RECORD AND REPLAY BASED VIRTUAL-MACHINE INTROSPECTION FOR SYSTEM SECURITY

BY

YASSER HUSSEIN SHALABI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Hardware security features need to strike a careful balance between design intrusiveness and completeness of methods. Securing against attacks like Return Oriented Programming (ROP) requires frequent and expensive checks. Complete security defenses have been proposed yet modern systems are still vulnerable to ROP attacks. We provide complete security by decomposing the solution into two stages. The first stage raises alarms based on an imprecise, low cost hardware detector. The second stage applies complete methods in order to accurately distinguish real attacks from false alarms. This decomposition is enabled with Record and Deterministic Replay. The original execution is recorded and subjected to replay analysis as alarms are raised. In this way the Replay infrastructure can compensate for the occasional hardware imprecision.

We demonstrate this approach by applying it to thwart ROP attacks on the Linux kernel. We call the design RnR-ROPSafe. It reuses a simple Return Address Stack (RAS) as the hardware detector. The RAS is slightly modified to prevent corruption of the RAS due to multithreading and due to non-procedural returns—improving its performance as a ROP detector. Rare false positives due to underflows are eliminated via replay instead of hardware over-design. RnR-ROPSafe relies on two on-the-fly replayers: an always-on, fast *Checkpointing* replayer that periodically creates checkpoints, and a detailed-analysis *Alarm* replayer that is triggered when there is a threat alarm. We find that the first one has execution speed comparable to that of the recorder, and can be replaying all the time, while the latter has to handle only very few false positives.

To my mother and my big sister, for their love and support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CFI	Control Flow Integrity
DFI	Data Flow Integrity
JOP	Jump Oriented Programming
Malware	Malicous Software
RnR	Record and Replay
ROP	Return Oriented Programming

CHAPTER 1

INTRODUCTION

As security attacks are becoming more frequent and varied, there is increasing interest in augmenting processor and system hardware with security features. As a result, processor manufacturers have developed new hardware architectures, such as Intel's MPX [1], AMD's Secure Processor [2], and ARM TrustZone technology [3].

A general difficulty in this area is that security threats are continuously evolving, circumventing existing security defenses. What used to be an effective defense yesterday is less effective today. For example, to defend against code injection attacks, W⊕X [2, 4] features have been widely deployed in processors. They prevent the execution of data by enforcing the invariant that memory pages are either executable or writable, but never both. As a result, new attacks have appeared that do not need code injection. In particular, an attack based on code reuse called Return Oriented Programming (ROP) [5] is now the preferred technique. It builds attack code by chaining together multiple snippets of code from the victim program, allowing complete bypass of W⊕X defenses. Systems today remain vulnerable to such attacks despite the existence of provable prevention techniques [6, 7]. Such techniques cannot be implemented in modern systems either due to prohibitive performance costs [6, 7] or unsatisfiable requirements [8, 9]. Less expensive techniques have been proposed [10, 11, 12, 13, 14], but they can be undermined due to the incompleteness of their methods [15, 16, 17, 18, 19].

An intriguing primitive that can be used to defend against security threats is Record and Deterministic Replay (RnR) (e.g., [20, 21, 22]). With RnR, a workload's initial execution creates a log, which can be deterministically replayed on another machine. RnR has been used for security purposes, most often off-line, to provide insight into how and when an attack took place [20, 21]. It has also been used to support speculating past security checks [22].

In this thesis, we explore a novel approach to hardware security design where RnR is used to *complement* a hardware security feature—to offload intrusive checks and/or to eliminate imprecision. Specifically, security hardware is allowed to be less precise at detecting attacks and potentially report false positives; it relies on an on-the-fly replayer to transparently verify whether the alarm is a real attack or a false positive. This approach relies on two types of on-the-fly replayers: an always-on fast replayer that periodically creates state checkpoints of the monitored execution (*Checkpointing* replayer), and an analyzing replayer—triggered by an alarm—which starts from a checkpoint and analyzes the execution to determine whether the alarm indicated a real attack or was a false positive (*Alarm* replayer).

This thesis then applies this approach to thwart ROP attacks *on the kernel*—a challenging target to defend. We call the design *RnR-ROPSafe*. The micro-architecture that it builds on is the Return Address Stack (RAS). A RAS misprediction occurs for benign software, making the RAS an imprecise ROP detector as is. Hence, RnR-ROPSafe makes simple modifications to the RAS hardware to eliminate the vast majority of the false positives. The few remaining false positives are identified by the alarm replayer, thus minimizing hardware changes.

To evaluate RnR-ROPSafe, we execute a set of varied workloads on a Virtual Machine (VM) running Linux. We find that the RnR-ROPSafe architecture is an effective hardware-software co-design point. Thanks to the judicious RAS hardware extensions and hypervisor changes, the checkpointing replayer has comparable execution speed to the recorder, and can be replaying continuously. In addition, the alarm replayer has to handle only very few false positives.

Assumed System and Threat Models. ROP attacks can occur within the kernel or user contexts, and RepROP can secure both contexts. The target most difficult to secure is the kernel. We focus on evaluating RepROP's ability to detect kernel ROP attacks. The protected system (kernel and applications) runs inside a VM whose execution is continuously recorded. The recorded execution is then replayed, on a different platform, at which point it is checked for ROP attacks.

We assume that the attacker can launch a ROP attack against the kernel. We assume the host machine OS and hypervisor (recording and replaying

machines) to be benign and that they can safeguard against compromised guest VMs.

CHAPTER 2

BACKGROUND

2.1 Return Oriented Programming

The objective of attackers is to execute malware on a victim machine. In the past, attackers injected malware machine code into memory allocated for data. Later, execution is hijacked to fetch instructions that corrupted memory. The W⊕X [2, 4, 10, 23, 24] policy was designed to counter this specific attack vector. By enforcing that memory pages are either executable or writable—but never both—malware injected into memory can no longer be executed. To bypass W⊕X, "Code Reuse" based attacks were proposed. For these attacks, malware instructions are recovered from existing code instead of injected into memory. Return Oriented Programing (ROP) [5] is the dominating example of this approach.

Conceptually, an ROP attack executes multiple snippets of code from the victim program or software environment (e.g. libc) called *Gadgets*. Each gadget is terminated with a return—a branching instruction whose target is popped from the software stack. The attacker first loads into the software stack the addresses of the desired gadgets. Then, to trigger the attack, control flow is forced to the first gadget. As the first gadget terminates, its return instruction pops the next entry from the software stack, redirecting execution to the next gadget. Thus, by writing onto the stack the addresses of gadgets, the attacker can stitch together a desired sequence of gadgets required to achieve the desired malicious effects.

This type of attack is dangerous for several reasons. First, it has been shown that the right set of gadgets can construct a Turing-complete language [5] enabling an ROP compiler to translate malware from any other Turing-complete language (like C) to one expressed entirely in gadgets. Second, this attack bypasses the prevalent W⊕X defense techniques, because

there is no data being written and then directly executed: the malware executes existing code. Finally, any simple bug in the code enabling attackers to corrupt the stack can trigger the execution of a sophisticated chain of gadgets.

Figure 2.1 shows an example of an ROP attack that exploits a buffer overflow to execute three gadgets. We use a buffer overflow bug *for simplicity*; any bug that allows stack modification can be used to launch an ROP attack.

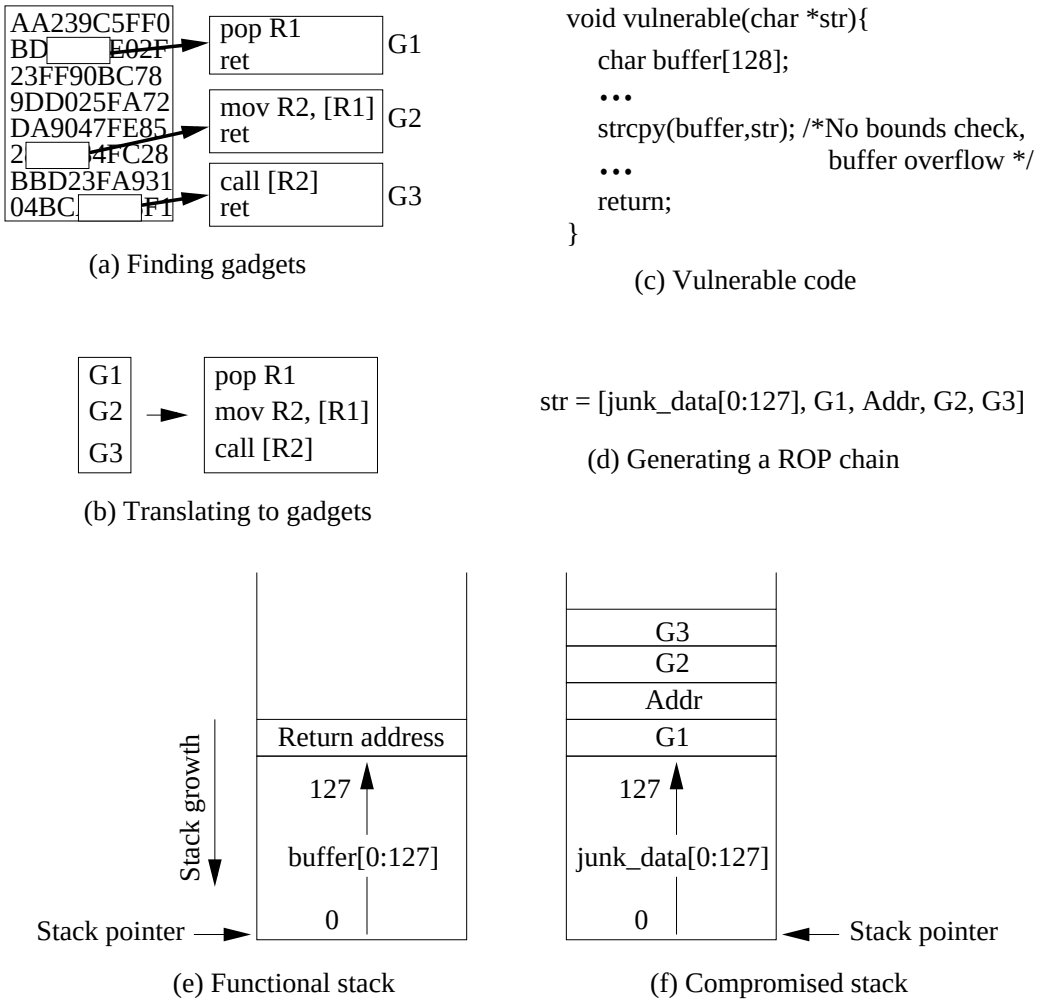


Figure 2.1: Example of Return Oriented Programming attack.

