LEVERAGING CONCURRENCY FOR PERFORMANCE AND SECURITY

BY

YASSER SHALABI

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Wen-mei Hwu
Assistant Professor Christopher Fletcher
Assistant Professor Jian Huang
ABSTRACT

In this thesis we explore methods for exploiting concurrency to improve the security and performance of computing systems. We put forth four proposals: the Concurrency Accelerator (ConcAcl), Record-and-Replay Safe (RnRSafe), ReplayConfusion, and ReplayEndurance.

With ConcAcl we accelerate concurrency management operations by creating a dedicated layer that is programmed by supervisor software (e.g. Operating System kernels or multi-threading runtimes). This layer is provisioned with dedicated compute and memory resources which are replicated across all cores in a multi-core processor. ConcAcl hosts procedures which are designed to exploit this unique arrangement to accelerate synchronization-heavy operations that are critical for concurrency. We use ConcAcl to offload functions related to event-synchronization, cross-core remote procedure calls, and task scheduling.

In addition to improving concurrency management we also explore techniques which exploit concurrency to extract security benefits. The difficulty of implementing hardware-enforced security policies is exacerbated by a trade-off between implementation intrusiveness and completeness of methods. Methods which can guarantee detection will often require radical architectural changes. In addition, security systems need to be flexible, as security threats continuously evolve. To help address these requirements, we propose utilizing a novel framework where “Record and Deterministic Replay” (RnR) is used to complement hardware security features. We call our approach RnRSafe. By recording non-deterministic behaviors concurrent replay can be used to investigate potential alarms. Thus, RnRSafe reduces the cost of security hardware by allowing it to be less precise at detecting attacks, potentially reporting false positives. We show how RnRSafe can be used to defend against Return Oriented Programming (ROP) attacks with minimal changes to the processor architecture.

We also propose exploiting concurrent record and replay to enable the detection of otherwise undetectable covert channel attacks using two techniques – ReplayConfusion and ReplayEndurance. Covert channels encode secret values in sub-architectural features like caches and buffers. To detect covert channels we propose techniques similar to our RnR-Safe approach. First, the original instruction execution is recorded. Then, in either offline or online fashion, a replay is performed under a slightly altered configuration designed to alter sub-architectural behaviors. Thus, by comparing the original instruction execution to the
modified replay-time execution, a signal can be extracted which measures the divergence between the recorded and replayed program in order to estimate the program’s sensitivity to sub-architectural behaviors. With ReplayConfusion we alter parameters which organize the last-level cache and with ReplayEndurance we modify those which govern speculative execution. Altogether, this enables the construction of robust defenses against these attacks which can defend systems despite insecure hardware.
For you, without whom I neither would’ve nor could’ve...
ACKNOWLEDGMENTS

First and foremost, I thank the Almighty for the mercy and generosity that I have received in my life. Whatever success that I have realized was only possible through people – far more than can be named here – who helped me throughout my life. That was especially true in my graduate studies which are concluded with this thesis. Family, friends, colleagues, mentors, and more – all were critical to the completion of this PhD. In truth, whatever words I select to fill these acknowledgments will be an insufficient homage. Which words can honor the impact of a gentle and forever patient advisor? Josep is a caring advisor who sincerely tries to prioritize his students and help them benefit from his wisdom. I cannot thank him enough — it is such an honor to graduate as his student! Finally, which words can memorialize the beautiful moments I had with my friends and lab mates? I have been blessed to have such great colleagues and friends — more than I can mention here. My words will definitely fall short, so instead I will say this: your company was a blessing and our memories are precious. Without you I wouldn’t find myself here writing this message. Thank you!

Without my family I certainly wouldn’t have finished this PhD. My mother, Maha Fraitekh, provided the initial push which sent me to Urbana-Champaign. She was the one to inspire me to pursue a PhD when I first began hunting for more interesting work. It wasn’t hard for her to convince me. Throughout my childhood she emphasized that education is the means to elevating oneself. I knew that I must seek higher education if I wanted to achieve something more meaningful and grand. She etched the importance of knowledge deep in my philosophy and for that I thank her. My siblings – Yasmine, Zein, Ahmed, and Noor – and my brother-in-law Abdallah also deserve more thanks than I can give. I am so lucky to have such a dream team in my corner. Yasmine (my big sis) is the epitome of empathy and care – you can’t find anyone who cares more than her. Zein is a fighter; a rebel who will succeed despite anyone in her path – she helped me keep my head up many times. Ahmed, my younger brother, is wiser than his age – he always helped me look beyond grad school. Noor is a bundle of youthful passion and optimism – throughout my PhD her perspective was refreshing. Abdallah is the big brother I never had and he provided me with the role model that I have looked up to since my teenage years. Love you family — and I especially love my amazing nephews Keenan, Adam, and Zayd! Finally, I want to thank my mentor
and friend Dr. Nahil Sobh. Although technically we are not family, this is the right place to thank him because he really has become like family to me. His mentorship – which often came over morning coffee – has shaped my perspectives on many things. Thank you, Dr. Nahil, graduate school wouldn’t have been the same without you!

Finally, I will conclude with a paragraph of thanks dedicated to my dear wife Farah Hariri. I was lucky enough to meet her in my first year in Urbana-Champaign and I was even luckier to have her say yes in my third year! The “yes” she gave me made the third year of grad school one of the best years of my life. With regards to this PhD, she has been nothing but pure support. It is easy for a grad student to end up in a desperate state — in need of support and yet totally oblivious to this need. It comes with the territory of a PhD student. Graduate students who end up in such states must recognize and resolve them alone, but I was lucky to have Farah in my life. She went out of her way to show me the quagmire I was in and to help power me out of it and across the finish line. She was very much the glue that held me together and she did all this while being pregnant with Abdurrahman, our first child, who was born during the final year of my PhD. Through Farah and Abdurrahman I have discovered a profound happiness. They will always be the most precious memory of this journey through graduate school.
# CONTENTS

Chapter 1  INTRODUCTION .............................................................. 1  
  1.1 Concurrency Management ...................................................... 1  
  1.2 Securing Systems ............................................................... 2  
  1.3 Thesis Outline ................................................................. 6  

Chapter 2  BACKGROUND .............................................................. 7  
  2.1 Concurrency Management ...................................................... 7  
  2.2 Coordination Challenges ...................................................... 10  
  2.3 Multivariant Execution ........................................................ 12  
  2.4 Serverless Architectures ....................................................... 13  
  2.5 Record and Replay .............................................................. 14  
  2.6 Return Oriented Programming (ROP) ...................................... 15  
  2.7 Return Address Stack .......................................................... 19  
  2.8 Covert Channels ............................................................... 19  
  2.9 Systematic Attack Characterization ...................................... 19  

Chapter 3  ACCELERATING CONCURRENCY MANAGEMENT .................. 28  
  3.1 A Concurrency Management Accelerator .................................. 28  
  3.2 Leveraging ConcAcl ............................................................. 30  
  3.3 Evaluation ............................................................................ 36  
  3.4 Discussion ............................................................................. 38  

Chapter 4  RNR-SAFE: EXPLOITING CONCURRENT RECORD AND REPLAY FOR SECURITY ................................................................. 45  
  4.1 An RnR Security Framework .................................................... 45  
  4.2 Example: Targeting Kernel ROPs .............................................. 48  
  4.3 Implementation Issues ............................................................ 54  
  4.4 Mounting A Kernel ROP Attack .............................................. 56  
  4.5 Evaluation ............................................................................ 57  

Chapter 5  REPLAYCONFUSION ....................................................... 66  
  5.1 Overall Design ....................................................................... 67  
  5.2 Evaluation ............................................................................ 72
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>REPLAYENDURANCE</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>Detecting Enduring Effects of Transient Instructions</td>
<td>82</td>
</tr>
<tr>
<td>6.2</td>
<td>Software Prototype</td>
<td>86</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluation</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>RELATED WORK</td>
<td>96</td>
</tr>
<tr>
<td>7.1</td>
<td>Security Defenses</td>
<td>96</td>
</tr>
<tr>
<td>7.2</td>
<td>Concurrency Management</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>CONCLUSION</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>103</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Optimizing for performance and resource-efficiency often necessitates exploiting concurrency in software and hardware. In addition, the security of systems can be enhanced by exploiting concurrency to detect and defend against threats. In this thesis we discuss both of these angles. We start by introducing the problems which we address.

