If a branch occurs, the lock is checked again and if it is still locked, the program is terminated. In general, CFLocking suffers from similar drawbacks as G-Free.

2.2 Binary Rewriting

The technique of binary rewriting aims to modify an existing program without recompilation.

A gadget-oriented tool that does not aim to remove useful gadgets but uses a randomization approach to make gadgets unavailable is ILR (Hiser et al., 2012). In contrast to ASLR, which randomizes only the base of a segment, ILR randomizes the address of every instruction. This cannot be achieved with x86 native code, which is why ILR uses a process-level virtual machine (PVM). The principle of operation consists of an offline and a run-time phase. In the offline phase, ILR analyzes the binary and creates a randomized binary and a fall-through map for the PVM to guide execution in the run-time phase. Even though ILR can randomize up to 99.7% of the instructions in a binary, research has proven that even very small programs can provide enough gadgets for a successful exploit (Schwartz et al., 2011). Furthermore, shared libraries are not affected by ILR. They are solely protected by ASLR and, thus, provide a potential attack surface.

Another instruction randomization approach is STIR (Wartell et al., 2012). Similar to ILR – but without the need of a virtual machine – it does not rely on any side information and, therefore, requires only the binary. However, STIR randomizes only the binary at a basic block granularity and uses x86 native code to dynamically determine the address of each basic block. The level of randomization is relatively high but a leaked basic block can again provide enough gadgets to break this defense (Schwartz et al., 2011).

The x86 architecture is generally an attractive target for code-reuse attacks (Roemer et al., 2010). Its instruction encoding is of variable length and must not be unaligned, which allows interpreting data as instructions. IPR (Pappas et al., 2012) is a sophisticated solution for IA-32 that aims to eliminate unintended gadgets. In general, IPR uses three binary rewrite techniques that do not change the semantics of the program but provide equivalent substitution of some binary patterns.

CCFIR (Zhang et al., 2013) aims to combine a gadget-oriented approach with control-flow integrity mechanisms. On the control-flow side, CCFIR restricts indirect call and jump instructions such that they may only transfer the control flow to the function entry point or are entirely prohibited (e.g. sensitive API calls may only be accessed using a direct call). To achieve this, CCFIR disassembles the binary and creates an entry for each indirect branch in a so-called *springboard*. As this approach uses control-flow integrity, ROP can be successfully prevented. However, an attacker might still jump to valid targets in the control-flow graph.

2.3 Binary Instrumentation

The approach of binary instrumentation is similar to binary rewriting. However, instrumentation does not only rewrite existing code but also adds additional checks – several instructions are, therefore, *instrumented*. In contrast to RIProtection, all these approaches instrument the binary in user-mode rather than in kernel-mode.

DROP (Chen et al., 2009) employs instrumentation to realize a gadget-oriented technique. Their defense tool relies on two parameters: the size of each potential gadget (G_size) and the length of contiguous

gadget sequences (S_length). If a specific threshold for G_size and max(S_length) is reached, it is identified as malicious ROP code. According to its DROP does not yield false positive or false negatives. Recent work, however, has shown that approaches which rely on gadget size and length cannot reliably protect programs and, thus, are still vulnerable (Göktaş et al., 2014).

Another instrumentation-based approach that tries to enforce control-flow integrity is ROPDefender (Davi et al., 2011). It uses the dynamic binary instrumentation framework Pin (Luk et al., 2005) to instrument any binary without having access to any side information. ROPDefender ensures that every `ret` reads only the RIP placed by its corresponding `call`, and utilizes a shadow stack to achieve this. The implementation is rather compact as the Pin framework provides most of the features that ROPDefender requires. In contrast to RIProtection, ROPDefender uses only a shadow stack of return addresses and does not save the value of the stack pointer. In our opinion, this could lead to malfunctioning, e.g. in some cases of exception handling, if the same return address exists more than once in the (shadow) stack.

ROPGuard (Fratrić, 2012) is another state-of-the-art defense technique that is part of the *Enhanced Mitigation Experience Toolkit* (EMET) (Defense, 2012). It tries to enforce a coarse-grained control-flow policy by using binary instrumentation to hook several critical Windows API functions on IA-32. When one of these functions is executed, ROPGuard performs two checks. At first it probes whether the RIP of the API function call is `call`-preceded. ROPGuard performs this `call`-preceded check not only on the RIP but also emulates a few steps in the control flow to catch a future non-`call`-preceded return. The second check is committed to the stack. ROPGuard detects attempts to modify stack pointers in such a way that they point into the heap or another attacker-controlled memory region. Additionally, the stack may not be set to executable. Nevertheless, also ROPGuard can be bypassed as shown in the literature (Davi et al., 2014).

The kBouncer (Pappas et al., 2013) approach not only relies on binary instrumentation but also uses the last branch record (LBR) of modern processors. As a top-level concept, it enforces a coarse-grained control-flow integrity policy that hooks all Windows API calls. The remarkable difference to ROPGuard is that it not only checks if the procedure is call-preceded but also probes previously adopted branches with the facilities that the LBR offers. Like other approaches that utilize coarse-grained control-flow integrity, kBouncer can also be bypassed (Carlini and Wagner, 2014). The basic steps to exploit a program if kBouncer is in use are the following: 1) place the ROP payload as usual but do not invoke the syscall yet, 2) flush history, 3) restore the state of registers that might have changed due to the history flushing, and 4) execute the syscall.