In Figure 2.1(a), the executable is scanned for instances of the return (*ret*) instruction. We decode a few bytes before three returns creating three gadgets (G1-G3). Executing the three gadgets in sequence is equivalent to executing the code in Figure 2.1(b). The code will result in a subroutine call to a function pointer loaded from a memory location stored on the stack. If this was executed during kernel execution, this can be a call to code giving

the user root privileges.

Figure 2.1(c) shows code that is vulnerable to a buffer overflow attack. The code copies a string into a 128-byte buffer without verifying that it can fit in the buffer. Figure 2.1(d) shows how a payload can be constructed to exploit this code to execute ROP malware. Figures 2.1(e-f) show the benign state of the stack and its state after being corrupted by the malicious input string. Now, returning from the vulnerable function takes us to G1, which will pop Addr into R1 and then return. The return will lead to G2, which will load into R2 and return to G3. Then G3 will perform the call.

ROP attacks can be detected with what is known as a *Shadow Stack*. The shadow stack operates with typical "Last in First Out" semantics. Whenever a call instruction is encountered, the address of the instruction following the call is pushed to the top of the shadow stack. On the other hand, return instructions pop from the shadow stack. ROP attacks can be detected anytime the return address used by the processor mismatches with the one popped from the shadow stack.

Unmet challenges have prevented the utilization of shadow stacks in practice. First, the validity of this technique hinges on the integrity of the shadow stack. Hence, it must be secured against the very software it protects—a non-trivial task. Also, codes can be highly nested (e.g. recursive), multi-context (e.g. kernel), or imperfectly nested (e.g. error/exception handling). Each of these requires special handling.

2.2 ROP Still Possible on Current Systems

Many proposals focus on protection against ROP attacks [6, 7, 12, 13, 25, 26, 27, 28, 29, 30, 31, 32]. However, these techniques remain unused. The reasons are as follows.

HW Intrusiveness. Some solutions [26, 28, 30] require intrusive hardware changes. SRAS [26] adds a secure hardware RAS to verify the return targets. System memory must back up the secure RAS, necessitating additional read/write ports. The PUMP [30] processor implements support for general metadata propagation. This can be used to implement various safety checks, including Control Flow Integrity (CFI). However, each stage of the pipeline must be changed to support tag storage and/or rule execution. REV [28]

hashes the instruction sequences within a basic block to verify a program's control flow. An additional 32KB first-level cache dedicated for caching signatures is required to avoid prohibitive slow-downs.

SW Impact. The completeness of instrumentation-based solutions such as [6, 7] is attractive. However, *securely* maintaining the shadow RAS at call/ret boundaries via binary instrumentation adds overheads that exceed 100% [7]. Other approaches [29, 33] propose recompiling the kernel and application code to target a secure virtual instruction architecture. This architecture is emulated by a compiler-based virtual machine (similar to the Java Virtual Machine). Aside from the performance costs, source code is not always available, which limits the applicability of this technique.

Completeness. Proposals looking for an alternative to CFI-based solutions propose monitoring execution properties for indicators of ROP execution [12, 13]. However, benign programs may also trigger these detectors. Also, ROP payloads can blend their signature to match that of benign code to evade detection [17, 34]. Probabilistic defenses [11, 35, 36, 37] use a secret value to encrypt or randomly arrange code/data. This significantly complicates the attacks, but other vulnerabilities [19] can leak secrets to negate the defenses.

2.3 Return Address Stack

Modern processors use a hardware structure called Return Address Stack (RAS) to predict the target of return instructions. When a procedure call instruction executes, the hardware pushes the address of the instruction that follows it into the top of the RAS. When a return instruction is decoded, the hardware pops the entry at the top of the RAS and uses its value as the predicted target of the return. In most cases, the prediction is correct. The IBM POWER7 [38] and POWER8 [39] processors have a RAS with 32 and 64 entries, respectively.

ROP attacks cause RAS mispredictions. Assume that the return from gadget G1 to gadget G2 in the example was correctly predicted by the RAS. This would require that a call was executed within gadget G2, so the address of G2 gets placed on the RAS. However, G1 comes before G2. On the other hand, a RAS misprediction *cannot alone* be used as an indicator of ROP attacks because the RAS sometimes mispredicts in the course of benign

program execution.

2.4 Record and Replay

Record and Replay (RnR) of workloads is a popular architectural technique (e.g., [40,41,42,43,44,45,46,47,48,49,50]). As a workload runs, RnR records all the non-deterministic events that can affect the execution and stores them in a log. Later, in a potentially different platform, the workload is re-run. At this time, the system injects the recorded events at the correct times, enforcing a deterministic execution (*Replay*). Typically, the non-deterministic events are the inputs to the workload and, in parallel programs, the memory-access interleaving.

RnR can be done at different abstraction layers. In this work, we use VM-level RnR [20,41,42,51,52]. Moreover, we consider uniprocessor hardware. As a result, the sources of non-determinism are interrupts raised and data copied by virtual devices into the guest machine. We also assume the widely used model of hypervisor-mediated I/O, as used in Xen [53] or Qemu [54]. These assumptions are not necessarily limitations, as RnR approaches compatible with multiprocessor [47] and virtualized I/O [55] exist.

There are several papers that investigate the use of RnR in a security-related scenario [20, 21, 22, 50, 51, 56]. ReVirt [20] shows an example of using VM-level RnR for post-facto offline analysis of a time-of-check to time-of-use race conditions in the Linux kernel. IntroVirt [21] explores using VM-level RnR to determine if systems were previously exploited once zero-day attacks are discovered. Speck [22] explores using a combination of OS-level speculation and program-level RnR to remove security checks from the critical path of a program. ParanoidAndroid [50] and Secloud [56] explore the possibility of maintaining replicas of mobile devices in the cloud, and perform program-level RnR in the cloud. Finally, Aftersight [51] suggests using VM-level RnR to perform online dynamic analysis of a system's execution. However, it does not address several important hardware-software design issues of such a model, including a key contribution of our work: separation between the fast checkpointing replayer and the exhaustive alarm replayer. We discuss the details in Section 9.

CHAPTER 3

RNR-ROPSAFE: THWARTING ROPS

We propose using a combination of existing processor hardware and well-known RnR techniques to provide complete protection against ROP attacks without the prohibitive costs of instrumentation-based approaches. Figure 3.1 shows the organization of our system, called *RnR-ROPSafe*. On the left side, a workload runs on a *Recorded* VM. Its hypervisor records all the non-deterministic events of the execution in a software log. Recording adds only modest overhead—low enough for the execution not to be noticeably slower. Note that we record at the VM level to also protect the operating system.

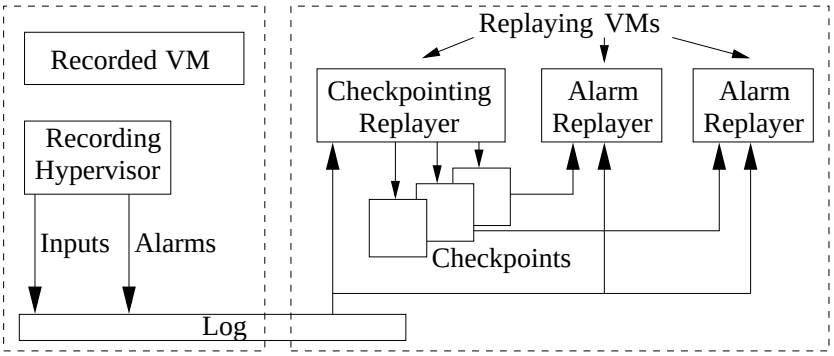


Figure 3.1: RnR-ROPSafe organization.

The designer has augmented the hardware in the recorded VM (e.g., processor and memory system) with support to detect a certain class of attacks. When the combined operation of this hardware and the recording hypervisor detect the attack, the hypervisor inserts an alarm marker in the log. At this point—and depending on the risk tolerance of the workload—the recorded VM may be stopped until the alarm is analyzed, or allowed to continue.

On the right side, one or more *Replaying* VMs re-execute the workload natively. They use the log to inject all the non-deterministic events. As a result, their execution deterministically follows the original one.