1.1 Concurrency Management

The evolution of the modern data-center has been driven by efforts to optimize for client-server applications. This is especially true for networking within a data-center. Within a single rack, nodes are inter-connected using a top-of-rack switch. These high-performance switches enable routing packets between co-located nodes with sub-microsecond delays. Such low-latencies enable access to remote compute, memory, and storage resources at modest performance costs. This has led to the success of the Remote Procedure Call and has in turn driven innovation in Network Interface Cards (NICs) in the form of hardware-terminated protocols (e.g. RDMA, Infini-band, NVMe-OF). The protocols can enable remote resource access without interrupting the host CPU – reducing the cost of the accesses to the point where they are only marginally more expensive than local equivalents.

Parity between local and remote access latencies creates a new challenge for software systems. In particular, single-microsecond scale IO cannot be mediated and managed by an OS kernel which itself introduces microseconds of overhead. The traditional, preemptive OS will not work and will prevent saturating the modern NICs. Present working solutions bypass the OS entirely [1–5] to avoid its scheduling and preemption penalties. Instead, non-privileged software will mediate IO operations and manage concurrency. Concurrency management revolves around selecting when and where tasks run. Ideally, task selection should be coordinated across cores to promote efficiency, fairness, and performance. This coordination requires mechanisms to perform load-balancing, task-stealing, and event-synchronization. The efficiency of these mechanisms is critical for fine-grained concurrency.

Modern processor architectures complicate concurrency management. Coordinating across cores is disruptive to applications. The difficulty in coordinating concurrency stems from
the multicore nature of processors. Minimizing the costs of synchronization and cache-coherence pushes the OS to minimize sharing [6, 7]. This results in the privatization of thread-management data-structures. While distributing structures in a per-core fashion may eliminate coherence and synchronization penalties this approach is not a panacea. Replication and privatization creates challenges when the OS needs to interact with these structures across cores. Balancing thread queues and waking threads – two common functions – will require reading and modifying a remote cores private memory. Aside from violating the share-nothing principle of privatization and replication – this process requires expensive spin-locks. This complicates and limits the kernel’s ability to rely on load-balancing and task-stealing. Sub-optimal performance in these functions results in sub-optimal scheduling [8]. The only way to avoid this violation is through the combination of shared-memory message passing and inter-processor interrupts – a process which is prohibitively expensive on multicore architectures.

Effective concurrency management is critical for performance and efficiency. Typically, this management takes place in a runtime with a global view of scheduling and resource allocations. This runtime is must be kept simple as it imposes a direct overhead on the applications it is servicing. We tackle this problem by accelerating concurrency management by providing it with dedicated on-chip resources.

1.2 Securing Systems

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 [9], AMD’s Secure Processor [10], and ARM TrustZone technology [11].

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 features [10, 12] 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) [13] 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., [14–30]). With RnR, a workload’s initial execution creates a
Concurrent to machine’s execution is a deterministic replay of the workload 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 (e.g., [16, 19]). We explore three uses of this setup: RnR-Safe, ReplayConfusion, and ReplayEndurance.

We first explore an 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 inexpensive 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, we apply 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.

With the mobile era cloud computing has become increasingly used by companies and users alike to minimize the infrastructure cost. This has led to the hardware virtualization paradigm which allows sharing of hardware resources among different users. Thus, mutually distrustful users will execute code and store private data on the same physical machine. This leads to the possibility of a malicious user who leverages the shared physical medium to steal private information. Such attacks are intelligently designed to utilize covert communication channels built on top of contended-shared resources. These channels can leak secrets without leaving a trace and cannot be regulated by a user’s security policy making them very effective.

A covert timing channel is defined as a communication channel that encodes information in terms of timing variations that are detectable by a receiver. Such channels can be utilized by a trojan running on the victim virtual machine, which acts a sender, to communicate sensitive information to a receiver program. Covert timing channel based attacks have been
demonstrated through various shared components such as a system file object [31], network stacks/channels [32,33], and even input devices [34]. In addition, it has been shown that the covert channels can be established even across different virtual machines, through the contention in L2 caches and/or microarchitectural structures [35]. Subsequent work [36–39] enhanced the effectiveness of the attack by improving the communication bandwidth and reliability in virtualized environments.

In order to defend against such cache based covert timing channel attacks, several approaches have been proposed [40–44]. These techniques guard the system by providing better isolation in the caches. However, they are insufficient as they either fail to thwart all possible covert channel attacks, or they introduce an unacceptable performance overhead. For example, CC-hunter [45] detects cache based covert channel attacks by dynamically tracking the conflict patterns introduced by the attacker. This technique accurately detects the covert channel communications that alternate between the sender and the receiver. However, it fails to detect any variation of the attack that does not require alternating send/receive phases [36–38,46]. Hence, we need more robust detection mechanisms that do not rely on a particular communication protocol and can detect any communication between the sender and the receiver.

Understanding the different kinds of protocols employed by cache based covert channel attacks is essential to designing a good defense mechanism against them. So, we first conduct a systematic characterization of the attacks based on the type of protocol they use. We also develop a taxonomy that formalizes these properties. As a result of this study, we make the following key observation.

Any variant of a cache-based covert timing channel attack is highly sensitive to the cache configuration of the system under attack.

In other words, a small change in the cache configuration would result in a significant change in the cache behavior. So, if we deterministically replay the program with a carefully modified cache configuration, then we can compare the cache behavior to the original execution and detect a cache-based covert timing channel attack. We call this approach Replay Conflict Confusion, as we replay the system with a modified cache configuration; it confuses the attack and results in a conflict with the original cache behavior.

The incessant push to maximize the performance of the general-purpose processor has created a modern processor whose immediate behavior is actively shaped by its estimations of future behavior. Recent memory accesses are optimistically cached on the expectation of future reuse. Least recently used entries are pessimistically evicted on the expectation of continued disuse. These are best-effort optimizations because erroneous operations only impact performance and not correctness. Even more aggressive examples exist. Future
instructions are predicted, decoded, and issued to functional-units for out-of-order execution. Correctness requires ensuring a program state that is the product of instructions which were correctly predicted and which respect program order. However prescient the processor may be, incorrect predictions and occasionally premature execution are inevitable. While repairable, the security implications of leaving room for a processor to incorrectly execute was not understood until recently.

