Record-Replay Architecture as a General Security Framework

Yasser Shalabi, Mengjia Yan, Nima Honarmand,† Ruby B. Lee,‡ and Josep Torrellas
University of Illinois at Urbana-Champaign †*Stony Brook University* ‡*Princeton University*
http://iacoma@cs.uiuc.edu *nhonarmand@cs.stonybrook.edu* *rblee@princeton.edu*

***Abstract*—**
Hardware security features need to strike a careful balance between design intrusiveness and completeness of methods. In addition, they need to be flexible, as security threats continuously evolve. To help address these requirements, this paper proposes a novel framework where Record and Deterministic Replay (RnR) is used to *complement* hardware security features. We call the framework *RnR-Safe*. RnR-Safe reduces the cost of security hardware by allowing it to be less precise at detecting attacks, potentially reporting false positives. This is because it relies on on-the-fly replay that transparently verifies whether the alarm is a real attack or a false positive. RnR-Safe uses two replayers: an always-on, fast *Checkpoint* replayer that periodically creates checkpoints, and a detailed-analysis *Alarm* replayer that is triggered when there is a threat alarm.

As an example application, we use RnR-Safe to thwart Return Oriented Programming (ROP) attacks, including on the Linux kernel. Our design augments the Return Address Stack (RAS) with relatively cheaper hardware. We evaluate RnR-Safe using a variety of workloads on a VM running Linux. We find that RnR-Safe is very effective. Thanks to the judicious RAS hardware extensions and hypervisor changes, the checkpointing replayer has an execution speed comparable to the recorder. In addition, the alarm replayer needs to handle very few false positives.

***Keywords*-component; formatting; style; styling;**

I. Introduction

As security attacks become 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. Hence, defense systems have to be flexible. 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. Unfortunately, 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, hence bypassing W⊕X defenses.

An intriguing primitive that can help defend against security threats is Record and Deterministic Replay (RnR) (e.g., [6], [7], [8], [9], [10]). 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 [7], [9]. It has also been used to support speculating past security checks [11].

In this paper, we explore a novel approach to hardware security design, where RnR is used to *complement* hardware security features—to offload intrusion checks and/or to eliminate check imprecision. We call the framework *RnR-Safe*. In RnR-Safe, we *reduce* the cost of security hardware, by allowing the hardware to be less precise at detecting attacks, potentially reporting false positives. This is because we rely on an *on-the-fly* replayer that transparently verifies whether the alarm is a real attack or a false positive. The result is a very general security framework that can be combined with a variety of relatively cheaper security hardware.

RnR-Safe relies on two types of on-the-fly replayers running on a different machine: an always-on fast replayer that periodically creates checkpoints of the monitored execution (*Checkpointing* replayer), and an analyzing replayer—triggered by an alarm—that starts from a checkpoint and analyzes the execution to determine whether the alarm was due to a real attack or a false positive (*Alarm* replayer). The alarm replayer can execute multiple times, with different levels of analysis, until the attack is fully understood.

As an example application, this paper then applies this approach to thwart ROP attacks—including *on the kernel*, a challenging target to defend. A ROP attack causes a Return Address Stack (RAS) misprediction. However, a RAS misprediction is an imprecise ROP detector, as it may also occur for benign software. Hence, rather than augmenting the RAS hardware to guarantee perfect detection, RnR-Safe 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. This follows the RnR philosophy.

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

The contribution of this paper is threefold. First, we propose a novel RnR-based hardware-software approach to enhance security called RnR-Safe. Second, we tailor RnR-Safe to detect ROPs in the kernel space, and discuss the

challenges encountered during its implementation. Finally, we evaluate the cost of RnR-Safe in this use.

Assumed System and Threat Models. ROP attacks can occur within the kernel or user contexts, and RnR-Safe can secure both. The target most difficult to secure is the kernel. We focus on evaluating RnR-Safe'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 concurrently being replayed on a different machine, where ROP attacks may be found.

We assume the attacker can launch a ROP attack against the kernel via any combination of address disclosure and memory corruption vulnerabilities to hijack and corrupt the kernel stack. We assume that the host machine OS and hypervisor (in the recording and replaying machines) are benign, and that they can safeguard against compromised guest VMs through traditional memory page permissions.

II. Background

A. Record and Replay

Record and deterministic Replay (RnR) of workloads is a popular architectural technique (e.g., [12], [13], [14], [15], [10], [16], [17], [18], [19], [6], [20], [8]). As a workload runs, RnR records all the non-deterministic events that can affect the execution and stores them in a log. Then, 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 interleaving of memory accesses.

RnR can be done at different abstraction layers. In this work, we use VM-level RnR [7], [13], [14], [21]. 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 [22] or Qemu [23]. These assumptions are not necessarily limitations, as RnR approaches compatible with multiprocessor [6], [18] and virtualized I/O [24] exist.