3.0.1 What Record and Replay Offers

The addition of RnR provides—in addition to its traditional usages—three security benefits.

Robustness. Perfect hardware detection accuracy often necessitates intrusive hardware. By separating alarm detection from attack verification using RnR, this can be alleviated. Now, the only requirement for the hardware security detectors is to support the common case. False alarms and rare corner cases are instead handled by software-based replay. Thus, RnR restores robustness to a system stack that includes imprecise security hardware.

Flexibility. RnR is intrinsically flexible. As attackers devise new attacks, defenders can add new analysis techniques to the alarm replayer to detect and/or prevent them. Defenders can even run parallel alarm replayers tracking different types of attacks at the same time. This is simplified because the analysis is in software.

Execution Auditing. RnR allows detailed analysis of executions. The execution context causing the alarm can be replayed to audit the code and data state. This is a general mechanism for identifying security violations by auditing sensitive flows in the system.

3.0.2 RnR-ROPSafe Modes of Execution

In RnR-ROPSafe, monitored recording consists of normal execution, while transparently recording all the non-deterministic inputs in a log, and transparently monitoring safety violations. If a violation is found or suspected, an alarm entry is inserted in the log. A key detail is that, in order to claim complete protection, the detector must catch all potential threats. In other words, false negatives are impossible.

In RnR-ROPSafe, the replay execution can be performed in two ways. One way is *Checkpointing Replay*. Such replay runs at recording-like speeds. It uses the log to deterministically replay the workload while creating state checkpoints at regular intervals. When an alarm marker is found in the log, the checkpointing replayer launches the execution of an alarm replayer from a recent (typically the latest is sufficient) checkpoint. Once old checkpoints and log entries are verified they can be discarded to save storage.

A second type of replay is *Alarm Replay*. Alarm replay replays log en-

tries from a given checkpoint until an alarm, while performing an extensive, attack-specific analysis of the replayed execution. Its goal is to resolve an alarm, either to show that it is a false positive or to verify and characterize the attack. It can be much slower than the recording execution.

Typically, alarms are rare events. Therefore, we envision one replaying VM to continuously run the checkpointing replayer. This replayer simply replays the workload at a speed comparable to the recorded execution, consuming the log, and periodically creating state checkpoints. If the checkpointing replayer finds an alarm marker in the log, it starts an alarm replayer in another VM. The alarm replayer deterministically replays from the latest checkpoint until it finds the alarm marker. This replay performs a detailed analysis, characterizing the attack to identify the vulnerability it exploited and to assess the extent of the damage.

This approach can be applied to protect against different attacks (Section 6). In this thesis, we focus on ROP attacks.

In the next section, we explain the techniques of RnR-ROPSafe that implement ROP protection flexible enough to protect against ROP attacks [57] on the kernel.

3.1 Main Idea in RnR-ROPSafe

The basic architecture primitive that can help detect ROPs is the RAS (Section 2.3). The RAS stores the addresses of the predicted targets of return instructions. At every call instruction, the hardware pushes the return address onto the RAS; at every return, the hardware pops the RAS and uses its address to predict the return target. Hence, a ROP attack causes RAS mispredictions.

To use RAS mispredictions to prevent ROP attacks requires that there are no false negatives. Fortunately, execution of ROP payloads is guaranteed to cause RAS mispredictions, making false negatives impossible. Furthermore, for this detector to be useful, false alarms should be infrequent. However, there are a few major sources of imprecision in the basic RAS operation. We will explain these sources with Linux kernel examples, where we found them to be most common.

First, there is the effect of multithreading. In a multithreaded environ-

ment, when the kernel switches from Thread i to Thread j, it leaves entries belonging to Thread i on the RAS. When executing code in Thread j (or other threads scheduled after i), these entries might be incorrectly popped and used for prediction. If so, not only will Thread j encounter mispredictions, but also Thread i's entries will no longer be available for their use after i is rescheduled. Hence, there will be mispredictions and false positive ROP alarms.

A second effect is non-procedural returns in the kernel. Sometimes—e.g., during a context switch—the kernel inserts an address into the software stack, which will later be used by a return instruction as target. Since there was no prior call from that address, the RAS will not contain a corresponding entry and will mispredict.

RAS underflows are a third source of imprecision. If the code executes many nested procedure calls, the RAS may evict some of the earlier return addresses. Later, when the execution returns from the inner calls and tries to pop entries corresponding to the outer calls, the RAS will be empty (*underflow*) and will mispredict.

Imperfect nesting in procedure calls is another reason for RAS mispredictions—a situation where a procedure is called but never returned from. Within the kernel, these are rare events that typically only take place as part of bug recovery processes in the kernel. When the kernel execution encounters a recoverable bug, it initiates a recovery process, as part of which it terminates the current thread of execution, leaving all the RAS entries of the current thread orphaned. For user-mode code these occur more commonly—for example, exception handling is implemented using setjmp/longjmp.

These effects show that the RAS is an imprecise detector of ROPs and, therefore, unusable as is. For RnR-ROPSafe to use it as the initial indicator, two steps are needed. First, we robustify the RAS detection capability with simple hardware and hypervisor support. The goal is to minimize the false positive rate. To completely eliminate false positives requires disruptive software changes and intrusive hardware changes. The second step is to use deterministic replay to distinguish the false alarms from the real attacks—and to characterize any detected ROPs.

An alarm replayer is invoked when there is an alarm. Since it has to provide a response quickly, replay cannot start from the beginning of the VM execution. Instead, it starts from a nearby checkpoint created by the

checkpointing replayer.

In the following, we describe the components of RnR-ROPSafe and the steps taken to protect the kernel despite its unique RAS challenges.

3.2 Basic Design

As shown in Figure 3.1, the workload (applications + kernel) runs in a *Recorded* VM. As it runs, the hypervisor creates an input log that is sent to and consumed by a *Replaying* VM.

The traditional behavior of the RAS is slightly augmented in this *basic* design of RnR-ROPSafe. Specifically, if we are executing a return instruction in kernel mode, and a mismatch is found between the predicted target in the RAS and the actual return target, the hardware sets a flag (called *ROP_Alarm*) in the ROB entry for the return instruction. When a return reaches the ROB head, if the *ROP_Alarm* bit is set, a VM exit is triggered. Then, the hypervisor inserts a ROP alarm entry in the input log. Depending on its configuration, the hypervisor may or may not stop the recorded VM until the alarm is fully processed in the replaying VM.

In the meantime, the checkpointing replayer is consuming the log to create regular checkpoints. If it finds the alarm entry in the log, it triggers the execution of the alarm replayer, starting from the most recent checkpoint. The alarm replayer determines whether it is a false alarm or a real ROP. It is possible—but rare—that an older checkpoint is needed to confirm/refute the attack.

This basic RnR-ROPSafe design will not miss an attack, since a ROP has to execute a return instruction. However, this is insufficient, as a large source of false alarms remains: those due to multithreading and non-procedural returns. Alarms involve costly VM exits and lengthy replay, incurring high overheads with this design. Next, we extend this basic design to reduce its false positives.

3.3 Supporting a Multithreaded Environment

In a multithreaded environment, a thread might be de-scheduled—e.g., due to pre-emption or performing a blocking operation—while executing in kernel mode. The return address entries left by this thread on the RAS might be popped and used (incorrectly) by subsequent threads, and this thread itself might pop and use RAS entries belonging to other threads once it is re-scheduled. The result is RAS mispredictions and a large source of false ROP alarms.

To address this problem, RnR-ROPSafe extends the processor hardware. On a context switch, the hardware automatically saves the current RAS into a safe memory area, and restores the RAS state as needed for the upcoming running thread. The hypervisor helps by setting a hardware pointer to point to the correct memory area to move data out and in. For that, we augment the set of structures that the micro-coded virtualization hardware already saves and restores at the context switch to also include the RAS.

The structures are shown in Figure 3.2. The software structure in memory is an array of backed-up RASes (*BackRAS*). Each entry belongs to a thread, and has a RAS and a counter with the number of entries in the RAS. The counter is needed to know the number of entries that need to be read later on. The processor hardware includes a pointer (*BackRASptr*) that points to the backed-up RAS of the currently running thread. The pointer is set by the hypervisor and used by the hardware to access the correct BackRAS entry.

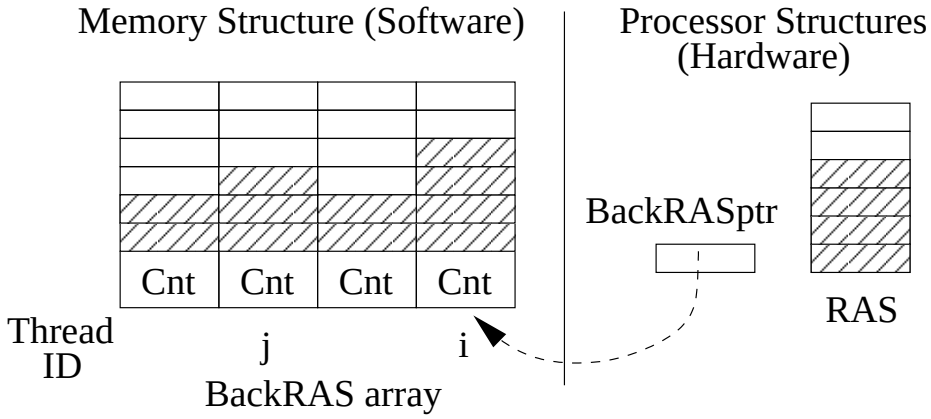


Figure 3.2: Structures used to support multiple threads.