With Spectre [47] and Meltdown [48] a new class of attacks to exploit this design choice emerged. On modern processors failed instructions are necessarily under-scrutinized. With “transient-execution attacks” this is exploited to indirectly circumvent security checks. An attacker can systematically induce the execution of instructions which are doomed for discarding in order to maliciously access secrets. This is possible because the security violations of these “transient instructions” are ignored by processors. In the Spectre form attack, the attacker co-opts the branch-predictor to maliciously redirect CPU execution to instructions which read a secret and use it in a leaky manner. From the vantage point of the CPU, these instructions are simply the erroneous consequences of an occasionally incorrect branch predictor. Eventually, the processor will realize its error and discard the results of all instructions which emerged after the branch. Simply discarding these results is not enough to prevent attacks as secrets can still leak through the microarchitectural state of the processor. For example, a secret might have been used to index an attacker-controlled array. By timing the accesses to this array the attacker can recover the value of the secret.

These attacks represent a significant threat to cloud services, browsers, and operating systems. Cloud and browser platforms provide execution environments which allow mutually untrustful software to execute alongside each other. Privileged software (e.g. browser and OS) isolates mutually untrusting applications through sandboxing, access control, and dynamic permission and privilege checks. This is necessary to ensure the safety of applications leveraging serverless cloud services like AWS Lambda or Google Cloud Functions. The functions of serverless applications are hosted in shared cloud-runtimes which take care of scheduling the functions when necessary. The security challenge serverless runtimes face revolves around isolating applications despite sharing both the virtual address space and the physical CPU with other — potentially malicious — applications. The runtime isolates applications through its scheduling, resource allocation, and compilation decisions. Transient execution attacks can circumvent these protection mechanisms to allow malicious serverless functions to steal the secrets of co-located functions.

A variety of approaches have been proposed to mitigate these attacks. The first approach is to neutralize transient execution by compiling software such that transient execution is prevented from being exploited [49] or disabled altogether by serializing instruction streams. The second approach is to prevent the communication from the transient to the enduring by
containing the residual side-effects of speculative execution [50–52]. In other words instructions which fail to commit should not influence those which do commit. The third approach is to ensure that applications which execute untrusted code prevent secrets from being accessed through isolation techniques which are respected by hardware [53,54]. These approaches are expensive and can be difficult to implement (e.g. requiring correctly identifying all secrets, requiring identification of all sources of speculation failures, or necessitating stopping all sources of microarchitectural leakage).

1.3 Thesis Outline

Managing Concurrency: We propose ConcAcl — the concurrency accelerator. With ConcAcl, each core is equipped with dedicated resources which facilitate cross-processor coordination. On each core a special hardware context is used by the OS for performing concurrency management routines. Each context is giving dedicated storage and interconnection resources. Some of this memory is replicated across all cores providing globally consistent broadcast memory. This memory is used by the OS to invoke concurrency-related functions across the multicore. These functions modify private structures in order to fulfill the requested operation. Altogether, this allows the OS to offload coordination-intensive operations like TLB shutdown and futex-synchronization onto ConcAcl. In addition to these functions, we implement Agent Managed Thread Scheduling to offload OS scheduling onto the ConcAcl context. With AMTS, the OS delegate thread scheduling to ConcAcl contexts with low-level access to SMT structures. With this, ConcAcl is able to coordinate fine-grained concurrency across cores in a manner which can minimize QoS violations in client-server applications.

Detecting Covert Channel Attacks: Unmasking covert channel attacks is challenging as these attacks are implemented through indirect mechanisms which are impossible to reliably regulate. We propose ReplayConfusion and ReplayEndurance to detect LLC and transient-execution based covert channels. These techniques work by forcing the attacker to reveal his hand. By recording the original execution a baseline can be established. A concurrently running machine can replay execution under modified conditions which alter the impact on physical resources which attackers use to construct covert channels. In the case of LLC covert channels we modify the mapping between physical addresses and cache addresses. In the case of transient-execution based covert channels we nullify the effects of transient instructions. Modifying these mechanisms creates divergence between the replayed execution and the original recorded execution. We detect this divergence to detect the attacks.
Chapter 2

BACKGROUND

2.1 Concurrency Management

2.1.1 Techniques for Concurrency Management

Concurrency management revolves around scheduling and thread management. Where, when, and how threads are selected for CPU occupancy will impact performance and efficiency. The modern OS preemptively schedules threads. This involves periodic interruption of application execution to context-switch to a new task. Applications can cooperate with the OS and yield the CPU back to the kernel when applications need to wait. The Linux kernel scheduler exposes a variety of scheduling policies to applications for thread scheduling. The Completely Fair Scheduler prioritizes tasks which have consumed the least (virtual) CPU time in an effort to fairly divide CPU time. Alternatively, the FIFO scheduler can accommodate applications which can benefit from simple in-order scheduling policies.

Thread Implementation

Traditionally supported in the OS kernel, threads can be supported in the user-space through libraries, runtimes, and language specifications [55]. Procedures like setjmp/longjmp can be used in lieu of OS context switches. Indeed in recent systems like Go [55], millions of independent contexts are possible. Alternatives to threads have been well studied [56–61]. Some of these eschew OS involvement, promote it [61], or cooperate with it [60]. Circumventing the OS for concurrency management can improve performance but has its limitations. The OS must be aware of concurrency beneath it to avoid sub-optimal scheduling and allocation decisions. For example, during lengthy IO procedures (e.g. page-swaps) the OS can only overlap the delay with threads it is aware of. This is critical for properly specifying thread behavior [56]. This is, in part, the motivation behind increasing involvement of user-space threading in OS functions [57]. We observe that much of the problem stems from the fact that some applications cannot tolerate kernel-switching overheads or because OS scheduling
Figure 2.1: Scalability bottleneck in Linux futex and mprotect operations.
is too coarse grained. If these problems were solved then OS management can be deemed sufficient.

OS Scheduling

Fair, batch, and idle tasks underpin scheduling on Linux systems. In Linux, the Completely Fair Scheduler (CFS) is the default scheduler and implements a policy of fairly dividing CPU time across tasks. Fair tasks divide CPU time fairly in a manner which respects weights. Essentially, each task has a weight $W_i$ and receives a fraction of CPU time proportional to $F_i = \frac{W_i}{W_1 + \ldots + W_N}$. The granularity of these time-slices is governed by the kernel parameter sched\_latency\_ns. Batch tasks are also fairly scheduled but with a larger time-slice granularity. Idle tasks are only scheduled when no other tasks are ready. Hidden behind the relatively simple APIs of Kernel schedulers is significant complexity as the kernel coordinates independent schedulers to maintain load-balance, steal tasks, and wake threads. Indeed, the difficulty of implementing these operations has led to performance bugs which prevented the full utilization of system resources and remained hidden for years [8].

OS Event Synchronization

Futex synchronization allows application threads to suspend their execution until the manifestation of a future application-specific event. A common example of these events is the elimination of contention. Libraries like pthreads use futex operations to suspend threads which attempt to acquire contended locks. Once the contended mutex is released, the library will issue a futex operation to wake the most recent thread. Aside from contention, these events can be available work in a web-server or the completion of an asynchronous RDMA operation. This event-driven pattern is commonly used in highly-concurrent, request-oriented applications like web-servers.

By leveraging threads and event-driven execution web-servers like Apache can hide network latency. By using threads to serve requests, it is able to serve a large number of requests with fewer system resources. Apache HTTP Server always tries to maintain a pool of spare or idle server threads, which stand ready to serve incoming requests. In this way, clients do not need to wait for a new threads or processes to be created before their requests can be served. The Apache process contains a listener thread which listens for connections and passes them to a server thread. The server thread waits for requests, which are passed to workers for processing. Idle works utilize pthread condition variables to wait for available work. Since web requests are often intermittent, at any giving time most worker threads are suspended. This architecture promotes resource efficiency which enhances the throughput
and latency qualities of Apache.