There are several papers that investigate the use of RnR in a security-related scenario [7], [9], [11], [21], [20], [25]. ReVirt [7] 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[9] explores using VM-level RnR to determine if systems were previously exploited once zero-day attacks are discovered. Speck[11] explores using a combination of OS-level speculation and program-level RnR to remove security checks from the critical path of a program. ParanoidAndroid [20] and Secloud [25] explore the possibility of maintaining replicas of mobile devices in the cloud, and perform program-level RnR in the cloud. Finally, Aftersight [21] 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 one key contribution of our work: separation between the fast checkpointing replayer and the exhaustive alarm replayer. We discuss the details in Section IX.

B. Example Application: Return Oriented Programming (ROP)

RnR-Safe is a general framework that can be used to thwart a variety of attacks. As an illustration, in this paper, we consider ROP [5] attacks, including those on the kernel. Appendix A describes ROP attacks.

At a high level, ROP attacks can be detected with what is called a *Shadow Stack* [26]. The shadow stack operates with "Last In First Out" semantics. When a call instruction is encountered, the address of the instruction following the call is pushed to the top of the shadow stack. Return instructions trigger a pop from the shadow stack. ROP attacks can be detected when the return address used by the processor mismatches the one popped from the shadow stack.

However, shadow stacks present several implementation challenges. First, the validity of the shadow stack hinges on its integrity. Hence, the shadow stack must be secured against the very software it protects. This includes protecting it against the kernel itself, which might be compromised and malicious. Also, codes can be highly nested (e.g., recursive), multi-context (e.g., the kernel), or imperfectly nested (e.g., error and exception handling). Since false positives are unacceptable, proper handling of these corner cases is necessary.

As a result, there have been a variety of approaches that focus on protection against ROP attacks (e.g., [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]). Unfortunately, while some of them are provable prevention techniques, they appear insufficient from a practical point of view. Indeed, it is a matter of fact that systems today still remain vulnerable to such attacks. We discuss the reasons in the next section.

C. ROP is Still Unsolved Today

There are three reasons why existing ROP protection techniques have limitations from a practical point of view:
Hardware Intrusiveness. Some approaches [30], [32], [35], [29] require intrusive hardware changes. For example, both SmashGuard [29] and SRAS [30] add a hardware stack used to verify the return targets. If the hardware stack overflows, it spills into system memory. Also, it needs additional read-write ports. The PUMP [35] processor supports general metadata propagation. This can be used to implement various safety checks, including Control Flow Integrity (CFI). However, each stage of the pipeline is modified, to support tag storage and/or rule execution. REV [32] hashes the instruction sequences within a basic block to verify a program's control flow. It requires an additional 32KB first level cache dedicated to cache signatures, to avoid prohibitive slowdowns.
Software Impact. The completeness of instrumentation-based CFI-enforcing solutions such as [27], [28] is attractive.

However, *securely* maintaining a shadow RAS at call/ret boundaries via binary instrumentation adds overheads of over 100% [28]. Compiler based solutions can add 6–8% overhead [39]. Other approaches [34], [40] propose recompiling the kernel and application 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 performance costs, source code is not always available, which limits the applicability of this technique.

Completeness. Other proposals explore alternative approaches by either detecting ROP attack characteristics [33], [36], [37] or by making the ROP attacks difficult to mount [41], [42], [43], [44]. By monitoring control flow characteristics that are indicators of ROP execution [33], [36], [45], some ROP payloads can be detected. Unfortunately, ROP payloads may be able to blend their signature to match that of benign code to evade detection [46], [47]. Similar criticisms can be levied against other proposals which provide probabilistic defences through randomization [44] and encryption [48], [49], [50]. These defences will randomize code locations or encrypt on-stack return addresses, thus complicating critical steps in attack code. In particular, Address Space Layout Randomization (ASLR) [41], [42], [43], [44] is a widely deployed defence adopting this approach. While useful in the short-term, these approaches remain insufficient in the long-term as attackers have learned to circumvent them [51], [52], [53]. We discuss ASLR in more detail in Section IX.

Recently, Intel has introduced Control-Flow Enforcement Technology (CET) [54], which has shadow stacks that verify return targets. CET prevents traditional memory-modifying instructions from modifying shadow stack pages. However, to our understanding, since these shadow stack pages are in memory, it is conceivable that an attacker can subvert the system and update the pages. The same concern arises for Griffin [55]—a CFI verification technique based on analysis of branch traces. The traces are provided by Intel's Processor Trace, and are stored in the system memory directly by the hardware.

In this paper, we use RnR-Safe to support ROP protection in a different manner, without a hardware shadow stack. Effectively, the *replayer* in a *secure* machine implements the shadow stack *in software*.

D. Return Address Stack

Modern processors use a hardware Return Address Stack (RAS) to predict the target of return instructions. When a procedure call 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 [56] and POWER8 [57] processors have a RAS with 32 and 64 entries, respectively.

ROP attacks cause RAS mispredictions because the attacker modifies the stack to return to an unexpected instruction. However, 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.

III. An RnR Security Framework

Figure 1 shows the organization of RnR-Safe, our envisioned security framework. 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—less than 15% on average, according to Pokam et al. [18]. Note that we record at the VM level to also protect the operating system.