The logic used is shown in Figure 3.3. On a context switch, as part of the transition to the hypervisor, the hardware saves the RAS to the entry to

which BackRASptr is pointing. In addition, it computes and stores the count of saved entries. Our measurements show that a transition to the hypervisor already takes about 1,000 cycles. We estimate that backing up the RAS will add about 20% more time. Later, when the hypervisor runs, it changes BackRASptr to point to the entry for the new thread. Finally, as part of the transition back to the guest, the hardware reads the correct BackRAS entry into the RAS. We also estimate about 20% additional overhead.

To program the BackRASptr, the hypervisor needs to be informed of context switches in the guest kernel and identify the new thread to be scheduled. Section 4.2 explains how this can be done without modifying the guest kernel.

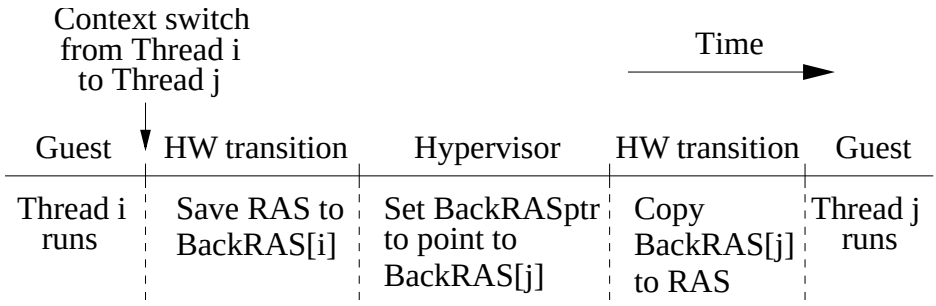


Figure 3.3: Algorithm and timeline to handle multiple threads.

With this support, when a thread is scheduled, it will find its correct state in the RAS, thus eliminating many false alarms.

3.4 Supporting Non-Procedural Returns

Sometimes, the kernel uses the return instruction as an indirect branch. Specifically, it inserts an address into the software stack, and then executes a return that uses that address as target. Since there was no corresponding procedure call, the RAS did not push an entry, and will mispredict. Consequently, in these cases, the RAS should not be popped, as doing so would corrupt the RAS state.

In the Linux version we use, this use of returns outside of the procedural abstraction occurs once, when a context switch is complete. At that point, right before launching the next thread, the kernel executes such a return in order to start executing code on behalf of the new thread. This code is written in assembly and directs the control flow to a few well-defined locations in the

kernel code. These locations complete the task switching based on whether it involves forking a thread, executing a kernel thread, or rescheduling a task.

To address this problem, RnR-ROPSafe extends the processor hardware with a table of "whitelisted" addresses. For our Linux version, there is a single-entry return whitelist (*RetWhitelist*) with the PC of the single return used as indirect branch, and a target whitelist (*TarWhitelist*) with the PC of the three instructions that can be the target of this return. During return address prediction, if a return and its target PC match entries in the tables, then the RAS is not popped and no alarm is raised. The potential for these lists to bypass our security checks is limited as they are only writable by the hypervisor.

The logic used and its timeline are shown in Figure 3.4. When an instruction is decoded and identified as a return, the hardware checks if its PC is in the RetWhitelist. If so, the RAS is not popped and a *Whitelisted* flag in the return's ROB entry is set. Later, when the target address is accessed, if the Whitelisted flag is set, the hardware checks if its PC is in the TarWhitelist. If it is not, the ROP_Alarm bit is set in the ROB entry for the return instruction. When the return reaches the ROB head, if the ROP_Alarm bit is set, a VM exit is triggered.

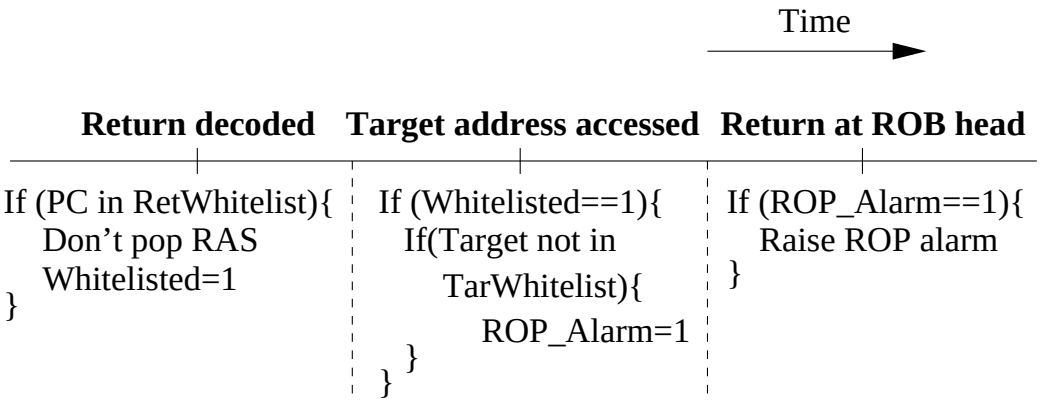


Figure 3.4: Timeline to handle non-procedural returns.

The whitelisted addresses can be found by analyzing the binary image of the guest kernel. Then, the hypervisor can populate RetWhiteList and TarWhiteList using the identified addresses when entering the VM as explained in Section 4.1.

3.5 RAS Underflows and Imperfect Nesting

It is possible that the kernel executes many nested procedure calls, causing the RAS to evict some of the earlier return addresses. In this case, when the hardware accesses the RAS in a return instruction, it may find it empty. This will cause a RAS misprediction, and will trigger a ROP alarm when the return instruction reaches the ROB head. These alarms are likely to be false positives.

RnR-ROPSafe could prevent this problem by adding more entries in the RAS or opportunistically saving/restoring the RAS. However, this requires expensive hardware that is rarely used. Hence, RnR-ROPSafe lets these events raise ROP alarms, and relies on the replayer to identify them as false positives. Since the replayer models an unbounded RAS, it has no underflows and can filter out such false positives.

Similarly, we let the processor raise ROP alarms for mispredictions due to imperfect nesting. Such alarms are easily filtered out by our alarm replayer. It should be noted that these events are very rare—we encountered only a few underflows in our benchmark runs.

3.6 Replaying Platform

The input log is passed to another platform, where the VM execution is deterministically replayed in one or several guest VMs. At all times, there is at least one VM running the checkpointing replayer. In addition, at certain times, there may be one or more VMs running alarm replayers. As indicated above, the checkpointing replayer consumes the log as it is received, and creates checkpoints at regular intervals. When it finds a ROP alarm marker in the input log, it initiates an alarm replayer at the immediately preceding checkpoint. The alarm replayer carefully analyzes the execution until it reaches the alarm marker, to determine if it is a true ROP or a false alarm.

3.6.1 Checkpointing Replayer

To understand the operation of the checkpointing replayer (CR), we first describe the contents of a checkpoint. Figure 3.5 shows three checkpoints. Each

checkpoint has three components. The first one is all the pages with the VM state. These include the memory pages plus a page with the processor state (PC, stack pointer, and the rest of the registers at the time of checkpoint). They also include the virtual disk image contents. This is the state that the VM being recorded wrote to the virtual disk. We need to checkpoint it because, if the execution later reads this data, the data will not appear in the input log. Note, however, that the state checkpoints are *incremental*. Since we take regular checkpoints, a given checkpoint keeps copies of only the pages that have been modified since the previous checkpoint; for each unmodified page, it keeps a pointer to the page in the latest checkpoint that modified it.

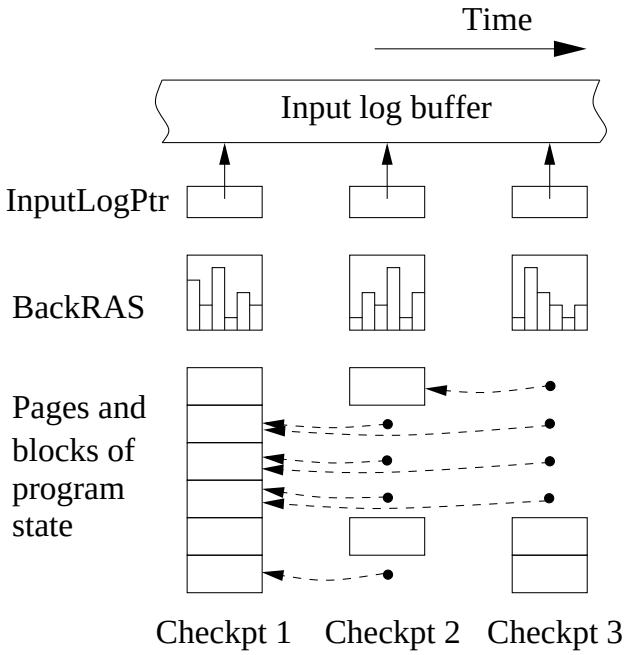


Figure 3.5: Checkpoints created by the checkpointing replayer.

The second component of a checkpoint is a pointer to the input log buffer (*InputLogPtr*). The pointer points to the next input log entry to be processed after the checkpoint. Finally, the last component is the BackRAS at the time of the checkpoint. We will see in Section 3.6.2 that the alarm replayer needs this state.