Mitigating the performance impact of asynchronous delays is a critical function for systems. The sources of these delays include contention and IO delay. Operating Systems can opt to suspend blocked threads instead of busy-polling until the blocking subsides. By preempting blocked threads a thread scheduler can switch to threads which can leverage the available CPU resource to perform useful work. Thus, this strategy represents a trade-off between latency and throughput. Suspending threads will degrade latency due to context-switching delays but will promote throughput by overlapping work with delays. With interfaces like epoll and sys_futex Linux applications can take a more active role in this process.

Non-Kernel Scheduling

Concurrency abstractions can be implemented in various layers of the modern system. This choice will determine the cost and the capabilities of both thread scheduling and thread management. When implemented in the Operating System, all management and scheduling operations will require context switches from unprivileged (i.e. user-mode) to privileged (i.e. kernel-mode) execution. These switches are expensive as they involve saving/restoring execution states and because they create contention for microarchitectural resources. To avoid these penalties threads can be implemented outside the OS. Go and C++ – two popular systems languages – take this approach and define OS-independent concurrency abstractions: the Goroutine and Coroutine. These abstractions allow millions of concurrent threads of execution with limited (if any) OS interactions.

2.2 Coordination Challenges

2.2.1 Asynchronous Cross-Processor Execution

smp_call_function API

In some scenarios a Linux kernel thread must execute a procedure on a specific CPU. This process, for example, is required when a kernel thread needs to modify physical structures which are only accessible a remote CPU. The Translation Lookaside Buffer (TLB) is one such example. Remote TLBs can potentially contain stale translations whenever a kernel thread updates a shared page-mapping – e.g. during an mprotect call. In such scenarios the kernel must invoke TLB invalidation functions on remote CPUs. The Linux kernel utilizes the smp_call_function_many coordinate this process by enqueuing call-functions across all
CPUs requiring TLB invalidations.

Broadly speaking, this process follows four steps. First, to dispatch a function to a remote processor, the kernel will atomically enqueue descriptor entries into a per-processor linked list. Next, to trigger the execution of this function the kernel thread will use Inter-Processor Interrupts (IPIs) in order to notify the remote processor of the pending call-function. Upon receiving the interrupt, the remote processor will perform the function (e.g. TLB invalidation sequence). To ensure completion of the call-function, the initiating processor will wait for completion by polling the descriptors for completion status. Figure 2.1b shows the delays encountered when using smp_call_function to coordinate a TLB invalidation function across a many-core. Observe that the efficiency of a whole-processor TLB shutdown is less than 1% due to the limitation of a serial notification mechanism implementing a broadcast operation. This inefficiency will cripple applications which frequently invoke fine-grained functions across processors are unable to rely on Linux call-functions.

Managing Replicated Structures

Processors today are unable to efficiently coordinate updates to private state across processors. Consider changing the permissions of a shared page of memory from read-write to read-only. The natural expectation of the programmer is that the page would become immutable upon completion of the mprotect system call. The Translation Lookaside Buffer – a private cache of page translations – can contain outdated information. To avoid this, the OS kernel must coordinate the execution of TLB invalidations at all cores which may have dirty TLBs. Only then can the page be considered truly immutable. This TLB shutdown process requires interrupting the target processors so that they may update their private TLBs. The cost of this process is linear with the number of processors in the system and can stretch over many microseconds. Applications sensitive to this cost (e.g. tail-latency sensitive applications) are unable to use mprotect as the large cost of page-permission transitions cannot be tolerated. Figure 2.1a shows this linearly increasing cost, as measured on a Xeon PHI many-core.

Beyond managing TLBs, cross-core coordination can also be useful for software structures. For example, threads can be suspended in private structures associated with the core hosting the thread prior to its suspension. When a thread running on another core wakes that thread it will be forced to modify entries in a data-structure it does not own. In this scenario, to avoid synchronization, it would be useful to coordinate the wake-up process across cores. This delegation like process can improve performance by minimizing penalties due to coherence and synchronization. Another example can be found in scheduling on multithreaded kernels. The kernel uses run-queues to hold threads which are awaiting a scheduling slice. Private run-queues are used to avoid lock contention on the kernel context-switching slice.
Using multiple run-queues can lead to load-imbalance and wasted resources. For this reason task stealing is attempted prior to any CPU going idle. In addition, load balancing is used both periodically and opportunistically (when waking threads) to avoid load-imbalance. During these functions remote queues must be locked to ensure consistency.

2.2.2 Event-Driven Execution

Unfortunately, futex operations cannot efficiently mitigate fine-grained delays due to scalability bottlenecks in the Linux kernel. Figure 2.1d visualizes the latency of key steps of the futex operation (y-axis) as thread counts increase. As a reference, on the rightmost part of the figure, we visualize the latency of Non-Volatile Memory (NVM) access, a disk access, and a Wide Area Network (WAN) access. The queue-like structure in which the Linux kernel maintains suspended threads is stored in a hashmap. The latency of indexing this hashmap – using GetWaiterQueue – is already enough to make futex operations ill-suited for mitigating delays due to NVM operations. Importantly, the time from wait to resume (Wait()-to-Resume()) stretches from 3 to several tens of ms. This is a potentially prohibitive latency for datacenter applications where RDMA operations can complete in a few microseconds. From this data, we see that applications subjected to fine-grained asynchronies cannot rely on futex-based event-driven architectures.

2.3 Multivariant Execution

Multivariant execution seeks to enhance the reliability of software systems by leveraging diversity. The essence of the idea is to construct semantically equivalent variants and then execute them while monitoring their execution for divergence. If the variants are receiving the same input then the divergence can be used as an indicator to detect bugs or to discover malicious input. The divergence can be measured in terms of outcome (e.g. system call results) or in execution behavior (e.g. system call patterns). This concept has lead to a large number of research prototypes [62–79]. Although these systems serve a variety of different purposes, they do have some essential similarities.

The typical multivariant engine (MVE) executes at least two variants in lockstep. The monitoring typically occurs in online fashion whereby a monitoring component compares the behavior across variants and distributes inputs to them. In addition, there is the option to select which output gets committed. The most common implementation strategy is to detect for divergence at the system-call interface. Therefore, the typical approach will synchronize variants at each system call and have the monitor compare the system call arguments and
results. This approach enables implementations to leverage kernel tracing capabilities for a clean implementation which does not require source code modifications to the software variants. The drawback is that this comes at a high cost due to the frequent context switching between the attached monitor and the traced application. The runtime overhead is similar to the slowdown experienced when a debugger is attached to an application. Alternative approaches aim to reduce the overheads through binary rewriting [76], virtualization features [67], or kernel modules [64,69–71] to intercept and cross-check system calls without these context switches.

We leverage implement multivariant execution at the granularity of individual cloud functions. In our case, divergence is detected by comparing control-flow traces of individual branches in that function. By leveraging hardware branch tracing we can avoid expensive software instrumentation and we can detect fine-grained divergence.