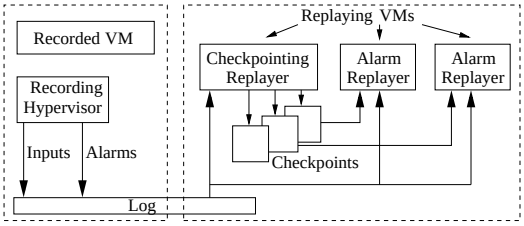


Figure 1: RnR-Safe organization.

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, with potentially some false positives. When this hardware or the recording hypervisor suspect an 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 determistically follows the original one. If an alarm is found in the log, the replayer characterizes the alarm, detecting either an attack or a false positive.

A. RnR-Safe Modes of Execution

In RnR-Safe, monitored recording consists of normal execution, while transparently recording all the non deterministic inputs in a log, and transparently monitoring for 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 not acceptable.

In RnR-Safe, the replay execution is performed with two types of replayers. One is the *Checkpointing Replayer*. Such replayer runs all the time, at roughly recording speeds. It uses the log to determistically 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* out of a recent (typically the latest) checkpoint. Once old checkpoints and log entries are verified, they can be discarded to save storage.

The second type of replayer is the *Alarm Replayer*. An alarm replayer replays log entries from a given checkpoint until an alarm marker, 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 characterize the attack. It can be much slower than the checkpointing replayer. Typically, alarms are rare events.

B. What RnR-Safe Offers

RnR-Safe provides three security benefits.

Robustness at Relatively Modest Hardware. Perfect detection accuracy often necessitates very intrusive hardware. RnR-Safe minimizes intrusiveness by separating alarm detection from attack verification using RnR. False positive alarms and rare corner cases are handled by software-based replay. Thus, RnR restores robustness to a system built out of imprecise security hardware. The one requirement of the alarm hardware is to avoid any false negatives.

Flexibility. RnR-Safe is flexible. As attackers devise new attacks, defenders can augment the recorded VM with hardware and software for new alarm generation, and the replaying VMs with software for new analysis techniques. The addition of new replayers is particularly compelling, as the analysis is performed in software. Multiple types of attacks can be tracked at the same time.

Execution Auditing. RnR-Safe allows detailed analysis of executions offline. An execution context 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.

C. How to Use RnR-Safe

The RnR-Safe framework can be tailored to protect against different attacks. For each attack, we first need a first-line of defense that targets that threat in the machine. The key advantage of RnR-Safe is that such a defense, implemented in hardware or software, can be imprecise—i.e., it can suffer false positives. This property often makes the design less intrusive or complicated.

While it is currently unclear to us which types of attacks would be best suited for RnR-Safe to defend, Table I outlines three example attacks: ROP, jump-oriented programming (JOP) [58] and denial of service (DOS) [59]. For each, the table outlines the alarm trigger, possible first defense, and the role of relay. The first entry is ROP, which will be discussed in detail in the rest of this paper. The alarm trigger is a RAS misprediction, and the first defense is based on dumping a modified form of the RAS, and using a whitelist of acceptable RAS mispredictions (as we will see). The role of replay is to model a kernel-compatible shadow stack.

JOP is a related class of attacks, mounted by redirecting branches and call instructions to execute the victim's code. Preventing such attack requires a different solution, based on preserving the CFI of call and branch instructions. A first defense against JOP can be 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 replay verifies that the target is one of the less common 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 raised, and RnR analyzes and identifies the code that has dominated the system's execution time. Due to space limitations, this paper focuses only on the ROP attack next.

Attack	Alarm Trigger	Possible First Defense	Role of Replay
ROP (this paper)	RAS misprediction	Dump the RAS, Whitelist	Execute a *kernel-compatible* shadow stack algorithm
Jump Oriented Programming (JOP)	Stray indirect branch	Table of entry and exit addresses of the most common functions	Verify if the target is one of the less common functions
Denial of Service (DOS)	Kernel scheduler inactivity	Counter of number of context switches	Identify reason for low switching frequency

Table I: Examples of potential RnR-Safe uses.

IV. Example: RnR-Safe to Thwart Kernel ROPs

A. Main Idea

The architecture primitive that we use to help detect ROPs is the RAS. The RAS stores the addresses of the predicted targets of return instructions. A ROP attack, by causing returns to unexpected addresses, induces RAS mispredictions.

To use RAS mispredictions to thwart ROP attacks in RnR-Safe requires that there be no false negatives. Fortunately, execution of ROP payloads is guaranteed to cause RAS mipredictions, making false negatives impossible. Furthermore, for this detector to be useful in RnR-Safe, false positive alarms should be infrequent. There are a few major sources of false positives in the basic RAS operation. We will explain these sources with Linux kernel examples.

First, there is the effect of multithreading. In a multithreaded environment, 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*, 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.

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 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 of procedure calls is another reason for RAS mispredictions—a situation where a procedure is called but never returned from. Within the kernel, these are 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—e.g., in exception handling implemented using setjmp/longjmp.

These effects show that the RAS is a detector of ROPs with many false positives. For RnR-Safe to use it as the initial defense, we need to robustify the RAS detection capability with simple support to minimize the false positive rate. To completely eliminate false positives requires disruptive software changes and intrusive hardware changes.