The processor hardware on which the CR runs operates slightly differently than how it operated for the recorded VM—in order for the hardware to help create the checkpoints. Specifically, the hardware dumps the RAS into the BackRAS not just at context switching points, but also at every VM exit

while in the kernel. This ensures that, at the point of the checkpoint (which is also a VM exit), the CR has the up-to-date state of the BackRAS to stash in the checkpoint. There is no restoring of the RAS at non-context switching VM exits. As indicated in Section 3.3, we estimate a VM exit and subsequent entry to take $\approx$ 2,000 cycles, and saving the RAS to be $\approx$ 10% of it.

A second modification is that the processor hardware's ability to trigger ROP alarms is disabled. This is because replay does not create alarms.

With this background, we now describe the CR operation. The CR executes the recorded VM, in a deterministic manner, while consuming the input log. After every checkpoint, all the pages comprising the VM's memory and disk state are marked as *copy-on-write*. When a page is modified for the first time since the last checkpoint, a copy is made and used from now on. When the CR decides to create a checkpoint, it interrupts the processor and dumps the processor state (PC, stack pointer, and all registers) into a memory page. The RAS is automatically saved as part of the VM exit. The CR then creates the checkpoint by saving: (1) all the modified memory pages and disk blocks, together with pointers to the unmodified ones, (2) the current BackRAS, and (3) the current InputLogPtr. Then, the CR restores the processor state, marks all pages copy-on-write, and continues execution.

The CR regularly recycles checkpoints. However, it can only recycle a memory page or disk block if it is not pointed to by a later checkpoint.

3.6.2 Alarm Replayer

The goal of the alarm replayer (AR) is to determine whether an alarm is caused by a ROP or if it is a false alarm. If the former, the AR immediately provides the state of the processor, memory, and disk at the point of the ROP attack.

The processor hardware on which the AR runs neither dumps the RAS state nor triggers ROP alarms. Both capabilities are disabled because they are not needed.

The AR VM starts its execution by initializing the VM state using a checkpoint. It marks all the pages pointed to by the checkpoint as copy-on-write to avoid modifying the initial state. Then, it reads the checkpoint's BackRAS into its own software data structure that it uses to simulate the RAS. Next, it

loads the processor state from memory into the processor registers. Finally, it starts execution, reading from the log starting from the InputLogPtr.

The AR executes the recorded VM natively, in a deterministic manner, consuming the input log until it reaches the alarm marker. The AR models unbounded, per-thread RAS structures in software. As such, the AR traps every call and return instruction, inducing VM exits and transferring control to the hypervisor. Then, the hypervisor runs in software the basic RnR-ROPSafe algorithm plus its extensions for multithreading and non-procedural returns. The simulated RAS cannot underflow as it is unbounded.

Once the AR encounters the alarm in the log, it checks whether the RAS mismatch can only be explained as an ROP attack. If so, an expert can carefully study the execution state— by performing multiple replays with increasingly targeted instrumentation—to glean information about the attack. Chapter 5 shows an example. Note that this design readily accommodates running multiple ARs to analyze multiple ROP alarms in parallel.

CHAPTER 4

IMPLEMENTATION ISSUES

Following Intel's VT terminology, we use VMCS (VM Control Structure) to refer to the in-memory control structure through which the hypervisor communicates with and configures the virtualization hardware. We use *VMEnter* to mean transferring execution from the hypervisor to the VM, and *VMExit* to mean the opposite transfer.

4.1 Hardware Implementation of RnR-ROPSafe

The hardware changes required to implement RnR-ROPSafe are minimal. First, we need to allow the hypervisor to program the contents of BackRASPtr, RetWhiteList and TarWhiteList. This can be done by extending the VMCS with three new fields. The microcoded logic of VMEnter reads these fields to program these three processor hardware structures. In addition, it uses the BackRAS entry pointed to by BackRASPtr to populate the RAS. Similarly, on a VMExit, its microcoded logic dumps the RAS content into the active BackRAS entry.

The second set of hardware changes has to do with the interaction between the RAS hardware and in-window speculation. In a conventional processor, RAS entries can be pushed and popped by speculative call and return instructions that may be squashed later—e.g., due to a branch misprediction or an exception in upstream instructions. However, when our hardware dumps the RAS content on a VMExit, we need to dump only those entries that correspond to architecturally retired instructions. This requires the following changes.

In conventional processors, the RAS is typically implemented as a circular buffer with a single pointer, *Top*, pointing to the most recent entry. Call instructions increment this pointer and write to the top entry (push); re-

turn instructions read from the top entry (pop) and decrement this pointer. Both instructions typically modify the RAS while still speculative (i.e., before reaching the ROB head). In RnR-ROPSafe, we augment this design by adding two more pointers, *NonSpec_Top* and *NonSpec_Bottom*, which respectively point to the youngest and oldest RAS entries that correspond to architecturally retired (non-speculative) instructions. Then, on a VMExit, the hardware only dumps the RAS entries between these two pointers.

In our design, the RAS continues to use the Top pointer for its normal push and pop operations and for making predictions. NonSpec_Top is incremented (decremented) by the retirement of call (return) instructions. Therefore, it always points to the architecturally accurate top of the RAS. NonSpec_Bottom is used to deal with the cases where a call instruction, which is later squashed, pushes to an already full RAS and overwrites an older, non-speculative RAS entry. Such overwriting can only happen when the RAS is full and, with a big-enough RAS, will be a rare event. To avoid having to recover the lost RAS entry, we increment NonSpec_Bottom when the entry that it points to is overwritten by a RAS push. This way, the net effect of overwriting the RAS entry will be an underflow (when the return corresponding to the overwritten entry is executed), instead of polluting the RAS dump with speculative content.

4.2 Hypervisor and RAS Hardware Interaction

In this section, we explain how the hypervisor is modified to use the hardware extensions of RnR-ROPSafe.

Programming BackRASPtr on a Context Switch

In RnR-ROPSafe, the hypervisor needs to interpose on all context switches in the guest kernel during both recording and replay. In Linux, there is a single instruction where the stack pointer is changed from pointing to the current thread's stack to the next thread's stack. By setting a trap on this instruction, the hypervisor forces a VMExit when the guest executes this instruction. As part of the VMExit's microcoded logic, the hardware dumps the RAS into the memory location pointed to by BackRASPtr.

Once the VMExit is complete and the control is transferred to the hypervisor, it can use a technique known as *VM introspection* to inspect the state of the guest OS. This allows the hypervisor to identify the next thread to be scheduled. In Linux, a thread's descriptor (called *task_struct*) can be easily found if the thread's stack pointer is known. Since we set the trap on the instruction that changes the processor's stack pointer, we can find the next thread's stack pointer by examining the register content of the VM, which is available in the VMCS after a VMExit. Using this stack pointer, we find the corresponding *task_struct* descriptor in the VM's memory, and from that descriptor, read the next thread's ID.

The hypervisor stores the BackRAS in a memory area inaccessible to the guest machine. It stores it as a hash table mapping a thread's ID ("key") to its BackRAS entry ("value"). Using this organization, once the thread ID is found, the hypervisor checks the map to determine if there is already an entry for that thread. If not, it means that the next thread is executing for the first time, and the hypervisor allocates a new entry for it. In either case, the hypervisor sets the BackRASPtr field of the VMCS to point to the BackRAS entry.

4.2.1 Recycling BackRAS Entries

In Linux, threads are constantly being created and killed, and their IDs may be reused. To keep the BackRAS consistent, we need to remove from the BackRAS a thread's entry when the thread is killed and its ID can be reused. Similarly to the case of context switching, the hypervisor sets a trap on the function that implements this functionality in the guest kernel to force a VMExit when it is executed. At that point, the thread ID can be found by introspection and then used to delete the corresponding BackRAS entry.

4.3 Complexity Discussion

One of the primary motivations for our work is finding a solution that is largely compatible with commodity systems and ensures detection of ROP payloads. The need for strong guarantees necessitates fine-grained CFI [6] and not probabilistic measures [11, 35]—without prohibitive costs. It is also

important that our solution suffer neither the overheads of instrumentation-based approaches [7] nor the intrusiveness of hardware-based approaches [25, 26, 28].

As discussed in Section 3.1, most of our hardware for verifying return instruction targets reuses the existing hardware RAS along with its read/write ports. The actual hardware added by RnR-ROPSafe includes: the two whitelist tables, a bit per ROB entry to mark an alarm, the BackRASptr register, and two pointers in the RAS. The maintenance of the BackRAS array in Figure 3.2 is performed in microcode, as the processor executes VM entries and exits. The requirement for RnR can be considered the most substantial change. However, RnR is well understood and accepted as a useful primitive for debugging and program analysis [21, 22, 51, 58].

CHAPTER 5

APPLICATION FOR KERNEL ROP DETECTION

We built and ran the ROP attack of Figure 2.1. In the recording VM, as the workload calls the *vulnerable* procedure of Figure 2.1(c), the hardware pushes into the RAS the address of the instruction at the call site (call it *CallSite*). This is the same address that is stored next to the buffer in the software stack of Figure 2.1(e). After the malicious string copy, the software stack becomes Figure 2.1(f). As the program executes the return of the *vulnerable* procedure, the hardware uses the RAS to predict that execution will transfer to *CallSite*. In reality, the target of the return is resolved to be the address of gadget G1, as shown in Figure 2.1(f). This mismatch causes the recorded VM to raise an alarm.

The recorded VM hypervisor then inserts an alarm marker in the log and may decide to stall the VM. When the checkpointing replayer sees the alarm marker in the log, it starts an alarm replayer from the most recent checkpoint. As the alarm replayer executes, it models the RAS in software. At the point of the alarm, it observes the mismatch between the return's predicted target (in the RAS) and the actual target (in the software stack), hence declaring a ROP attack.