### 2.4 Serverless Architectures

```javascript
const AWS = require('aws-sdk');
const fs = require("fs");
const s3 = new AWS.S3();
exports.handler = function(ev, context, callback) {
  // Call S3 to list the buckets and save to /tmp
  data = s3.listBuckets(function(err, data) {
    if (!err)
      return [data.Owner.ID, data.Owner.DisplayName];
    return ["/tmp/NoS3","None"])};
fs.writeFile("/tmp\"+data[0],data[1], function(m) {
  if (err) context.fail("Failure: " + m); });
context.succeed("Success");
```

Figure 2.2: Example JavaScript AWS Lambda function. Calls S3 and creates a file with the owner ID and name.

Serverless computing is category of cloud services function/platform as-a-service. These services are used to avoid the efforts of deploying, maintaining, and scaling server platforms. AWS Lambda [80], a “function as a service”, was the first serverless service to see wide use. Since then many other implementations and service providers have increased in adoption [81–83]. The cloud service provider will take charge of deployment management, platform configuration, system management, resource allocation/scheduling to the cloud provider.
Figure 2.2 provides a toy example of an AWS serverless function. The function registers a connection to Amazons S3 service (object storage). The serverless function will make a call to S3 to list the buckets and pass a callback function to return the results. Then, the serverless functions will create some files in /tmp. Every function is allocated some limited space for files. The typical use-cases for serverless functions rely primarily on services for persisting files instead of the temporary storage.

The trend to move to serverless application design is expected to continue [84, 85]. Aside from reducing development effort, this also provides applications with an economic advantage as they are able to share the costs of the traditional computing stack (OS, scheduling, network) with other serverless customers. Serverless applications consist of many independent functions which are implemented in popular languages like JavaScript, Python, and Go. These functions are stateless and are scheduled on-demand in an event-driven manner. The events trigger these functions often flow through services provided by the cloud provider (e.g. AWS S3 or AWS DynamoDB). In this way, application functionalities are implemented by composing independent functions which communicate through cloud services. Application specific events will trigger the execution of a function which can then trigger subsequent execution. Functions from different serverless applications are cooperatively scheduled on a cloud service runtime process. This means serverless applications are less isolated than traditional (e.g. virtual machine) cloud applications. For this reason attacks like Spectre – which allow attackers to bypass software isolation mechanisms – represent a significant threat to serverless applications.

2.5 Record and Replay

Record and deterministic Replay (RnR) of workloads is a popular architectural technique (e.g., [14–30]). 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 [15, 16, 22, 23]. 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 [86] or Qemu [87]. These assumptions are not necessarily limitations, as RnR approaches compatible with multiprocessing [17, 29] and virtualized I/O [88] exist.
There are several papers that investigate the use of RnR in a security-related scenario (e.g., [16, 19, 21, 22, 27, 89]). ReVirt [16] shows an example of using VM-level RnR for post-facto offline analysis of time-of-check to time-of-use race conditions in the Linux kernel. IntroVirt [19] explores using VM-level RnR to determine if systems were previously exploited once zero-day attacks are discovered. Speck [21] explores using a combination of OS-level speculation and program-level RnR to remove security checks from the critical path of a program. ParanoidAndroid [27] and Secloud [89] explore the possibility of maintaining replicas of mobile devices in the cloud, and perform program-level RnR in the cloud. Finally, Aftersight [22] 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 7.1.

2.6 Return Oriented Programming (ROP)

RnR-Safe is a general framework that can be used to thwart a variety of attacks. As an illustration, in our proposal, we consider ROP [13] attacks, focusing on those on the kernel. Appendix A describes ROP attacks.

At a high level, ROP attacks can be detected with a structure called Shadow Stack (e.g., [90–93]). When a call instruction is executed, the address of the instruction following the call is pushed to the shadow stack. At return instructions, the top of the shadow stack is popped. 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), multicontext (e.g., the kernel), or imperfectly nested (e.g., error and exception handling). Since false positives are unacceptable, proper handling of all these corner cases is necessary. Most solutions add instructions to manipulate the shadow stack in these cases. Such instructions, even if privileged, provide a security vulnerability.

There have been a variety of approaches that focus on protection against ROP attacks (e.g., [90–103]). 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.
2.6.1 Return-Oriented Programming Attacks

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 policy [10,12,104–106] 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 [13] 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 [13], 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 2.3 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.

In Figure 2.3a, 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 2.3b. 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 2.3c 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 2.3d shows how a payload can be constructed to exploit this code to execute ROP malware. Figure 2.3e shows the benign state of the stack, and Figure 2.3f its state after being corrupted by the malicious input string. Now, returning from the vulnerable function takes us to G1, which
void vulnerable(char *str){
    char buffer[128];
    ...
    strcpy(buffer,str); /*No bounds check, buffer overflow */
    ...
    return;
}

str = [junk_data[0:127], G1, Addr, G2, G3]

2.6.2 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 [90,91,97,100] require intrusive hardware changes. For example, both SmashGuard [91] and SRAS [90] add a hardware stack used to verify the return targets. The hardware needs to be very carefully designed, as it intimately interacts with the speculation mechanism of modern processors [91]. In addition, it uses instructions in case of stack overflow and context switch [90,91]. Such instructions, even if privileged, provide a security vulnerability. The PUMP [100] 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 [97] 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 slow-downs.

Software Impact. The completeness of instrumentation-based CFI-enforcing solutions such as [94,95] is attractive. However, securely maintaining a shadow stack at call/ret boundaries via binary instrumentation adds overheads of over 100% [95]. If the source code is available, compiler based solutions can reduce the overhead substantially [107]. Other approaches [99, 108] 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, this approach also requires source code, which limits its applicability.

Completeness. Other proposals explore alternative approaches by either detecting ROP attack characteristics [98,101,102] or by making the ROP attacks difficult to mount [109–112]. By monitoring control flow characteristics that are indicators of ROP execution [98,101,113], 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 [114,115]. Similar criticisms can be levied against other proposals which provide probabilistic defenses through randomization [112] and encryption [116–118]. These defenses will randomize code locations or encrypt on-stack return addresses, thus complicating critical steps in attack code. In particular, Address Space Layout Randomization (ASLR) [109–112] is a widely deployed defense adopting this approach. While useful in the short-term, these approaches remain insufficient in the long-term, as attackers have learned to circumvent them [119–121]. We discuss ASLR in more detail in Section 7.1.

Recently, Intel has introduced Control-Flow Enforcement Technology (CET) [93], which has shadow stacks that verify return targets. The shadow stack pages can only be accessed through special privileged instructions. However, to our understanding, since these shadow stack pages are in memory, and because they can be accessed by certain privileged instructions, it is conceivable that an attacker can subvert the system and update the stacks. The same concern arises for Griffin [122]—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 our proposal, we use RnR-Safe to support ROP protection in a different manner, with a great deal of flexibility, and a different set of tradeoffs than prior methods. In particular, we have no hardware shadow stack. Effectively, the replayer in a secure machine implements a shadow stack in software.
2.7 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. The RAS is not accessible by software. The IBM POWER7 [123] and POWER8 [124] processors have a RAS with 32 and 64 entries, respectively.

ROP attacks cause RAS mispredictions because the attacker forces a return to an unexpected instruction. However, a RAS misprediction cannot by itself be used as an indicator of ROP attacks because the RAS sometimes mispredicts in the course of benign program execution.

2.8 Covert Channels

2.8.1 Cache-based Covert Channel Attack

Virtualization of physical memory prevents direct communication across processes without explicitly mapping shared memory or by utilizing OS mediated inter-process communication. In cache-based covert channel attacks, sender and receiver communicate indirectly by leveraging the variations of cache access delay. Most cache based covert channel research targets LLC as it is the larger attack surface.