An alarm replayer is invoked when there is an alarm. It starts from a nearby checkpoint created by the checkpointing replayer. It distinguishes false alarms from real attacks, and characterizes any detected ROPs.

In the following, we describe the components of RnR-Safe to protect the kernel despite its unique RAS challenges.

B. Basic Design

As shown in Figure 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.

A conventional RAS is slightly augmented in this *basic* design of RnR-Safe. 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, 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.

As the checkpointing replayer consumes the log, if it finds an 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 not.

This basic RnR-Safe design will not miss an attack, but it suffers from false alarms. Next, we extend this basic design to reduce its false positives.

C. Supporting a Multithreaded Environment

In a multithreaded environment, a thread might be descheduled while executing in kernel mode. The return addresses left by this thread in 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 false ROP alarms.

To address this problem, RnR-Safe extends the processor hardware as follows. On a context switch, the hardware 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 context switch to also include the RAS.

The structures are shown in Figure 2. The software structure in memory is an array of backed-up RASes (*BackRAS array*). 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-in 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.

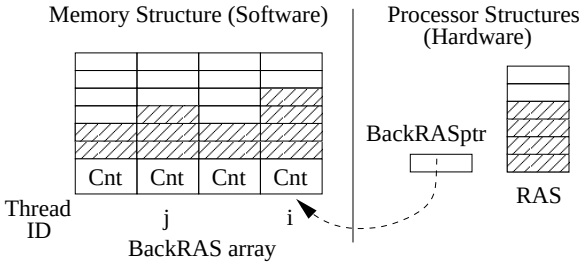


Figure 2: Structures used to support multiple threads.

Figure 3 shows the logic used. On a context switch, as part of the transition to the hypervisor, the hardware saves the RAS to the entry pointed to by BackRASptr. In addition, it stores the count of saved entries. Our measurements show that a transition to the hypervisor takes about 1,000 cycles. We estimate that backing-up the RAS will add about 200 cycles. 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, taking another 200 cycles.

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 V-B1 explains how this is done without modifying the guest kernel.

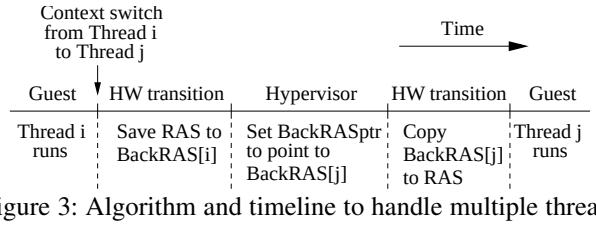


Figure 3: Algorithm and timeline to handle multiple threads.

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

D. 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 will corrupt the RAS state.

In the Linux version we use, this use of returns occurs in one place, namely 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 three 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-Safe 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. These lists are only writable by the hypervisor.

The logic used and its timeline are as follows. 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 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, a VM exit is triggered.

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

E. 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.

RnR-Safe 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-Safe lets these events raise ROP alarms, and relies on the replayer to identify them as false positives. Since the replayer models an unbounded RAS in software, it can filter out such false positives.

Similary, we let the processor to raise ROP alarms for mispredictions due to imperfect procedure nesting in the kernel. Such events are hard to handle transparently in hardware. However, they are easily filtered out by our alarm replayer. It should be noted that both of these events are very rare.

F. Replaying Platform

The input log is passed on-the-fly to another platform, where a VM running the checkpointing replayer deterministically replays the execution.

1) Checkpointing Replayer: To understand the operation of the checkpointing replayer (CR), we first describe the contents of a checkpoint. Figure 4 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.

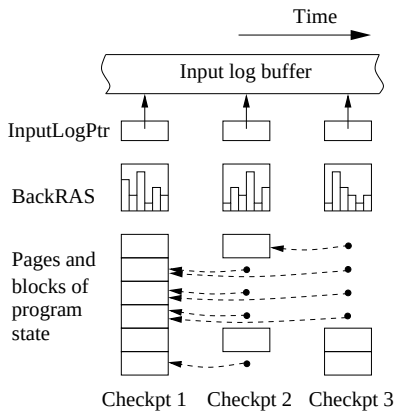


Figure 4: Checkpoints created by the checkpointing replayer.

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 third component is the BackRAS at the time of the checkpoint. We will see in Section IV-F2 that it is needed.

The hardware on which the CR runs works slightly differently than in the recorded VM. Specifically, the RAS is dumped into the BackRAS not just at context switching points, but also at VM exits 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 IV-C, we estimate a VM exit and subsequent entry to take $\approx$ 2,000 cycles, and saving the RAS to take $\approx$ 200 cycles.

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

With this background, we can describe the CR operation. The CR executes the recorded VM in a deterministic manner,

periodically creating checkpoints. 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 memory pages and disk blocks modified since the previous checkpoint, 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. When a page is modified for the first time since the last checkpoint, a copy is made and used from now on.

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.

2) Alarm Replayer: When the CR encounters an alarm, it initiates an alarm replayer (AR). The AR determines whether the alarm is caused by a ROP or is a false alarm. If the former, the AR provides the state of the system at the point of the ROP attack. Moreover, the AR can be re-run multiple times, with increasing levels of instrumentation, to fully characterize the attack.