At this point, the hypervisor performs an analysis of the system. It can use VM introspection to analyze the VM state, which has not been polluted by the execution of any gadget. It can also invoke additional replays farther back in time to perform a deeper analysis of the system.

One question replay analysis can answer is: how was the attack possible to begin with? The hypervisor uses the return instruction that caused the alarm to determine that the attack occurred in the *vulnerable* procedure. It uses the address at the top of the RAS to determine the call site. An analysis of the *vulnerable* procedure can conclude the presence of buffer overflow. Another question is who attacked the machine? The hypervisor can determine the thread ID of the current thread, extract which users are logged in, and

determine which network connections are established. Yet another question is: what did the attacker do? An analysis of the software stack can reveal the gadgets used by the attacker. In this case, they did not execute. If they did, the hypervisor can use VM introspection to analyze what files were touched, what sockets were utilized, and what processes were forked [59]. This information is easy to get now because the workload is not running.

CHAPTER 6

FUTURE EXTENSIONS

The RnR primitive we utilize can empower other security defenses. The key advantage is that the replay can be used to compensate for imprecise first lines of defense. The replay can distinguish the false alarms by using additional information or complete methods.

For example, Table 6.1 considers jump-oriented programming (JOP) [60] and denial of service (DOS) [61]. The table shows the alarm trigger, the hardware needed, and the role of RnR. For example, JOPs can be detected with a hardware table of addresses of the most common functions. An indirect branch target is compared to the table and is legal if the target is the first instruction of a function, or any target within the current function. Otherwise, an alarm is triggered, and the RnR will check against all the remaining functions. A DOS attack on the OS can be detected with a counter that increments every time the kernel performs a context switch. If the counter has not increased much for a while, an alarm is triggered, and the RnR analyzes and identifies the code that has dominated the system's execution time.

Table 6.1: Potential uses of the RnR based approach.

Attack	Alarm Trigger	Hardware Needed	Role of RnR
Kernel ROP	RAS misprediction	Dump the RAS, BackRASPtr, Whitelist	Perform *kernel-compatible* shadow stack algorithm
Jump Oriented Programming (JOP)	Stray indirect branch	Table of addresses of most common functions (entry and end addresses)	Verify control flow integrity of calls to less common functions
Denial of Service (DOS)	Kernel scheduler inactivity	Counter of number of context switches	Identify reason for low switching frequency

CHAPTER 7

EXPERIMENTAL SETUP

To evaluate RnR-ROPSafe, we use two evaluation environments. The first one evaluates the performance of our recording and replaying modes. For this, we use Insight [62], a VM RnR tool based on a modified Linux KVM hypervisor and QEMU devices. Since the KVM hypervisor can leverage Intel VTx extensions to virtualize the processor in hardware, the performance numbers from this setup are representative of real-world machines.

The second environment evaluates the correctness of our techniques and the functional characteristics of our proposed hardware. For this, we use QEMU in emulation mode. In this mode, QEMU also emulates the processor using dynamic translation of the systems software. This mode makes it easy to simulate our hardware and evaluate its function.

Table 7.1 shows the system configuration we used for our performance evaluation, and Table 7.2 shows our benchmarks.

Table 7.1: System configuration for performance evaluation.

Host machine	
CPU: Xeon E3-64bit,4-cores,3.1GHz	Memory: 8 Gbytes
OS: Ubuntu, Linux kernel 2.6.38-rc8	
Guest machine	
CPU: uniprocessor	Memory: 1 Gbyte
OS: Debian, Linux kernel 3.19.0	Disk: 32 Gbytes

Table 7.2: Benchmarks executed.

Benchmark	Parameters
apache	-n100000 -c20
2*fileio	–file-total-size=6G –file-test-mode=rndrw –file-extra-flags=direct –max-requests=10000
make	linux-4.0 config with all-no
2*mysql	-test=oltp -oltp-test-mode=simple -max-requests=500000 -table-size=4000000
radiosity	-p1 -bf 0.005 -batch -largeroom

7.1 Handling Non-Deterministic (ND) Events

The log contains three kinds of ND events. Here we describe the types of ND events and how they are recorded and replayed.

Synchronous ND Events. Instructions such as *rdtsc* (read time stamp counter) or *rdrand* (read random number generator) return ND results. Accesses to memory regions like Memory Mapped IO (MMIO) are also ND. The VMCS controls when the processor will perform a VMExit. We configure the controls to synchronously trap these ND accesses, allowing the hypervisor to log their results. With similar configuration of the controls on the replaying system, these events are deterministically reproduced during replay.

Network inputs are a special case and are also synchronous in our system. The arrival of network packets to the physical NIC is inherently asynchronous but the data is delivered to the VM at the boundaries of synchronous VMExits. Thus, this simplifies the recording and replaying of network events.

Synchronous ND Events. Instructions such as *rdtsc* (read time stamp counter) or *rdrand* (read random number generator) return ND results. Accesses to memory regions like Memory Mapped IO (MMIO) are also ND. The VMCS controls when the processor will perform a VMExit. We leverage this fact to trap these ND accesses, allowing the hypervisor to log their ND results. These VMExits are deterministically reproduced during replay. The replay time VMExits can be associated with the corresponding VMExits from the recorded execution if a synchronous VMExit count is maintained during record. Delivering the inputs during these VMExits will faithfully replay them.

Network. ND inputs from network traffic can constitute significant portions of the input log. Network traffic addressed to the VM arrives as packets through the host machine's physical NIC and are subsequently delivered to the appropriate VM virtual NIC by the QEMU IO thread. The ND packet contents and the point where the data is injected into the virtual NIC must be logged in the input log. The injection into the virtual NIC occurs at boundaries of synchronous VMExits. Therefore, as before, the synchronous VMExit number is enough to enable faithful replay.

Asynchronous ND Events. Asynchronous events are more challenging to replay. These events are due to external (to the processor) interrupts. Examples include inter-processor interrupts and interrupts from physical devices

like disks.

The VMCS can be configured to cause these events to trigger asynchronous VMExits. Since these VMExits are asynchronous, they will not naturally occur during replay. Faithful replay requires that these events be delivered at the same point where they originally occurred. Therefore, we are forced to recreate them manually.

Replicating these VMExits is not straightforward. Insight uses performance counters to cause a VMExit close to where there needs to be one. Then, we perform one VMExit per instruction, single-stepping until execution reaches the injection point. Each VMExit costs about ≈2,000 cycles.

Asynchronous ND Events. Asynchronous events are more challenging to replay. These occur from external interrupts. These interrupts originate from other processors or from physical devices like disks. The VMCS structure can also be configured to cause a VMExit on these events. These VMExits, however, are asynchronous and will not repeat on the same instruction during replay. Therefore, for faithful replay, replay has to manually recreate them.

Trapping the VM at the same processor context is not straightforward. Insight uses performance counters to cause a VMExit as close as possible to the required point in replay. From there, the processor is single-stepped until execution reaches the desired injection point. Each step will suffer the overhead of a VMExit (≈2,000 cycles).

7.2 Evaluating Replay Overhead

To evaluate the overhead of checkpointing replay, we reuse the Linux copy-on-write implementation used during fork system calls. Virtual memory belonging to the VM is allocated within a user-space QEMU process running on the host machine. With minor modifications, a checkpoint can be created by forking the QEMU process.

The alarm replayer models the RAS at every call and return instruction. Unfortunately, current Intel VTx extensions do not support trapping call and return instructions. Hence, to measure the performance impact of alarm replay, we modified GCC to instrument binaries by inserting a debug exception before kernel context switches, and before call and return instructions. The debug exception is a single byte opcode (0xCC) used to trap instructions by

raising debug exceptions. The VMCS is configured to cause VMExits on debug exceptions. This allows us to mimic the behavior of the alarm replayer, modulo a minor performance impact due to a 0.11% increase in the size of the Linux binary.

7.3 Evaluating the Proposed Hardware

In binary translation mode, QEMU virtualizes the processor using software only. This mode is significantly slower, but it allows for simulation of hardware. We use this mode to evaluate our proposed hardware modifications in RnR-ROPSafe. We simulate a 48-entry RAS by default.

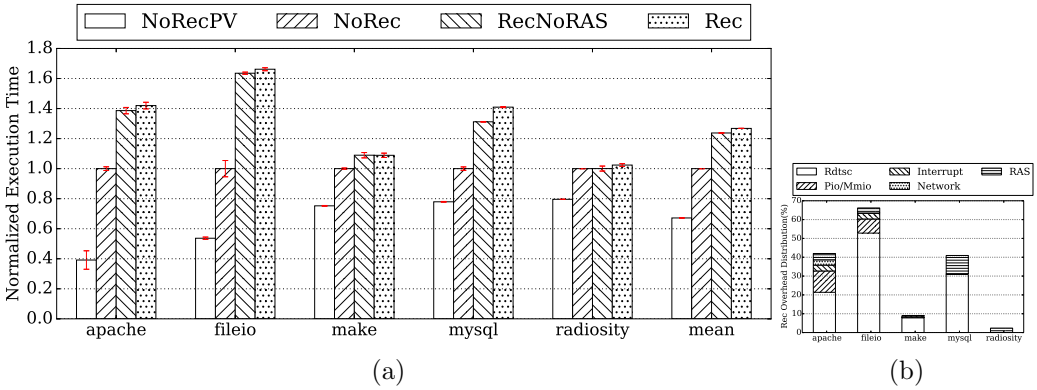
CHAPTER 8

EVALUATION

8.1 Recording

Our recording scheme generates the log and also saves/restores the RAS at context switches. Recall we require hypervisor-mediated I/O, which prevents the use of para-virtualized network drivers (PV). We call the scheme *Rec*. Figure 8.1(a) compares *Rec*'s execution time to three other setups: no recording with PV drivers (*NoRecPV*), no recording and no PV drivers (*NoRec*), and recording without dumping the RAS (*RecNoRAS*). Each benchmark is normalized to *NoRec*.