Prime+Probe [36, 125, 126] is a general technique used by the receiver to figure out which sets the sender has accessed in cache. During Prime, the receiver fills selected cache sets with its own data. Then it turns Idle, during which the sender will either evict the receiver’s data or leave it in the cache. The choice the sender makes will depend on the message being sent. Later, the receiver will conduct a Probe by accessing the same addresses accessed during Prime. The receiver will learn the sender’s decision by timing its accesses to the addresses.

2.9 Systematic Attack Characterization

The information encoding scheme and the timing of Prime+Probe can be varied, resulting in different possible communication protocols (listed in Table 2.1).

In covert channel communication attacks, a sender and receiver will agree on a protocol to create and interpret timing differences in memory accesses. In this section we introduce the taxonomy we developed to categorize and understand the fundamental decisions that go into
the design of these protocols. To design effective detection hardware and methodology, it is important to understand these aspects to appreciate the behavioral differences possible between different attacks — and the impact such differences can have on detector effectiveness.

Figure 2.4 overviews this taxonomy. The two primary characteristics revolve around time and space. We will describe those two characteristics and conclude with a characterization of existing attacks according to our taxonomy.

2.9.1 Cache Time Factors

The timing part of the protocol decides the ordering requirements between sender and receiver channel interactions. Two approaches have been proposed in the literature: round robin and parallel.

Round Robin

The round robin protocol requires a strict ordering between the sender and receiver; therefore, no concurrent accesses happen.

The top of Figure 2.5 visualizes the round robin protocol. The sender will send a bit once
every $T_s$ time units. This window of time is larger than the time it takes to encode the bit. The remaining time is needed to properly synchronize with the receiver to meet the ordering requirements of the protocol. This is critical in order to ensure that sufficient time is given to the receiver to be scheduled so that it can decode the bit. Likewise, the time of the read $T_r$ must also be longer than the time it takes to receive the bit to synchronize the receiver with the sender. Correctly setting this synchronization makes the round robin protocol the most challenging to implement — especially in virtualized environments [38].

Due to the strict ordering, the round robin protocol results in an alternating pattern between sender and receiver, which exposes it to possible detection.

Parallel

The parallel protocol removes the ordering restrictions and allows concurrent accesses between sender and receiver. This is shown in Figure 2.5.

The parallel timing protocol [38] has been proposed to avoid the need to synchronize sender and receiver. The concurrent accesses sometimes introduce errors, which can be overcome by transmitting and decoding a bit multiple times. The encoding time period is intentionally extended to guarantee that the receiver will get several correct measurements of the bit.

Due to requiring repeated operations to transmit and receive one bit, this protocol enlarges the attack footprint.

### 2.9.2 Cache Space Factors

Receiver uses **Prime + Probe** to decode a message that is encoded by sender. The space factor defines how sender and receiver choose addresses to create conflicts. Knowledge about the cache system can be leveraged by the protocol to improve bandwidth and reliability and to minimize the footprint of the attack.

In this section we will define and describe two aspects of the spatial description of the covert channel protocol: group count and mapping strategy.

#### Group Count

A group is a set of several cache sets. The state of a group can be either cached or flushed. To put a group in the flushed state, the sender will evict all the receiver cache lines that were previously cached in it. To put the group in the cached state, the sender will make no accesses to the group, thus leaving all receiver cache lines in it.
<table>
<thead>
<tr>
<th>Category</th>
<th>round robin</th>
<th>parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>multigroup, aware</td>
<td>[35], [39]</td>
<td>[36], [46]</td>
</tr>
<tr>
<td>single-group, aware</td>
<td>-1</td>
<td>[38]</td>
</tr>
<tr>
<td>single-group, unaware</td>
<td>-1</td>
<td>[37]</td>
</tr>
</tbody>
</table>

Attackers can use a single group to communicate and receiver distinguishes the bit based on the two possible states of the group. In order to detect when a bit is not sent, at least one additional group is needed. The inability to detect the lack of a bit is a significant drawback to using a single group as it leads to high bit error rates.

Using multiple groups allows using techniques like differential coding to improve reliability. The differential coding technique [35] represents the bit as the difference between two groups. This improves reliability, since the noise will be removed by the difference operation. The additional groups can also be used to implement multibit transmission to improve bandwidth. To be able to define multiple groups, the protocol must understand how to distinguish addresses based on which set they resolve to.

**Mapping Strategy**

As discussed in Section 2.9.5, attacker needs to leverage knowledge about the cache address translation process, i.e. set index function and slice hash function, to figure out which slice and set an address maps to. In this way, attacker utilizes what we categorize as cache distribution aware protocol to divide group or achieve controllable cache conflicts between the sender and receiver.

Alternatively, attacker can use what we call cache distribution unaware protocol by allocating a large enough buffer to achieve measurable conflicts. The main drawback of cache distribution unaware protocol is attacker can only use a single group to communicate. Moreover, it is difficult to choose the threshold to distinguish cached state set by sender and system noise, because total number of conflicts varies from execution to execution.

2.9.3 Categorize Existing Attacks

Protocols in [35, 39] use round robin protocols and two groups to communicate. Their two-group setup is a common one where one group contains the odd sets of the LLC while the other contains the even sets. A parallel protocol communicating multiple bits concurrently was demonstrated in [36]. The protocol used in [46] is a parallel protocol which uses techniques to infer the processor’s slice hash function to minimize the number of accesses required. A
parallel, single-group protocol [38] used a subset of the LLC by choosing addresses based on
the processor’s set index function. A parallel, single-group protocol [37] was shown to be
successful when the sender time period was much longer than the receiver.

2.9.4 Challenges in Covert Channel Detection

Communication occurs by creating conflict misses. A naive detector might try to look for a
recurrent pattern of spikes in the cache miss rate during program execution. This approach is
ineffective as such repeated cache miss patterns occur quite often in benign programs. This is
especially true for programs which have alternative memory access intensive and computation
intensive phases.

As discussed in Section 2.9.6, CC-Hunter is designed to detect alternating patterns of cache
conflicts between sender and receiver. CC-Hunter logs the processor ids of the victim cache
line along with the id of the memory access causing the eviction. Round robin protocols
cause an alternating pattern within this log because the sender and receiver alternate.
Parallel protocols allow concurrent cache access from sender and receiver. This eliminates
the alternating pattern of conflicts signal that CC-Hunter detects. We verified this against
parallel protocol based attacks which all successfully evaded the detection of CC-Hunter.

In this chapter we aim to address this coverage gap. We propose new techniques capable of
detecting both round robin and parallel protocols using only cache miss profiles constructed
using common performance monitoring hardware. These techniques work because they enable
measurement of the spatial characteristics of covert channel protocols.

2.9.5 Relevant Aspects of Modern LLC Design

Caches are divided into cache blocks, and organized in sets and ways. For performance
reasons, the last-level cache (LLC) is often further subdivided into slices — each identically
configured. To access the LLC, hardware will translate an address into a slice ID, a set index,
and a physical tag. Conflict only happens between addresses that mapped to the same slice
and same set.

The complicated translation process in modern LLC introduces address uncertainty for
attackers in two ways. First, because LLC is generally physically indexed and physically
tagged (PIPT), attacker does not have full control over the index of the set an address is
mapped to. Second, the undocumented slice hash function makes attacker difficult to figure
out the slice an address is distributed to.