The 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 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. There, it models in software an unbounded RAS with our extensions for multithreading and non-procedural returns.

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 study the execution state to glean information about the attack. Section VI shows an attack. Note that this design allows running multiple ARs concurrently to analyze multiple ROP alarms in parallel.

V. Implementation Issues

This section summarizes the hardware and hypervisor support required for the architecture described. 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.

A. Hardware Support

The hardware support required is modest. It largely reuses the existing RAS hardware and, of course, RnR hardware. On top of that, it adds the BackRASptr register and the two whitelist tables. 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. The RnR infrastructure can be reused for a large variety of debugging and security analyses.

The maintenance of the BackRAS array in memory, and of the BackRASptr and whitelist tables is performed in microcode. Specifically, we extend the VMCS with three new fields for the BackRASptr and the two whitelist tables. Microcode reads these fields to program these three processor hardware structures. Then, microcode uses the value of BackRASptr to dump the RAS contents into the active BackRAS entry in certain VMExits, and to read the active BackRAS entry into the RAS in certain VMEnters.

B. Hypervisor Support

1) Programming BackRASPtr on a Context Switch: 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 microcode, the hardware dumps the RAS into the memory location pointed to by BackRASPtr.

Once the VMExit is complete and the control is transfered to the hypervisor, it can introspect the state of the guest OS 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 next thread's ID is found, the hypervisor checks the map to determine its BackRAS entry. Then, the hypervisor sets the BackRASPtr field of the VMCS to point to the BackRAS entry.

2) 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.

VI. Mounting A Kernel ROP Attack

We built and mounted the ROP attack of Figure 10. In the recording VM, as the workload calls the *vulnerable* procedure of Figure 10(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 above the buffer in the software stack of Figure 10(e). After the malicious string copy, the software stack becomes Figure 10(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 10(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 an 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 further back in time to perform a deeper analysis of the system.

One question that 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 analyis 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 reveals 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 [60]. This information is easy to get now because the workload is not running.

VII. Experimental Setup

A. Goal of the Evaluation

In this paper, the goal of our evaluation is to assess the overhead of recording, and of replay using the checkpointing and alarm replayers. We also want to know the rate of log generation, the bandwidth consumed to save/restore the RAS, and the frequency of alarms. Additional information includes the time window between attack and detection, the log generated during this window, and the number of checkpoints that the system needs to retain. In our work, we only mount the kernel ROP attack that we describe in Section VI. Such attack is representative of ROP attacks, as they all use the same gadget-based pattern. Collecting and analyzing multiple real-world kernel ROP attacks is left as future work.

Note that our design does not support recovery. When an alarm is raised, the machine being recorded can either continue while a replay is underway, or it can pause and wait for alarm verification. We do not envision roll back and recovery. A future design may consider it.

B. Evaluation Environments

To evaluate RnR-Safe, we use two evaluation environments. The first one evaluates the performance of our recording and replaying modes. For this, we use *Insight* [61], 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 II shows the system configuration we use for our performance evaluation, and Table III shows our benchmarks.

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

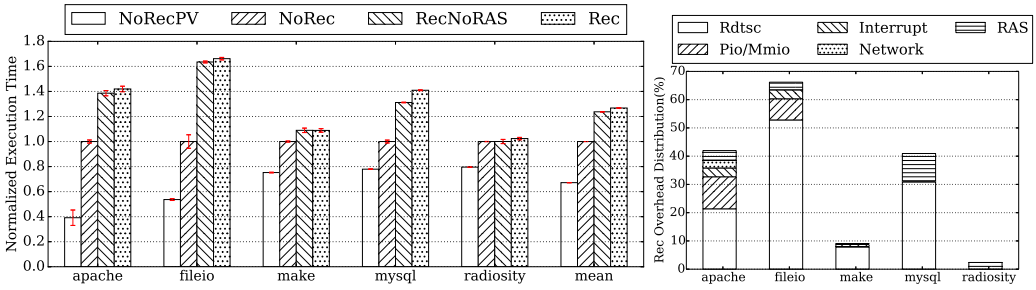
Table II: System configuration for performance evaluation.

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

Table III: Benchmarks executed.

C. Handling Non-Deterministic (ND) 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



(a) (b)

Figure 5: Execution time of recording setups (a) and breakdown of the *Rec* overhead over *NoRec* (b).

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.

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).

D. 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.

E. 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-Safe. We simulate a 48-entry RAS by default.

VIII. Evaluation

A. 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 paravirtualized (PV) network drivers. Figure 5(a) compares the execution time of our scheme (called *Rec*) to that of 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%. 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. Note, however, that RnR has been successfuly applied to PV drivers [24]; applying those techniques in our solution would eliminate this overhead from our system.