2.4 Kernel- and Hardware-based Approaches

This section describes approaches which have in common that they all attempt to use hardware features and/or are implemented in kernel mode.

Address Space Layout Randomization (ASLR) is a standard mitigation technique that every modern operating system has integrated today. The OS maps program sections during the load-time of a process. Most notably, the sections for the stack and heap are randomized and, therefore, an attacker cannot overwrite the RIP with an address that points into any segment that is covered by ASLR. Besides being vulnerable to brute-force attacks (Shacham et al., 2004), it is constantly being bypassed using information leakage attacks (Sotirov and Dowd, 2008) and, as a result, has the same attack surface as canaries. This is a major issue, because even though ASLR and canaries are widely used nowadays, they can be disabled using the same vulnerability.

The *No eXecute* (NX) bit is a page entry bit that marks an entire page as either executable or not. Together with ASLR, it is one of the default defense techniques (van der Veen et al., 2012). When an instruction is scheduled for execution, the memory management unit (MMU) probes the page that the address belongs to and if the NX bit is set, the processor yields an exception causing the process to be terminated. With NX enabled, it is no longer possible to execute payload injected via a buffer overrun, because the stack is not marked as executable. The method of choice to circumvent NX these days is ROP (Shacham, 2007). This technique does not call functions but only a short sequence of instructions preceding a `ret`. As a consequence, the program counter is always loaded with an address of an executable section. Thus, the NX bit cannot prevent the execution of malicious code because the PC is only loaded with addresses from executable segments.

CFIMon (Xia et al., 2012) uses hardware performance counters and the Linux kernel subsystem `perf events` to detect control-flow violations. Therefore, they use the branch trace store (BTS) of recent processors. When BTS is used, all taken branches are recorded in memory. The protection that CFIMon offers is to run whenever the protected application calls

`execve(2)`. At this point, CFIMon reads the BTS and each branch is qualified as legal, illegal or suspicious. In contrast to the LBR, using the BTS has a notable performance impact as branches are traced to memory rather than to specific registers. Furthermore, the assumption that every exploit ultimately leads to a call to `execve(2)` a weakness because malicious code can obviously deal damage without issuing an `execve(2)` system call.

ROPecker (Cheng et al., 2014) is a state-of-the-art ROP prevention tool that performs various checks to enforce some sort of coarse-grained control-flow integrity. ROPecker is triggered on some "risky system calls" like `mmap(2)` and, additionally, if code is executed that resides on another page as the previous page. Whenever ROPecker is triggered during the run-time phase, it first consults the database created in the offline phase and the LBR to check whether the address points to a gadget. If a specific gadget chain threshold is exceeded, ROPecker terminates the process. To fill the gap until the next critical system call or page fault, ROPecker also implements future instruction emulation. Even though the authors admit that their approach could especially struggle with short gadget chains, they do not acknowledge this as a realistic threat. However, it has been proven that ROPecker can indeed be defeated using history flushing and gadget lengths below the threshold in a similar fashion as the exploitation of kBouncer requires (Göktaş et al., 2014).

3 DESIGN AND IMPLEMENTATION

In this section, we present RIProtection, a novel approach that does not require recompilation, binary rewriting or any other side information such as debug symbols. First, we describe the general idea behind RIProtection and, subsequently, present our design decisions for instruction-level monitoring along with some implementation details of our approach. Finally, the technical implementation is outlined and a typical RIProtection life cycle is shown.

3.1 Basic Idea and Concept

The high-level concept of RIProtection is rather generic. As illustrated in Figure 2, it consists of two parts. One part is responsible for instruction-level monitoring (ILM). ILM determines the kind of branch and invokes the second part, one of the underlying security modules SecMod_1 to SecMod_n. The selected security module SecMod_i then is aware of

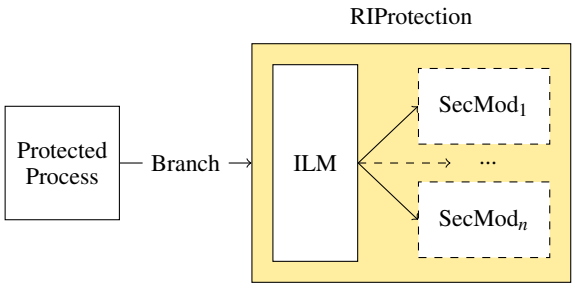

Figure 2: The concept of RIProtection consists of a component that is used for instruction-level monitoring (ILM) and of several possible security modules (SecMod_x).

which branch was executed and can enforce a policy of choice. RIProtection allows different security mechanisms on a per-process basis. Even though, for our work, only `call` and `ret` instruction are of interest, RIProtection is capable of monitoring any branch instruction. Therefore, the monitoring of e.g. `jmp` instructions could be realized to implement a mechanism to prevent jump-oriented programming in the future (Checkoway et al., 2010).