We see that disabling PV increases the execution time of these benchmarks by 25-150%. As previously mentioned, RnR has been successfully applied to PV drivers [55]; applying those techniques in our solution would eliminate this overhead from our system. Apache and fileio are affected the most, while mysql is not impacted much as it avoids disk accesses by caching recently accessed tables in memory.



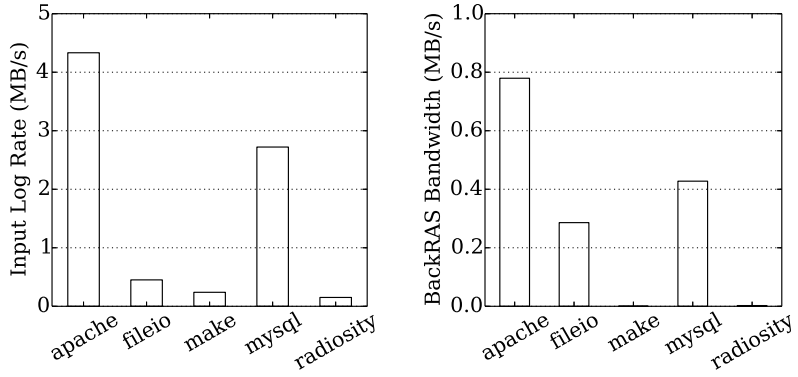
(a) (b)

Figure 8.1: Execution time of recording setups (a) and breakdown of the *Rec* overhead over *NoRecNoPV* (b).

Recording (*Rec*) takes, on average, 32% longer than *NoRec*. Recording without saving/restoring the RAS (*RecNoRAS*) takes 28% longer than *NoRec*. These overheads are modest. To understand their source, Figure 8.1(b) shows again the slowdown of *Rec* over *NoRec* and breaks it down into their sources, namely recording timer reads (*rdtsc*), port and memory-mapped I/O accesses (*pio/mmio*), interrupts, network packet contents, and saving/restoring the RAS.

We see that the dominant overhead across all benchmarks is due to recording *rdtsc*. This event occurs very frequently, especially in fileio and mysql, where the application itself issues many timer reads to measure transaction speed. In addition, fileio issues disk command and control signals using pio. It also has DMA activity, which causes interrupt events to signal file access completion. Apache receives network packets and uses mmio accesses to the NIC to retrieve the packets. The more computation-intensive benchmarks (make and radiosity) have little overhead. Finally, saving/restoring the RAS induces only 4% overhead on average.

Figures 8.2(a) and (b) show the input log generation rate, and the bandwidth of RAS saving and restoring, respectively, for all our benchmarks. We do not compress the data. We see that the rates are low. Apache has the highest input log rate (4 MB/s) because it records network packet contents. RAS save/restore bandwidth is very small.



(a) (b)

Figure 8.2: Input log generation rate (a) and bandwidth to save/restore the RAS (b).

8.2 Minimizing False Alarms

The RnR-ROPSafe hardware eliminates most of the false alarms in the kernel, allowing only a few false alarms (in our case due to RAS underflow) to be reported to the replayers. Figure 8.3 shows the number of kernel false alarms reported to the replayers (*FalseAlarm*) and those suppressed with the whitelist and with the BackRAS. The figure shows the number per million instructions. Since the number of remaining false alarms is so small, the *FalseAlarm* category cannot be seen, and we put the number on top of the bars. All the benchmarks except Apache have practically no kernel false alarm. Apache has a few false alarms because it has some deep procedure nesting under network stress conditions. Both the whitelist and the BackRAS are very effective at removing false alarms.

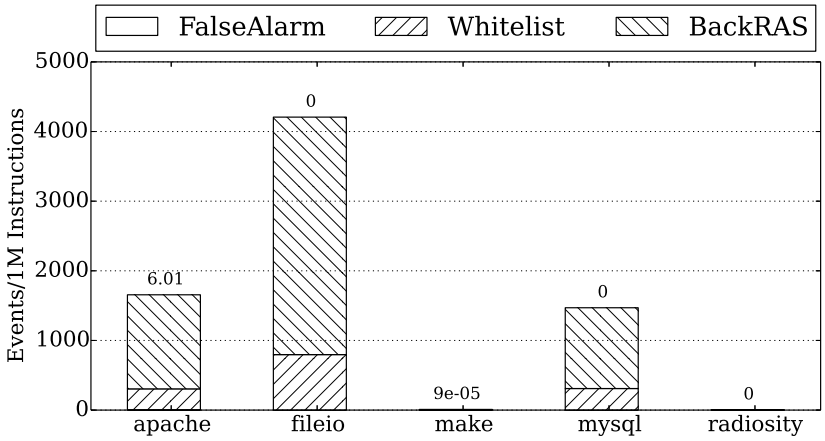
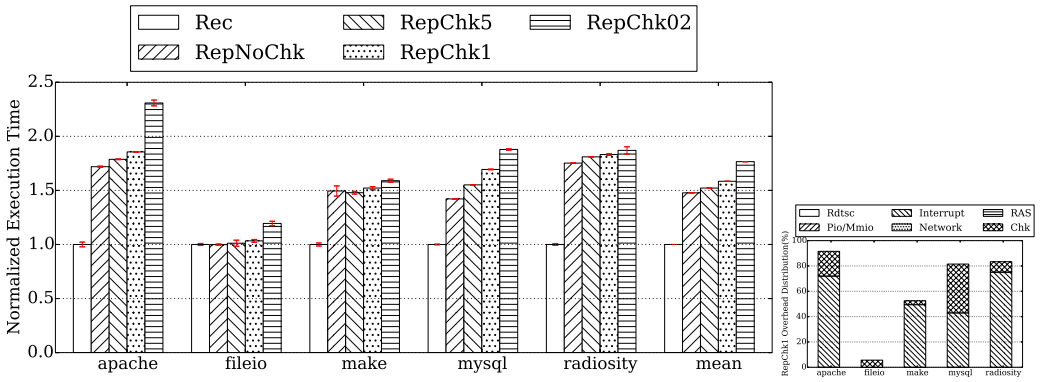


Figure 8.3: Kernel alarms and alarms suppressed.



(a) (b)

Figure 8.4: Execution time of checkpointing replay setups (a) and breakdown of the *RepChk1* overhead over *Rec* (b).

8.3 Replaying

8.3.1 Checkpointing Replay

Figure 8.4(a) compares the execution time of various checkpointing replay setups to the recording setup (*Rec*). The replay setups use no checkpointing (*RepNoChk*) or checkpoint every 5, 1, or 0.2 seconds (*RepChk5*, *RepChk1*, and *RepChk02*, respectively). The bars are normalized to *Rec*. From the data, we see that checkpointing every 1 second (*RepChk1*) increases the execution time over *Rec* by 59% on average.

These results show that checkpointing replay runs at a speed that is roughly comparable to that of recording. As a result, checkpointing replay can be *on* all the time. While checkpointing replay is a bit slower, it can catch up with recording because busy machines are rarely 100% utilized — they are often waiting for multiple reasons. During that time, recording slows down but replay can continue. If the replay gets significantly behind, we can use backpressure to temporarily slow down recorded execution.

The figure also shows that increasing or decreasing the checkpoint period changes the speed. Interestingly, even without checkpointing, replay already takes on average 48% longer that *Rec*.

To understand these effects, Figure 8.4(b) shows again the slowdown of *RepChk1* over *Rec* and breaks it down into its sources. The sources are those during recording plus creating checkpoints (*Chk*). During recording, *RAS* involved saving/restoring the RAS at context switches; now it additionally includes saving (but not restoring) the RAS at VMExits.

The breakdown in the figure shows that creating checkpoints contributes noticeably to the total overhead. This is why the frequency of checkpoints matters. The actual overhead depends on the memory write characteristics of the workload; poor memory locality causes more page copies, increasing checkpointing overhead.

Interestingly, we see that interrupt overhead dominates. The reason is that interrupts are asynchronous events, while *rdtsc*, *pio/mmio*, and *network* are synchronous. Identifying the instruction that should get the asynchronous interrupt injected during replay is time-consuming. As indicated in Section 7.1, it requires single-stepping VMExits over several instructions. This is the reason for the overhead of Figure 8.4(b). It also explains why replay-

ing without checkpointing (*RepNoChk*) already has significant overhead over *Rec*.

8.3.2 Alarm Replay

Finally, Figure 8.5 compares the execution time of alarm replay (*RepAlarm*) to previously shown environments: checkpointing replay (*RepChk1*) and recording (*Rec*). The bars are normalized to *Rec*. Alarm replay needs to trap on every call and return instruction. Hence, the slowdown of this mode directly relates to how many kernel call and return instructions were executed. We see that replaying make and mysql takes 30-40x longer than recording them. For apache, it takes 50x. On the other hand, for radiosity, with its modest kernel activity, it takes 2.8x.