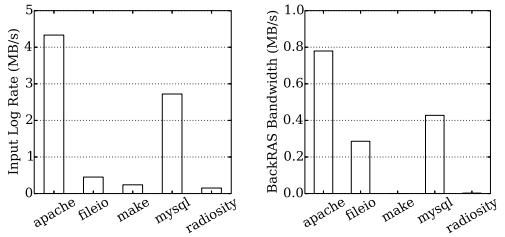
Recording (*Rec*) takes, on average, 27% longer than *NoRec*. Recording without saving/restoring the RAS (*RecNoRAS*) takes 24% longer than *NoRec*. These overheads are modest, and are likely to decrease in a reasonably-optimized implementation of recording—e.g., Pokam et al. [18] measure that their implementation of recording adds only 13% overhead.

To understand the source of overheads, Figure 5(b) takes 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 6(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 of log generation are low. Apache has the highest input log rate (4 MB/s) because it records network packet contents. We also see that the RAS save/restore bandwidth is very small. Overall, the impact of the architecture on the memory system is modest.



(a) (b)

Figure 6: Input log generation rate (a) and bandwidth to save/restore the RAS (b).

B. Minimizing False Alarms

The RnR-Safe hardware eliminates most of the false alarms in the kernel, allowing only a few false alarms to be reported to the replayers. Figure 7 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, mosty because it has some deep procedure nesting under network stress conditions. Both the whitelist and the BackRAS are very effective at removing false alarms.

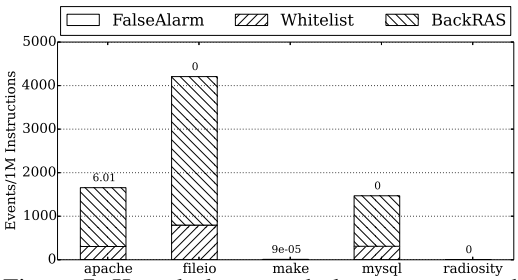


Figure 7: Kernel alarms and alarms suppressed.

C. Replaying

1) Checkpointing Replay: Figure 8(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 nomalized 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 even 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(b) takes the slowdown of *RepChk1* over *Rec* and breaks it down into its sources. The sources are the events that we saw in Figure 5(b) for the recording, plus creating checkpoints (*Chk*). During recording, the *RAS* category 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 VII-C, it requires single-stepping VMExits over several instructions. This is the reason for the overhead of Figure 8(b). It also explains that replaying without checkpointing (*RepNoChk*) already has significant overhead over *Rec*.

2) Alarm Replay: Finally, Figure 9 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.

D. Time Window to Respond to an Attack

The amount of time it takes to detect a ROP is the difference between the time when the alarm replayer confirms a ROP, and the time when the recording execution logged the alarm. Such time window and the length of the resulting log that was generated in between the two times depend on

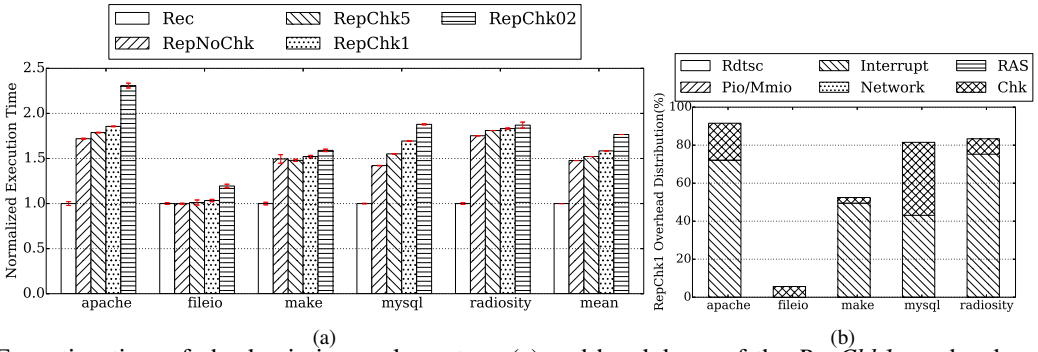


Figure 8: Execution time of checkpointing replay setups (a) and breakdown of the *RepChk1* overhead over *Rec* (b).

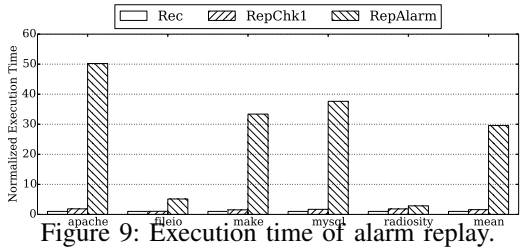


Figure 9: Execution time of alarm replay.

two factors: the workload characteristics and the number of machines dedicated to replay. For the system described, we measured that the time window is on average a few seconds, and the log size several MBs (Figure 6(a)).

The number of checkpoints that the system needs to retain depends on how far back we want execution to roll to fully understand the attack. Strictly speaking, to reproduce the state at the point of the attack, the alarm replayer only needs to start from the most recent checkpoint. Such checkpoint is, at worst, one second old. If that is all that is desired, RnR-Safe only needs to keep as many checkpoints as the duration of the time window mentioned above in seconds—this is to ensure that the correct checkpoint is not prematurely overwritten.

However, if the user wants to analyze the last *N* seconds of execution before the attack was triggered to understand the context of the attack, RnR-Safe needs to keep an additional *N* checkpoints. Finally, checkpoints can be stored indefinitely, if the user wants their entire history recorded. The user can be motivated to do this as the recorded history can be used for forensics and to audit prior executions to detect intrusions.