3.2 Single Step on Branches

To realize ILM, RIProtection makes use of the Intel x86 debug capabilities like the hardware supported feature for single stepping. If the TF flag in the EFLAGS register is set, the processor generates an exception on every instruction. Since single stepping allows monitoring all instructions, it is slow in terms of performance because each instruction causes a context switch to the kernel. A rather unknown flag in the `IA32_DEBUGCTL` MSR, the branch trap flag (BTF), causes the processor to treat the TF flag in the EFLAGS register only as a *single step on branches* rather than a single step on each instruction (Corporation, 2014).

A branch is actually every instruction that does not incrementally raise the PC such that noticeable overhead is especially accounted by instructions of the `jmp` family. Every time the PC is explicitly loaded with an address rather than being implicitly incremented, a debug exception is generated. A debug exception caused by the BTF flag is a synchronous interrupt and is categorized as a trap (Corporation, 2014). This means that the processor generates an interrupt *after* an instruction was executed. This information is important as it affects the steps needed to access and modify the RIP.

The handling of `call` and `ret` is as follows. When a `call` is executed, the RIP is already on the stack and can directly be processed by one of the security modules. The `ret` case, however, is more complex

because the `ret` instruction occasionally reads illegal addresses, e.g. if the RIP on the program stack is modified by the security module. But even though the program counter is already loaded with the RIP, the processor has not yet begun with the instruction fetch. Consequently, the PC can be processed without touching the RIP on the stack. It becomes challenging to determine the branch type if the interrupt is triggered after instruction execution. In a bare software approach, this task would involve reading the RIP from the stack and disassembling it to find the instruction that caused the branch. This is tricky because of the variable length of CISC instructions.

However, modern processors offer the possibility to monitor the source and destination of taken branches, interrupts and exceptions in a set of registers called the last branch record (LBR). As these entries are captured using bare hardware and registers instead of memory, it has near-zero performance impact. Although not widely used in the past, LBR registers are available since the Pentium 4 architecture (Corporation, 2014) and initially offered four model-specific registers that record the from and to address of a taken branch. Furthermore, the LBR can be configured to record only taken branches in the user-level. Therefore, any architecture that supports the LBR is suitable for RIProtection.

3.3 XOR Security Module

We implemented two security modules, one based on XOR encryption and another based on a shadow stack, both of which rely on monitoring `call` and `ret` instructions.

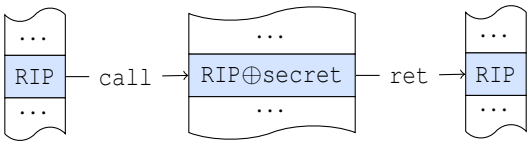

Figure 3: The XOR security module.

The XOR module acquires a unique random key when the process is protected. During execution of the program, `call` and `ret` can be treated in the same fashion because XOR is used to encrypt and decrypt the RIP with the process secret. This approach does not have any logic probing whether a RIP was modified because the XOR encryption on each `call` and `ret` already establishes an implicit program termination if an attacker tampered a RIP. Overwriting the RIP with plaintext addresses results in illegal addresses and consequently in process termination due to segmentation violation as the RIP on the stack is XOR-processed on each `ret`. 3 illustrates these steps.

3.4 Shadow Stack Security Module

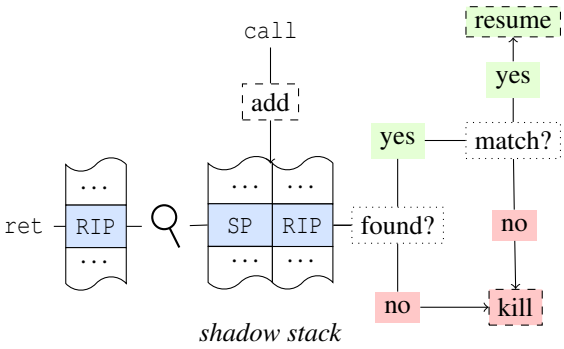

Figure 4: The shadow stack security module.

As visualized in 4, the shadow stack approach does not modify the RIP on the program stack. In contrast to the XOR module, the shadow stack requires additional bookkeeping structures for the RIPs. On `call`, a new mapping on the shadow stack is created using the SP as key and the RIP as value. On `ret`, the SP is used to retrieve the shadow RIP. Our implementation of the shadow stack uses a map of key-value pairs rather than only saving the return address. Related work such as the ROPDefender only saves the RIP on a `call` and, therefore, needs separate handling for functions or exceptions that jump over several stack frames. If the SP value is saved as a key, no special treatment is required and the list is traversed in reverse order to find the matching entry. The correct entry, according to the stack data structure, is usually the last entry added. If, however, no entry is found or the PC and the shadow RIP do not match, the program is killed. Otherwise, the program is resumed.

3.5 Implementation

This section explains the technical implementation of the design described in the previous section. The implementation of RIProtection is carried out as a Linux v3.13 kernel patch for Ubuntu 14.04.1 LTS in about 1,000 lines of code. Overall, the patch modifies 12 existing files and adds four new files.