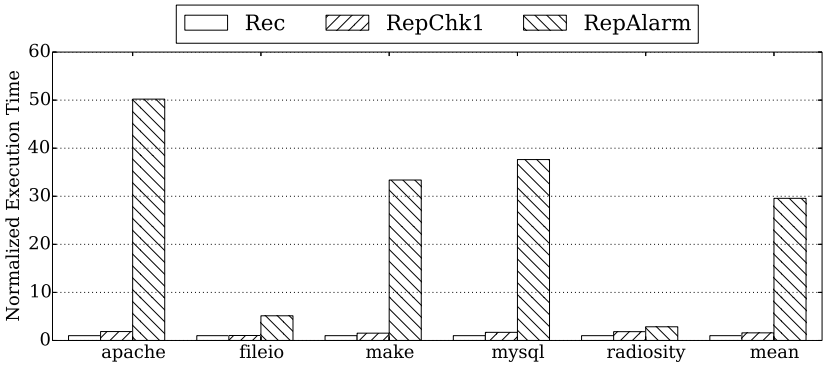


Figure 8.5: Execution time of alarm replay.

CHAPTER 9

RELATED WORK

Hardware-Software Approach. Our approach shares similarities with Raksha [63]. Raksha raises security exceptions on potential security violations. A software exception handler investigates the threat. Our approach of replaying the execution upon threat identification is more powerful. This is because an exception handler is limited to inspecting a single system state—whereas replay can inspect the steps leading to the exception. If the exception occurs after the attack, our replay approach is better suited to assess the damage of the attack.

Targeting the RAS for Security. The behavior of call and return instructions has been targeted in the past to secure against buffer overflow attacks [25, 26, 37, 64]. Tuck et al. [36] explored using hardware to encrypt variables used to modify control flow, a technique proposed by PointGuard [37]. The proposal is for programs to ensure that control flow modifying variables are stored encrypted and decrypted prior to being used. Assuming the attacker cannot discover the secret key, the attacker will not be able to redirect control flow even if the attacker can rewrite a control-flow variable. These techniques are starting to be used today [35] in protecting against code reuse attacks. SmashGuard [25, 26] proposed using secured RASes that are backed by memory which require non-trivial hardware changes.

Record and Deterministic Replay (RnR) for Security. The closest previous work to ours is Aftersight [51]. It suggests using VM-level RnR to perform online dynamic analysis of a system's execution. Although it lays out the general direction for VM-level RnR for online analysis, Aftersight does not address some important aspects of such a model. For example, unlike our proposal, Aftersight assumes that the replay analysis is constantly running and is able to catch up with (or only modestly slow down) the recording; otherwise, it loses precision and might introduce false positives. This is not a reasonable assumption in case of heavy-weight analysis such as our ROP

detection algorithm. In contrast, RnR-ROPSafe advances the state of the art by proposing a holistic architecture that captures many practical aspects of RnR-based online security analysis. These key practical aspects are: (1) Co-designed hardware-software mechanisms (e.g., the RAS extensions) to achieve reasonable overhead while keeping hardware changes simple; (2) separate checkpointing and alarm replayers; and (3) need-based triggering of analysis replays (as opposed to constantly-running analysis).

CHAPTER 10

CONCLUSIONS

In this thesis we propose a departure from the traditional approach to building defenses against malware. The traditional approach is to monitor and/or enforce the security properties inline with the execution. Complex security properties such as control-flow or data-flow integrity require monitoring and tracking common events—like memory access or branches. The price of instrumenting programs with the code to perform these checks and make these measurements is too high. As such, modern systems do not preserve these properties, despite their need to prevent modern malware. Our approach decouples the security checks from program and system execution via Record and Replay. There are additional benefits, which we have only alluded to in this thesis.

First, since replays can be instrumented, the system gains the ability to introspect prior executions. This can be used to assess damage from prior intrusions or to discover new intrusions in cases where new malware is discovered. One way this can be used is to verify the veracity of potentially misplaced security alarms. Previously, it was unacceptable for a security alarm to be incorrect, as it implied an innocuous program was halted. Being perfect for all executions requires a detector just as accurate in its assessment of corner case executions as it is for common case ones; otherwise, the attacker will either escape detection by exploiting the detector's gap in coverage or turn the detector against the very system it protects by exploiting its inaccuracy. Unfortunately, being correct for all executions is expensive as corner cases are often pathological in nature. Thus, verifying security alarms after the fact with RnR is a welcome flexibility that allows the detector to narrow its focus on the common case executions.

Our approach allowed simple, non-intrusive, and inexpensive techniques to be used to protect systems. RnR was utilized to filter false-positives and compensate for detector shortcomings. With the RAS-based ROP detector,

the rare cases were highly nested codes which caused RAS underflow events. Replay was used to reconstruct these relevant events and analyze them—in a way which the detector could not—to corroborate or refute the alarms. Thus, we were able to use the RAS—despite its imprecision—and RnR to detect ROP attacks without significant overheads. Future work can explore the additional detectors and additional attack surfaces.

REFERENCES

[1] C. Otterstad, "A brief evaluation of Intel MPX," in *Systems Conference (SysCon), 2015 9th Annual IEEE International*, April 2015, pp. 1–7.

[2] American Micro Devices, "Amd64 architecture programmer's manual volume 2: System programming," 2006.

[3] J. Winter, "Trusted computing building blocks for embedded linux-based ARM Trustzone platforms," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, ser. STC '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1456455.1456460 pp. 21–30.

[4] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, December 2015, no. 253669-033US.

[5] H. Shacham, "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*, ser. CCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313 pp. 552–561.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165 pp. 340–353.

[7] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966920 pp. 40–51.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013, pp. 48–62.

[9] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046713 pp. 29–40.

[10] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies," 2004.

[11] PaX Team, "PaX address space layout randomization (ASLR)," 2003.

[12] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG et al., "Ropecker: A generic and practical approach for defending against ROP attack," 2014.

[13] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas pp. 447–462.

[14] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang pp. 337–352.

[15] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 575–589.

[16] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas pp. 417–432.

[17] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium*, 2014.

[18] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, ser. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372399.2372409 pp. 7–7.

[19] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124 pp. 298–307.

[20] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 211–224, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844148

[21] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095820 pp. 91–104.

[22] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346321 pp. 308–318.

[23] ARM, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition Issue C*, July 2012.

[24] PaX Team, "Non executable data pages," 2004.

[25] H. Ozdoganoglu, T. Vijaykumar, C. Brodley, B. Kuperman, and A. Jalote, "Smashguard: A hardware solution to prevent security attacks on the function return address," *Computers, IEEE Transactions on*, vol. 55, no. 10, pp. 1271–1285, Oct 2006.

[26] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, *Security in Pervasive Computing: First International Conference, Boppard, Germany, March 12-14, 2003. Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. Enlisting Hardware Architecture to Thwart Malicious Code Injection, pp. 237–252. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39881-3_21

[27] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 94–105.

[28] E. Aktas, F. Afram, and K. Ghose, "Continuous, low overhead, run-time validation of program executions," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.18 pp. 229–241.

[29] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, pp. 292–307.

[30] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694383 pp. 487–502.

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

[32] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755934 pp. 195–208.

[33] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294295 pp. 351–366.

[34] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-ROP defenses," in *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 88–108.

[35] B. Spengler, "Grsecurity," 2006. [Online]. Available: http://grsecurity.net/rap_announce.php

[36] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, Dec 2004, pp. 209–220.

[37] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard TM: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, 2003, pp. 91–104.

[38] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "IBM POWER7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1:1–1:29, May 2011.

[39] B. Sinharoy, J. Van Norstrand, R. Eickemeyer, H. Le, J. Leenstra, D. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. Fernsler, "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2:1–2:21, Jan 2015.

[40] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629594 pp. 193–206.

[41] T. Bressoud and F. Schneider, "Hypervisor-based fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, February 1996.

[42] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346273 pp. 121–130.

[43] T. J. Leblanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 471–482, April 1987.

[44] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, June 2008, pp. 289–300.

[45] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629593 pp. 177–192.

[46] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772958 pp. 2–11.

[47] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "Quickrec: Prototyping an Intel architecture extension for record and replay of multithreaded programs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485977 pp. 643–654.

[48] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 3:1–3:24, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110359

[49] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, June 2003, pp. 122–133.

[50] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313 pp. 347–356.

[51] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," ser. USENIX ATC, June 2008.

[52] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," ser. USENIX Ann. Tech. Conf., April 2005.

[53] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[54] "Qemu open source process emulator," http://qemu.org.

[55] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr, "Abstractions for practical virtual machine replay," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2892242.2892257 pp. 93–106.

[56] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Comput. Secur.*, vol. 37, pp. 215–227, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2013.02.002

[57] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[58] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: A software-hardware interface for practical deterministic multiprocessor replay," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508254 pp. 73–84.

[59] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945467 pp. 223–236.

[60] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866370 pp. 559–572.

[61] "CVE-2015-5364." Available from MITRE, CVE-ID CVE-2015-5364., Dec. 3 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5364

[62] R. Senthilkumaran and P. Kulkarni, "Insight: A framework for application diagnosis using virtual machine record and replay," Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, Tech. Rep. TR-CSE-2014-57, January 2014.

[63] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250722 pp. 482–493.

[64] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix Security*, vol. 98, 1998, pp. 63–78.