IX. Related Work

Control Flow Integrity. Enforcing Control Flow Integrity (CFI) [27] is the sound technique to prevent code reuse attacks. It requires preventing branch destinations disallowed by the Control Flow Graph (CFG) and/or the shadow stack. Relaxed approaches [46] avoid the shadow stack and/or CFG by relaxing the definition of valid branch targets. Valid branch targets depend on either the type or location of the destination instruction. For example, Intel CET [54] re-purposes a multi-byte NOP instruction to mark valid destinations for indirect branches. Other approaches [31], [32] define validity via the proximity of the branch destination with respect to function boundaries. In general, such approaches fail to completely eliminate gadgets [46], [62], permitting ROP payload construction. Moreover, shadow stack integrity and longevity of CFI are additional points of concerns for CFI [63], [64].

Address Space Layout Randomization. ASLR hardens systems against ROP attacks by randomizing the locations of the stack, heap, and program instructions. Thus, attackers must first discover the location of the code and stack via address disclosure attacks [53], [52], [51], [65]. This additional requirement compounds the difficulty of mounting ROP attacks. Additionally, to further strengthen ASLR, there are proposals for hardening systems against address disclosure attacks [66], [67], [68]. In summary, ASLR is a practical, effective, and widely deployed hardening technique which indeed makes ROP attacks more difficult to mount. However, until the address disclosure attack surface is eliminated, ASLR cannot fully eliminate ROP attacks.

Record and Deterministic Replay (RnR) for Security. Bezoar [69] uses taint tracking hardware to identify network inputs originating from an attacker. Then, the VM is replayed while skipping the malicious network inputs.

The closest previous work to ours is Aftersight [21]. It suggests using VM-level RnR to perform online dynamic analysis of a system's execution. Although it lays out a general direction for VM-level RnR for online analysis, Aftersight does not address some important aspects of such a model.

For example, unlike RnR-Safe, Aftersight assumes that the full 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, as needed for ROP detection. In addition, RnR-Safe advances the state of the art by proposing an architecture that presents the key practical aspects of online RnR security analysis. These key practical aspects are: (1) Co-designed hardware-software mechanisms (e.g., the RAS extensions are co-designed with the capabilities of the replayers) to achieve reasonable overhead while keeping hardware changes simple; (2) separate checkpointing and alarm replayers; and (3) *need-based* triggering of the alarm

replayers (as opposed to constantly-running analysis).

X. Conclusions

This paper proposed RnR-Safe, a framework where RnR is used to complement a hardware security feature, allowing the latter to be imprecise and suffer false positives. This property often makes the feature design less intrusive or complicated. RnR-Safe uses two on-the-fly replayers: a checkpointing one and an alarm one. As an example, we applied RnR-Safe to thwart ROP attacks to the kernel, the most difficult target to secure. RnR-Safe augments the RAS hardware to eliminate false positives due to multithreading and non-procedural returns. We evaluated RnR-Safe on a VM running Linux. We found that RnR-Safe is a very effective co-design. The checkpointing replayer has comparable execution speed as the recorder, and can be replaying all the time. Also, the alarm replayer has to handle only very few false positives.

Acknowledgment

This work was supported in part by NSF grants CCF 16-29431, CCF 16-49432, and CCF 17-25734.

Appendix A: What is a ROP Attack?

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, and hijacked execution to fetch instructions from there. The W⊕X [2], [4], [70], [71], [72] 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, existing correct code unwittingly provides malware instructions. 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 10 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.

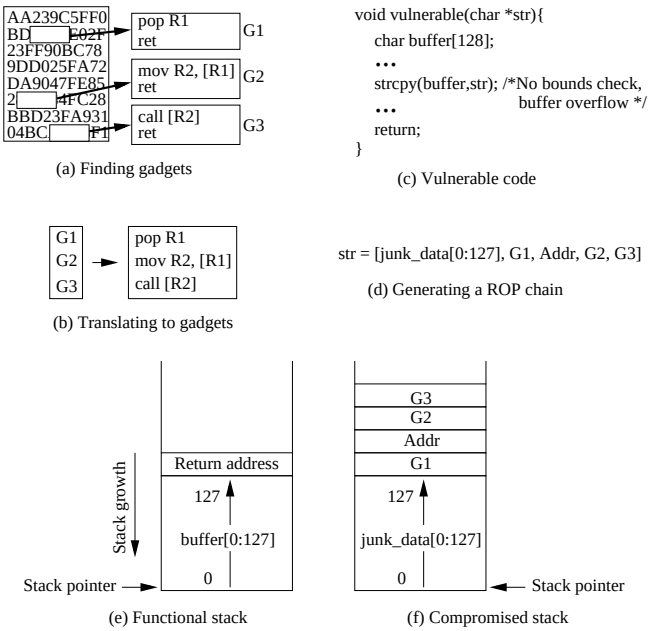


Figure 10: Example of Return Oriented Programming attack.