5 shows the architecture and execution flow of RIProtection. To indicate whether a program is to be protected by RIProtection, we modified the kernel code of the ELF binary handler. RIProtection introduces `EI_RIProtection` at byte 8 of the ELF header, which is labeled as *R* in 5. If *R* is set to a value greater than zero, RIProtection invokes `riprotection_protect` just before the new process is executed. In this step, RIProtection adds the task to its internal list of protected tasks and sets the TF flag in the EFLAGS register as well as the LBR and

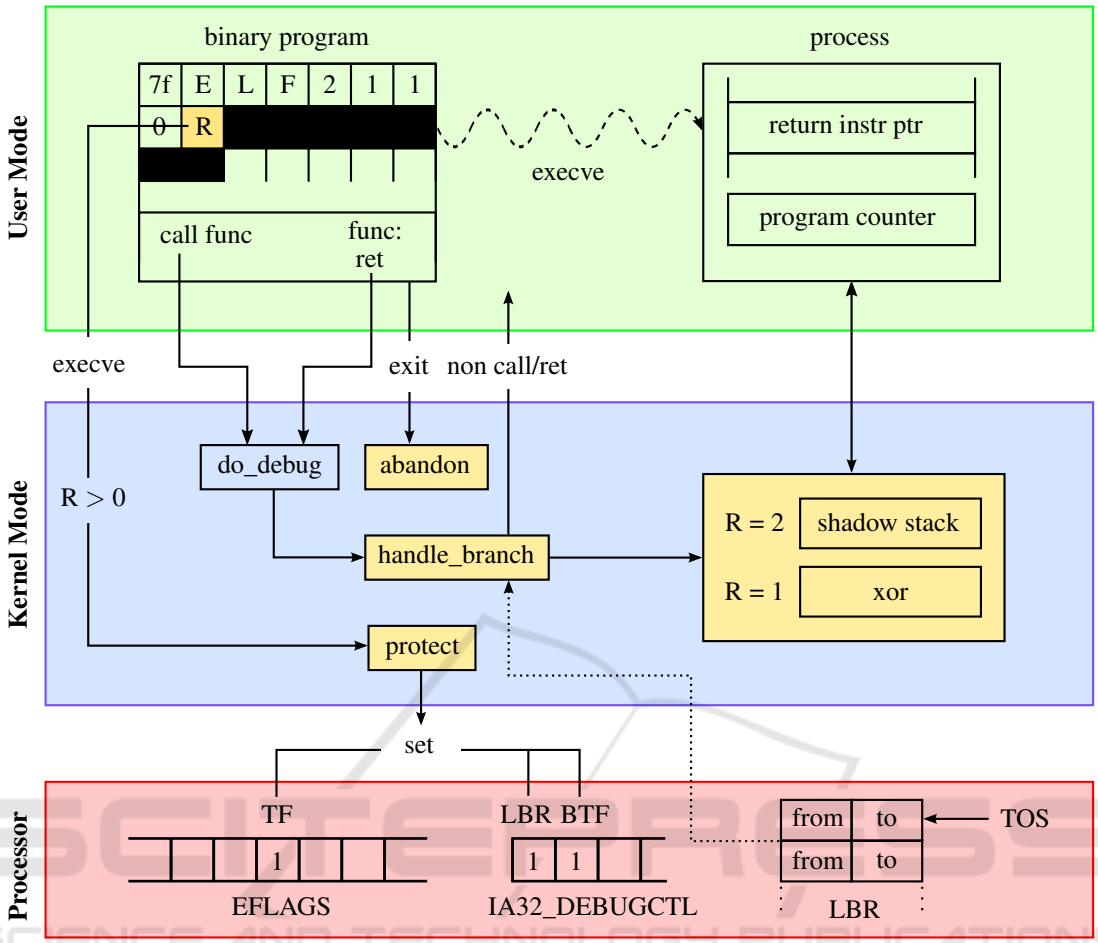

Figure 5: *The architecture of RIProtection*: The execution flow and interaction of user-mode, kernel-mode and processor features. Functions and features introduced by RIProtection are filled with a yellow background.

BTF bit in the `IA32_DEBUGCTL` MSR. From this point on, the CPU records all branches, interrupts and exceptions taken in user space. Furthermore, an interrupt is generated on each branch. Additionally, the `protect` function determines which security module (XOR or shadow stack) is used and invokes the initialization handler of the selected module. For the XOR method, the initialization involves only acquiring a task-specific 48- or 32-bit random key using `get_random_bytes` for x86_64 and IA-32 binaries respectively. If the shadow stack module is used, it allocates and initializes the head of the shadow stack list structure.

When the kernel receives an interrupt due to a branch instruction in the protected program, `do_debug` is called. In this function, `handle_branch` is executed if the task is in the list of protected tasks. In this case, `handle_branch` retrieves the most recent *from*-entry of the LBR to find out the address of the instruction that caused the interrupt. The `handle_branch` function also increments the task-specific counters for `ret`, `call` and the total amount of branches depending on the type of branch. These statistics are available through the `/proc` filesystem for all running tasks which are protected by RIProtection and are also available for the protected task that terminated most recently.