Several techniques have been proposed to figure out the specific set an address maps to.
First, large pages, i.e. 2MB page size, are preferred over smaller pages. This is because with larger pages, more virtual address bits are unchanged by translation. In Figure 2.6, we show the address translation process for a 2M 16-way cache. Bit 6-16 is used as set index. When using a 2Mb page, all the bits that decide the set index are within virtual address page offset — putting them under attacker control. When a 4K page is used, only the lower 6 bits can be controlled by attacker.

Secondly, research has developed techniques to assist attackers. The slice hash function can be reverse engineered or conflict sets can be constructed without requiring knowing the exact function implemented by hardware. This can be done by leveraging common per-slice performance counters [127] to detect misses to particular slices. The authors propose an automatic algorithm to reverse engineer the function by this information. Since variation in distance between cores and slices exists, recent work has shown that slice id can be determined by detecting access time variations when accessing cached addresses [128]. Both approaches above need to access a specific system file to determine the physical frame number of tested addresses. Other approaches propose techniques to construct precise address conflict sets [46,129] without knowing the slice hash function. All approaches need to either read performance counter data or read timer difference.

2.9.6 Timing-based Covert Channel Detection

Many of the existing covert timing channel detection techniques are designed to apply statistical methods or pattern recognition in timing analysis. The classification is based on certain timing features that can differentiate an attack from a legitimate program. For example, in the distribution system field, where the message is encoded in inter-packet delays (IPD), features like mean and variance of IPDs [130,131] have been used. More advanced techniques such as corrected conditional entropy (CCE) [132] use high-order entropy rate to
recognize repeated patterns in attack traffic.

Since cache-based covert channel is constructed by creating a conflict pattern between sender and receiver, timing features of conflicts can be used for detection. CC-Hunter [45] is the first scheme proposed for detection of covert timing channel in shared caches. LLC is augmented with a “conflict miss tracker” to obtain a cache eviction event train for program execution, where each event is characterized by set conflict direction. For example the event $C_s \rightarrow C_d$ is interpreted as “core $C_s$ caused an eviction of data inserted by core $C_d$”. For covert channel communications with alternating sender and receiver access to the covert channel, a recurring pattern of $S \rightarrow R$ events will be followed by similar amounts of $R \rightarrow S$ events. We show later that CC-Hunter is incomplete in coverage as it cannot detect common attacks where the sender and receiver do not alternate access to the communication channel.

2.9.7 Record and Replay in Security

Record and deterministic replay is a popular architectural technique, which can be conducted at various abstraction layers, e.g. hardware [20, 29, 133–135], operating system [88, 136], and application [24, 25] level.

Prior research has explored the usage of RnR for security [16, 19, 21, 89, 137] to improve program security, offload costs of security checks, or to analyze intrusions. All these techniques utilize functional RnR, which only requires the resolution of functional non-determinism. The correctness of these techniques does not depend on executing recorded execution in a way faithful to timing.

Timing deterministic replay (TDR) [138] is the first work to enable reproducing not only functional behavior of a program, but also the timing behavior. It is achieved by mitigating or eliminating various timing noise from scheduler uncertainty, paging and interrupt etc. In addition, the authors utilize TDR to detect network timing covert channels. They do this by comparing the network packet timing characteristics observed during record to ones produced using trusted applications on a different machine. Significant deviations between the two observations manifest if the recorded application used timing channels.

2.9.8 Transient Execution Attacks

Modern processors rely on optimizations which will periodically produce incorrect execution. To maintain correctness the processor will buffer instruction side-effects until they are absolutely safe to be performed. If erroneous execution is detected, a repair process is undertaken to roll back incorrect instructions and flush the pipeline. A variety of reasons can trigger these flushes. Interrupts and dynamic faulting conditions can emerge forcing
Transient Execution

Setup

Side-Channel Recovery

Figure 2.7: Transient execution attacks follow three phases. First, setup to prepare for and trigger the attack. Next, incorrect speculative execution results in transient instructions which access secrets. Finally, the secret is recovered by attackers via the residual microarchitectural side-effects of transient instructions.

the rollback of prior instructions. Branch targets are predicted to enable speculating past conditional branches. Sometimes the branch targets are incorrect. This results in the execution of instructions which were never allowed according to the program. These instructions cannot be allowed to affect program state.

As instructions execute they also impact the microarchitectural state. This state is reflected in buffers, caches, and various structures which are related to the implementation of the processor. Although not directly accessible through instructions, this state can be estimated through timing side-channels. For example, the residency in the Translation Lookaside Buffer (TLB) can be determined by monitoring memory access latencies for delays. Other examples include the caches and the availability of functional units. Instructions which fail to commit may impact these structures, yet these changes are not rolled back. These residual effects were previously thought to be inconsequential but were recently shown to be exploitable.

With the emergence of transient execution attacks [47, 48] the residual impact of in-flight instructions on the microarchitectural state was shown to be exploitable. Attackers could manipulate hardware into accessing program secrets using instructions which are destined for failure. These transient instructions are under-scrutinized by hardware, allowing attackers to circumvent security checks which isolate mutual untrusting applications. Attackers can systematically exploit this vulnerability to enable unrestricted access to program and/or system secrets. This vulnerability is particularly threatening in environments like browsers and cloud-computing platforms. In these environments software is frequently sharing physical and virtual resources with potential attackers and, as such, depends on the ability of hardware to remain faithful to the security checks which prevent unauthorized access to program secrets.
General Attack Strategy

Figure 2.7 explains the general flow of transient execution attacks. The attacker starts by setting up the attack. There are two aspects requiring setup. First, the microarchitectural state needs to be brought to a condition which is conducive to side-channel attacks. Caches and buffers are filled or cleared of data as necessary to enable the future side-channel attack. Second, to trigger the transient instruction execution the branch predictor must be set up in a manner which will result in attacker-controlled mispredictions. This also involves manipulating the hardware into a state which will result in the issuance of instructions which are incorrect and doomed to fail. These transient instructions are victim instructions being reused as gadgets. To be useful, these gadgets must access secrets in a leaky manner. To begin the next phase of the attack (the transient execution) the attacker will trigger hardware to issue these instructions. The nature of these setup phases depends on the nature of the vulnerability (i.e. the victim code).

The next phase is the transient execution phase (3). Here, in (4) the processor will err and begin executing illegal instructions. As part of the execution of these instructions, the processor will access secrets and act on them (5). Acting on the secret is critical as it will leave a microarchitectural residue which is secret-dependent. This is the residue which leaks information (bits) about the secret. Eventually, hardware will catch up to its mistake, roll back the transient instructions, and finally resume executing from the correct instructions (6).

In the final stage of the attack, the attacker executes instructions to recreate the transiently accessed secrets. These instructions work in tandem with the prior stage to enable a reliable transmission of information from the transient instructions to the recovery instructions. The transmission is built on a side-channel protocol. The inputs to the transient instructions are specifically chosen to leak information in a manner which can allow reverse engineering the secret. For example, the secret can be used to index into an attacker-controlled array. The recovery instructions can discover the index by timing accesses to the array to recover the index. Novel strategies for implementing these attacks have been steadily emerging [139–142].
Chapter 3

ACCELERATING CONCURRENCY MANAGEMENT

The ConcAcl design allows the OS to offload functions sensitive to context switches onto a specialized hardware context with special access to dedicated hardware resources. Each core contains two pages worth of dedicated storage. One page acts as a private scratchpad while the other page is replicated across all cores. Updates to variables in the replicated page are broadcast using a dedicated interconnect. The interconnect is modestly provisioned to enable whole-chip communication with sub-microsecond latency characteristics. This allows coordinating the behavior of the ConcAcl cores through message-passing. Altogether, these resources accelerate fine-grained cross-core coordination. We exploit these capabilities to implement functions related to event-synchronization, task scheduling, and TLB management.