In Figure 10(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 10(b). The code will result in a subroutine call to a function pointer loaded from a memory location stored on the stack. If this is executed during kernel execution, it can be a call to code giving the user root privileges.

Figure 10(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 10(d) shows how a payload can be constructed to exploit this code to execute ROP malware. Figure 10(e) shows the benign state of the stack, and Figure 10(f) 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.

References

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

[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*, STC '08, (New York, NY, USA), pp. 21–30, ACM, 2008.

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

[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*, CCS '07, (New York, NY, USA), pp. 552–561, ACM, 2007.

[6] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," ISCA, June 2003.

[7] 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, pp. 211–224, Dec. 2002.

[8] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," ISCA, June 2005.

[9] 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*, SOSP '05, (New York, NY, USA), pp. 91–104, ACM, 2005.

[10] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," ISCA, June 2008.

[11] 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*, ASPLOS XIII, (New York, NY, USA), pp. 308–318, ACM, 2008.

[12] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," SOSP, October 2009.

[13] T. Bressoud and F. Schneider, "Hypervisor-Based Fault-Tolerance," *ACM Transactions on Computer Systems*, vol. 14, February 1996.

[14] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," VEE, March 2008.

[15] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comp.*, April 1987.

[16] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," SOSP, October 2009.

[17] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," CGO, April 2010.

[18] 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," ISCA, June 2013.

[19] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "DoublePlay: Parallelizing Sequential Logging and Replay," ASPLOS, March 2011.

[20] 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*, ACSAC '10, (New York, NY, USA), pp. 347–356, ACM, 2010.

[21] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," USENIX ATC, June 2008.

[22] 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.

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

[24] 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*, VEE '16, (New York, NY, USA), pp. 93–106, ACM, 2016.

[25] 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, Sept. 2013.

[26] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, ACM, 2015.

[27] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, (New York, NY, USA), pp. 340–353, ACM, 2005.

[28] 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*, ASIACCS '11, (New York, NY, USA), pp. 40–51, ACM, 2011.

[29] 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, pp. 1271–1285, Oct 2006.

[30] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting hardware architecture to thwart malicious code injection," in *First International Conference on Security in Pervasive Computing*, March 2003.

[31] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *9th Annual International Symposium on Computer Architecture (ISCA)*, pp. 94–105, IEEE, 2012.

[32] 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*, MICRO-47, (Washington, DC, USA), pp. 229–241, IEEE Computer Society, 2014.

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

[34] 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*, pp. 292–307, May 2014.

[35] 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*, ASPLOS '15, (New York, NY, USA), pp. 487–502, ACM, 2015.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing.," in *USENIX Security*, pp. 447–462, 2013.

[37] 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*, pp. 1–12, IEEE, 2012.

[38] 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*, EuroSys '10, (New York, NY, USA), pp. 195–208, ACM, 2010.

[39] 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, pp. 63–78, 1998.

[40] 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*, SOSP '07, (New York, NY, USA), pp. 351–366, ACM, 2007.

[41] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?," in *2012 IEEE Symposium on Security and Privacy*, pp. 571–585, May 2012.

[42] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 475–490, USENIX, 2012.

[43] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "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, (New York, NY, USA), pp. 157–168, ACM, 2012.

[44] P. Team, "Pax address space layout randomization (aslr)," 2003.

[45] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 258–269, IEEE, 2013.

[46] 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.

[47] 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*, pp. 88–108, Springer, 2014.

[48] 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, pp. 91–104, 2003.

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

[50] B. Spengler, "Grsecurity," 2006.

[51] 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*, CCS '04, (New York, NY, USA), pp. 298–307, ACM, 2004.

[52] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*, pp. 191–205, May 2013.

[53] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*, pp. 574–588, May 2013.

[54] Intel Corporation, *Control-flow Enforcement Technology Preview* . No. 253669-033US, June 2017.

[55] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 585–598, ACM, 2017.

[56] 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, pp. 1:1–1:29, May 2011.

[57] 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, pp. 2:1–2:21, Jan 2015.

[58] 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*, CCS '10, (New York, NY, USA), pp. 559–572, ACM, 2010.

[59] "CVE-2015-5364.." Available from MITRE, CVE-ID CVE-2015-5364., Dec. 3 2015.

[60] S. T. King and P. M. Chen, "Backtracking Intrusions," SOSP, October 2003.

[61] R. Senthilkumaran and P. Kulkarni, "Insight: A framework for application diagnosis using virtual machine record and replay," 2014.

[62] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 385–399, USENIX Association, 2014.

[63] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 4:1–4:40, Nov. 2009.

[64] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 161–176, USENIX Association, 2015.

[65] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[66] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1342–1353, ACM, 2014.

[67] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, (New York, NY, USA), pp. 35–46, ACM, 2016.

[68] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 256–267, ACM, 2015.

[69] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. Ye, S. F. Wu, Z. Su, and F. T. Chong, "Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks," in *IEEE Network Operations and Management Symposium*, pp. 121–128, April 2008.

[70] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[71] D. Seal, *ARM architecture reference manual*. Pearson Education, 2001.

[72] P. Team, "Non executable data pages," 2004.