Both security modules need to read the RIP. If the debug trap occurred due to a `call` instruction, the address of the following address (i.e. the RIP) was already pushed on the stack. Hence, the `call` handler needs to copy the RIP to the kernel space. In the following, the XOR approach encrypts the RIP with the process secret and write it back. The shadow stack creates a new entry in the shadow list, which consists of key-value pairs while the current value of the SP acts as a key and RIP as a value.

As soon as a `ret` is executed, the program counter is already loaded with the entry found on the process stack as depicted by the current value of the SP. Therefore, the user space stack of the process is not accessed for retrieving the RIP. Instead, the `ret` han-

dler routine of the security module directly reads the task's program counter. Not even the XOR module needs to access the process memory after decrypting the address because the value of the SP was already increased and entries with a lower value than the current SP are free by definition. The detection of RIP tampering is implicit for the XOR module. There is no explicit logic in the code that marks code as malicious and causes RIProtection to terminate the process. Nevertheless, the process is indirectly killed due to a segmentation violation.

The `ret` handler of the shadow stack approach must first look for the correct entry in its internal list by comparing the SP key of entries with the current SP value incremented by one. Later, the shadow RIP is compared to the RIP loaded into the program counter. If both do not match, the RIP was tampered with and the process is terminated. In the case that no entry matches the incremented SP value, the program counter is loaded with an address that was not placed using a `call` instruction and the program is killed. In this situation, an attacker likely tried to launch a ROP attack and the normal control flow of the program was tampered. If an entry in the shadow stack is found and matches, the entry is removed from the list and the program is resumed.

If `handle_branch` could neither detect a `call` nor a `ret`, the control is immediately transferred back to the application. This is not a rare case as a trap is also generated for all `jmp` instructions that are common in every program. Unfortunately, it is impossible to filter out `jmp`s in hardware leading to a severe performance drawback of our approach as explained in 4.2.

For every process, `do_exit` is eventually called. This is also true if the program is killed instead of explicitly calling `exit(2)`. When this point is reached, `abandon` is called and disables TF and BTF, and releases the `perf events` kernel counter to stop capturing branches for that task. Ultimately, the task is removed from the list of protected tasks.

4 EVALUATION

In this section, we evaluate RIProtection and the XOR and shadow stack modules in terms of security, performance and compatibility. All three categories were examined running Ubuntu 14.04.1 LTS using a hyper-threaded, dual-core Intel Core i5-650.

4.1 Security

This section is dedicated to the security evaluation and discussion of RIProtection. The underlying XOR and shadow stack security modules are examined individually and in contrast to each other.

First, we will consider if it is possible to entirely disable the instruction-level monitoring facilities of RIProtection. The single step on branches method depends on two values. The TF flag in the EFLAGS register and the BTF flag in the `IA32_DEBUGCTL` MSR. Model-specific registers can only be accessed from the kernel-level and, therefore, are usually not accessible from the user space (Corporation, 2014). Consequently, the `IA32_DEBUGCTL` is well-protected and tampering with it is not easy. The EFLAGS register, however, can be modified using the `pushf` and `popf` instructions. This is not a security flaw, even though clearing the TF flag in the EFLAGS register disables single stepping completely because attackers must first gain control over the execution flow to execute these instructions. As covered in detail in the following section about the compatibility of RIProtection, the TF flag in the EFLAGS register may not be modified by the application itself. Otherwise, RIProtection cannot protect the program.

4.1.1 XOR Module

Although RIProtection's instruction-monitoring facilities cannot be disabled, it might be possible to get around its underlying security modules while they are in use. In many aspects, the protection and the attack surface of RIProtection operating in XOR mode can be compared to stack canaries as used by StackGuard. As explained in 2, canaries place a unique and random value beside the RIP. This value is checked in the function epilogue just before a `ret` is executed. If the check fails, the program is terminated. The XOR security module does not require an individual check after `ret` execution because the XOR with the process-specific secret creates an invalid address as long as the attacker does not know the process-specific secret. As already stated, however, stack canaries also rely on such a secret and it also has been proved that they can be bypassed (Bulba and Kil3r, 2000). Usually, this is achieved by exploiting an information leakage vulnerability that reveals the value of the canary. Afterward a buffer overflow vulnerability is exploited, for example, and the canary value is simply rewritten.

In a similar way, the XOR security module of RIProtection cannot withstand an information leakage vulnerability that allows the attacker to read the value of the RIP. The RIP is encrypted using a simple XOR of the secret and the original RIP. It comes in handy for the implementation of the approach that the inverse function of the XOR function is the XOR function again. However, if an adversary has knowledge of an encrypted RIP and knows the original ad-

dress, he or she can deduce the secret easily. As far as Linux is concerned, the address of the original RIP is often statically known because the `.text` section is sometimes unaffected by ASLR because of performance reasons.

Even with full ASLR enabled, the key space of the XOR module can be reduced due to the nature of the XOR function. The offsets between instructions are still known and if an adversary is able to leak more than one XOR encrypted RIP he or she can leak at most as many bits as the highest offset between two XOR encrypted RIPs. In general, the process secret of a 64-bit executable protected by RIProtection in the XOR operation mode has as an entropy of 48-bit as the remaining most significant bits are currently sign-extended with bit 47. The offset between the RIPs is statically known and the following expression is true: $RIP_1 \oplus RIP_2 = (RIP_1 \oplus secret) \oplus (RIP_2 \oplus secret)$. With those two conditions the key space is potentially reduced with every leaked address and an adversary might be able to compute the secret within a reasonable period. Nevertheless, this remains to be a rather theoretical vulnerability as the key can be easily computed if an attacker is able to leak memory.

At first glance, full ASLR cannot provide enhanced security, as ASLR is also vulnerable to information leakage attacks. Nevertheless, consider an example of a program with an information leakage vulnerability for which an attacker wants to execute ROP gadgets that reside in the program's `.text` section. Additionally, assume that the program is executed in an environment where full ASLR and NX are enabled. The first approach that an adversary would usually take to create the ROP payload is to read the RIP because it provides a perfect address for the `.text` section. If RIProtection with XOR encryption is in charge, this would fail due to the encryption of the RIP with a so far unknown secret. On the other hand, the adversary cannot compute the secret, even though he or she is able to read the encrypted RIP. To achieve this, the attacker needs to determine the real, unencrypted RIP, which is protected by ASLR. Nevertheless, even such a scenario cannot disable this kind of attack entirely. An attacker might leak a function pointer or any other data pointing to `.text` and, thus, be able to compute the process secret.

4.1.2 Shadow Stack Module

The shadow stack security module pursues a different approach than the XOR-based encryption and is slightly more complicated. When a `ret` instruction is executed after an attack begins, there are two cases where the shadow stack provides protection. If the RIP is modified directly, the handler function finds a shadow RIP that differs from the address that is loaded into the PC. Consequently, the process is terminated. The other case assumes that an attacker could start an exploit that does not take over the control flow by modifying the RIP. This can be achieved by GOT hijacking, for example. To execute further malicious instructions, a code-reuse attack like ROP can be used. If RIProtection is used in its shadow stack mode, this fails because ROP will execute a series of gadgets ending in a `ret` instruction without a corresponding `call`. As a result, the shadow list of the shadow stack security module cannot find an entry for the SP value and accounts for this by terminating the program. The attack surface of the shadow stack approach is substantially different from that of the XOR mode because it does not reveal any of the data it operates on to the process. Therefore, the shadow stack does not provide any points to attack for information leakage because its data resides entirely in the kernel space.

The shadow stack can not only protect the program from classic RIP modification and ROP, but is also not disabled easily. The fact that the XOR security module is vulnerable to information leakage makes it especially inferior because two widely used safeguards, ASLR and canaries, provide the same attack surface. Therefore, an attacker might circumvent ASLR, canaries and RIProtection in its XOR mode using the same vulnerability.

4.2 Performance

We acquired our performance data using the nbench2 (Mayer, 2011) CPU benchmark which measures the operations per second for various algorithms. Note that the benchmark offers three more tests but those were omitted to improve the readability of the results as even their native test performance scores below 10 operations per second.

Our evaluation compares the performance without and with RIProtection running in either the XOR or shadow stack operation mode. The results are depicted in 6 and visualize the run-time multiplier of RIProtection. Furthermore, we also evaluated how often RIProtection was invoked due to a `call`/`ret` instruction or due to any other branch. The benchmarking result for the run-time multiplier yields a noticeable performance overhead introduced by RIProtection ranging from 93 for the FP Emulation test to 827 for the LU Decomposition test. The average operations per second is almost equal, being 46.39 for XOR and 46.10 for the shadow stack. For tests with a high `call-ret` amount, e.g. the Fourier test, the XOR tends to perform slightly better than the shadow stack operation mode.

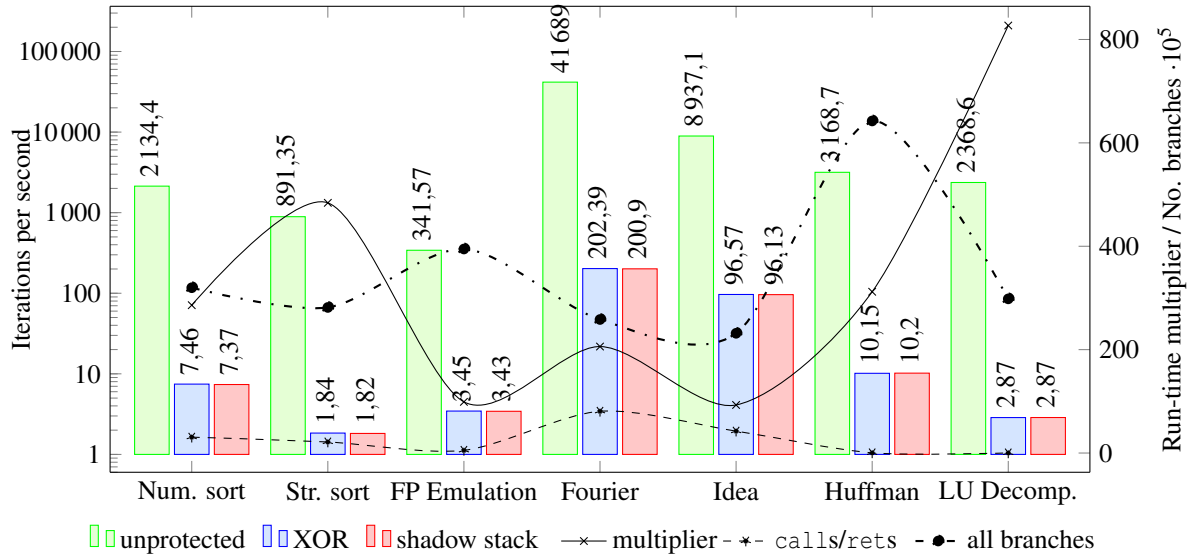

Figure 6: Benchmarking results for various tests of the nbench2 CPU benchmark. The green bar shows the iterations per second using nbench2 without protection while the blue and red bars indicate a run using the XOR and shadow stack operation modes respectively. The other graphs display the average run-time multiplier of both security modules while the dashed and dash-dotted graphs indicate the number of branches due to a `call` or `ret` and the total amount of branches.

In general, one would expect the XOR-based encryption to perform notably better than the shadow stack because it does not require any additional structures on each `call` or `ret` and provides constant operation time for every invocation of its logic. The results, however, indicate that the advantage of XOR is minimal and that most of the average run-time performance differences stem from measuring inaccuracy. This suggests that the overall bottleneck is not the underlying security module but the BTF-based instruction-monitoring approach of the ILM unit.

Furthermore, one would assume that the run-time multiplier is linearly dependent on the total count of branches because each one causes an interrupt. The LU Decomposition test result, however, has the highest run-time multiplier but a branch count that is 22% below the average number of branches. This suggests that the LU Decomposition test suffers from the context switch which also has an effect on the contents of the CPU cache.

The performance overhead introduced by RIProtection has primarily two reasons. The first one is that the BTF flag triggers a debug trap whenever a branch occurs. This is especially an issue for `jmp` instructions that occur often. The second reason is that an interrupt is per se a time-consuming operation that involves an expensive context switch to kernel-mode that also confuses hardware components like the CPU cache. Nevertheless, the performance evaluation shows that global statements on the performance of programs cannot be made because it is highly dependent on the structure of the program.

4.3 Compatibility

RIProtection can protect any 64-bit ELF program that runs in user-mode if the following requirements are met. In any case, RIProtection must be explicitly enabled , as explained in 3.5.

First, the hardware requirements of RIProtection must be met. As already stated in 3, the CPU needs to support single stepping on branches and the last branch record. The number of branches that the LBR can record has increased with processor generations but for RIProtection only one entry is required. As any Intel CPU that features the LBR also supports BTF, only a CPU with LBR is required.

Another primary requirement of RIProtection is that a program does not modify the TF flag in the EFLAGS register. The EFLAGS are indirectly modified by the `pushf` and `popf` instructions and, hence, protected applications are not allowed to use these instructions. Due to this limitation, single step debugging is neither allowed, because the debugger modifies the EFLAGS, nor possible, as the debugger does not receive any signals due to RIProtection. Moreover, the backtrace feature that many debuggers usually offer cannot be used if RIProtection protects the debugged process in XOR mode, because all the RIPs are encrypted.

Furthermore, the compiler that was used or is used to build the program needs to follow the convention

that a `ret` is a return from a procedure previously called using a `call` instruction (Corporation, 2014). This is certainly true for popular compilers like GCC and LLVM but may not be the case for some obfuscated binaries. The same applies to the use of (inline) assembler and therefore must not tamper with any RIP and must not introduce `ret` instructions that read a RIP not placed by its corresponding `call`.

Multi-threaded applications follow the same line. RIProtection supports multi-threading as long as threads are managed in kernel-mode. It is important that the scheduling of threads is carried out in the kernel as the scheduler needs to execute a `ret` without a corresponding `call`. This violates the fundamental assumption of RIProtection. However, this operation is not a problem inside the kernel because RIProtection applies its defense mechanisms only to user-level programs and, therefore, is only triggered there.

5 FUTURE WORK

The XOR security module could be hardened against information leakage by increasing the number of task-specific secrets to n. As a consequence, the $log_2(n)$ least significant or most significant bits could indicate the key to use. This has the drawback that $log_2(n)$ bits of entropy are lost. One needs to take care that the entropy does not drop too low. In general, the concept would require slightly more space but maintains an easy approach. Moreover, additional security could be introduced by implementing the concept of call-preceded `ret`s as is also found in related work. In a case where an adversary was able to leak the secret and, consequently, is able to generate a valid ROP payload, he or she is at least forced to use gadgets that are call-preceded.

Even though the security modules already rely on a modular design and can be easily implemented, they are still part of the Linux kernel. The development of additional security modules would be easier and more convenient if each security approach is a separate loadable kernel module (LKM).

The security evaluation of the XOR approach reveals that the shadow stack module is superior in terms of protection. However, the XOR approach is still of interest because it could be easily realized in hardware by introducing a register that holds the task-specific secret. The $\oplus$ encryption on `call` and `ret` could be achieved in a similar fashion to RIProtection. If implemented in hardware, the attack surface would remain the same but probably with a near-zero performance impact. In fact, AMD has a patent on *Hardware Based Return Pointer Encryption* (Kaplan, 2014) that was published in June 2014 and has a similar concept to the proposed idea.

The implementation of the shadow stack approach in hardware is a more challenging task as a logic must be implemented to find the appropriate entry in the shadow stack. The concept of a shadow stack would necessarily lead to a slow down because it involves memory access on every `call` and `ret`. To increase the performance, a cache that holds the most recent branches could be used.

Both of the presented security modules address only `ret`-based exploitation techniques like ROP. Therefore, they cannot protect programs that can launch an attack that e.g. relies on jump-oriented programming (Checkoway et al., 2010). As a consequence, a security module that protects against `ret`- and `jmp`-based exploitation is of interest. RIProtection is capable of monitoring arbitrary branches but program protection against `jmp`-based exploitation is a more challenging task because, in contrast to `ret`-based techniques, no corresponding instruction as `call` for ROP exists to determine if control is going to a legal branch target. Compared to the XOR and shadow stack security modules, an approach to mitigate jump-oriented programming would likely result in a similar performance evaluation because the overhead is for the most part caused by the instruction-level monitoring of RIProtection.

6 CONCLUSION

We presented RIProtection, a novel approach that utilizes the single step on branches hardware debug feature to provide instruction-level monitoring of branches. For its modular system, we developed a XOR and a shadow stack security approach where both operate whenever a `call` or `ret` instruction occurs. The chapter on the design and implementation of RIProtection showed that our generic concept can be adopted quite well by single modules. The following chapter evaluated RIProtection in terms of security, performance and compatibility. This concluding chapter summarizes the limitations of RIProtection has and proposes some ideas for how the implementation could be further improved.

As far as security is concerned, RIProtection can, in both operation modes, reliably prevent the exploitation of a buffer overflow that tries to overwrite the RIP with the raw value of an address found in an executable program section. As already discussed, the XOR approach might be circumvented by information leakage vulnerability. The shadow stack, however, is immune against memory disclosure and pro-

vides overall solid protection against `ret`-based exploitation.

The performance evaluation of RIProtection reveals that its usage in the field is highly dependent on the structure of the program and makes global statements on the expected performance of protected programs difficult. In any case, RIProtection introduces a significant performance impact.

In means of compatibility RIProtection provides excellent support for every x86_64 binary created from standard C with a compiler that respects conventions. The security module that should be used can conveniently be chosen by only altering the respective byte in the program ELF header.

ACKNOWLEDGEMENTS

This work was supported by DATEV eG as part of the research project “Software-based Hardening for Mobile Applications”.

REFERENCES

Bletsch, T., Jiang, X., and Freeh, V. (2011). Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC ’11, pages 353–362, New York, NY, USA. ACM.

Bulba and Kil3r (2000). Bypassing stackguard and stackshield.

Carlini, N. and Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC’14, pages 385–399, Berkeley, CA, USA. USENIX Association.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS ’10, New York, NY, USA. ACM.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In Prakash, A. and Sen Gupta, I., editors, *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin Heidelberg.

Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). Ropecker: A generic and practical approach for defending against rop attacks. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium*, NDSS’14. NDSS Association.

Corporation, I. (2014). Intel® 64 and ia-32 architectures software developer’s manual, volume 3 (3a, 3b & 3c): System programming guide.

Cowan, C., Beattie, S., Johansen, J., and Wagle, P. (2003). Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM’03, pages 7–7, Berkeley, CA, USA. USENIX Association.

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM’98, pages 5–5, Berkeley, CA, USA. USENIX Association.

Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC’14, pages 401–416, Berkeley, CA, USA. USENIX Association.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2011). Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS ’11, pages 40–51, New York, NY, USA. ACM.

Defense, M. S. R. . (2012). Introducing enhanced mitigation experience toolkit (emet).

Fratrić, I. (2012). Ropguard: Runtime prevention of return-oriented programming attacks.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC’14, pages 417–432, San Diego, CA. USENIX Association.

Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. (2012). Ilr: Where’d my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585.

Kaplan, D. (2014). Hardware based return pointer encryption. US Patent App. 13/717,315.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, pages 190–200, New York, NY, USA. ACM.

Mayer, U. F. (2011). Linux/unix nbench.

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. (2010). G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, pages 49–58, New York, NY, USA. ACM.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP ’12, pages 601–615, Washington, DC, USA. IEEE Computer Society.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA. USENIX Association.

Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2010). Return-oriented programming: Systems, languages, and applications.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA. ACM.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA. ACM.

Sotirov, A. and Dowd, M. (2008). Bypassing browser memory protections: Setting back browser security by 10 years.

van der Veen, V., dutt Sharma, N., Cavallaro, L., and Bos, H. (2012). Memory errors: The past, the present, and the future. In Balzarotti, D., Stolfo, S., and Cova, M., editors, *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*, pages 86–106. Springer Berlin Heidelberg.

Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA. ACM.

Xia, Y., Liu, Y., Chen, H., and Zang, B. (2012). Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA. IEEE Computer Society.