3.1 A Concurrency Management Accelerator

3.1.1 Overview

Figure 3.1 visualizes ConcAcl. For each core, a concurrency accelerator is available to coordinate execution across other OS threads. For example, consider the scenario visualized where the NIC deposits packets monitored by a thread whose execution is pinned on a core (1). This thread will monitor NIC queues, pre-process them, and construct batches of packets for other workers that process the packets to completion. Once the batch is constructed the monitor will wake a worker suspended on a remote core. This wake operation will leverage ConcAcl (2). Later, in (3) the thread is scheduled and the packet is processed.

3.1.2 Programming Interface

ConcAcl execution can be triggered by application threads in a manner similar to system calls. Each core contains registers through which requests are made to ConcAcl. Upon programming one of these registers, the ConcAcl context is activated and begins scheduling instructions from an OS defined location which contains the code to handle local ConcAcl
requests. The code contains instructions which determine the nature of the request in a manner similar to a traditional system call. During the execution of the requested operation the ConcAcl thread may need to coordinate its execution with remote ConcAcl threads. For example, the requested operation might require updating a distributed data-structure. Performing these operations leverages a software defined interface which can trigger remote ConcAcl execution.

3.1.3 Invoke: The ConcAcl Approach

Lock-synchronization is the standard approach to coordinating access to shared data-structures. Instead, with ConcAcl, message-passing is used to invoke the execution of procedures on threads which own the data-structures. This approach was inspired by microkernel OS designs. We call this primitive invoke and it is the basis for implementing coordination-intensive operations in ConcAcl.

To implement invoke the OS will reserve part of the global region to hold descriptors of function requests. To invoke a function across cores, a ConcAcl thread will reserve one of these descriptors, set it to indicate the requested function, and finally increment the command counter. Each ConcAcl thread will monitor this counter which stores the total number of invoke operations. This counter is stored in global ConcAcl memory. When the counter is modified, ConcAcl threads are activated and begin scheduling instructions. Thus, sequentially consistent tracking of commands is possible.
Table 3.1: ConcAcl operations.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke(F,D,M)</td>
<td>Execute function F with data D on the domains given by mask M</td>
</tr>
<tr>
<td>Ack(F)</td>
<td>Acknowledge invoked function F to the sender</td>
</tr>
<tr>
<td>Wait(F)</td>
<td>Wait for function F to be acknowledged by the receivers</td>
</tr>
<tr>
<td>Acquire(L)</td>
<td>Acquire lock L</td>
</tr>
<tr>
<td>Release(L)</td>
<td>Release lock L</td>
</tr>
<tr>
<td>Drain</td>
<td>Fence for coordination operations. Wait for all pending operations on this domain.</td>
</tr>
</tbody>
</table>

3.1.4 ConcAcl Coordination Commands

Table 3.1 lists the coordination operations which we used. Invoke will asynchronously launch a function for execution across the multicore. Examples can include TLB shootdowns, load-balancing requests, or futex operations. Ack and Wait are used to block until invoked operations. This is needed when the completeness of a particular invoked function is required for the correctness of subsequent execution (e.g., a TLB shootdown which must be completed prior to subsequent memory accesses). Drain will block execution until all prior functions are completed. We use this to prevent races with ongoing functions. Acquire and Release are used to acquire and release spin-locks.

3.2 Leveraging ConcAcl

There are two general approaches to leverage ConcAcl. The ConcAcl hardware context monitors the shared memory for coordination commands. This shared memory can be modified by software to broadcast coordination intensive operations across the multicore. With this approach, scalability limiting procedures in the OS kernel (e.g., smp_call_function_many) can be accelerated to improve the performance of the OS. The second approach is for the OS to extend the ConcAcl context to directly provide coordination intensive services without full kernel intervention. This is, in essence, similar to microkernel-like approaches. This can allow the OS to customize the implementation of these services in a manner which can more fully leverage ConcAcl.

Table 3.2 lists the ways in which we leverage ConcAcl. The first application is a direct use of invoke to replace smp_call_function_many for TLB shootdowns. The second application replaces the kernel sys_futex operation with a ConcAcl equivalent. To avoid spin-locks we distribute the threads across ConcAcl contexts. The final four applications replace key
## Table 3.2: Coordinated operations can originate from the kernel-space or the user-space.

<table>
<thead>
<tr>
<th>Operation；Origin</th>
<th>Steps (Initiator Core)</th>
<th>Steps (Other Core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalidate TLB entries</td>
<td><code>OS</code></td>
<td><code>Invoke(TLB_inval,TLB_entries,Mask)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>TLB_inval locally</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Drain</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Wait</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Suspend/wakeup</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Ack</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Enqueue/dequeue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Ack</code></td>
</tr>
<tr>
<td>Suspend a thread or wake-up one or more threads in a remote queue</td>
<td><code>App</code></td>
<td><code>Invoke(Suspend/wakeup,threads,Dst)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Wait</code></td>
</tr>
<tr>
<td>Enqueue/dequeue one thread to/from remote queue</td>
<td><code>OS</code></td>
<td><code>Invoke(Enqueue/dequeue,thread,Dst)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Wait</code></td>
</tr>
<tr>
<td>Enqueue a set of threads tagged with queue IDs to these queues</td>
<td><code>OS</code></td>
<td><code>Invoke(Distribute,threads,Mask)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Enqueue threads in local queue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Wait</code></td>
</tr>
<tr>
<td>Obtain work for the local queue</td>
<td><code>OS</code></td>
<td><code>Read remote queues. Identify 1 busy core</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Invoke(Dequeue&amp;callback,threads,Dst)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Dequeue threads from local queue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Invoke(Enqueue,threads,Src)</code></td>
</tr>
<tr>
<td>Global rebalance of run queues</td>
<td><code>OS</code></td>
<td><code>Find N remote queues with W inviolable threads</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Acquire(Rebalancing_lock)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Dequeue W/N threads</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Ack</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;N busy cores&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Enqueue W/M threads</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Ack</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;M-1 idle cores&gt;</code></td>
</tr>
</tbody>
</table>

Components in the kernel scheduler with ConcAcl equivalents. By replacing these components with ConcAcl equivalents scheduling can be managed by the ConcAcl context.

### 3.2.1 ConcAcl Event Synchronization

The goal is to enable low-latency futex operations using ConcAcl commands. The main challenges lie in ensuring FIFO wakeup order while minimizing communication. To overcome these challenges we maintain lists of suspended threads at each agent. To ensure a FIFO wakeup order a table of ticket locks is maintained (Figure 3.3). The table is indexed by the hash of the futex address. Two eight-bit counters are maintained. “NT” (next-ticket) indicates the wait position of the next thread to be suspended. “CS” (currently-serving) tracks the number of threads which were woken. The key advantage of our approach is that it can eliminate futex operation broadcasting. No global communication is necessary when futex operation pairs originate from within the same concurrency domain. Collisions on futex entries are resolved through the “Next” field. This serves as a pointer to the next entry in the field, as seen by the collisions between `Addr0` and `Addr2` in Figure 3.3. Collisions are resolved by preserving the next sequential entry and setting the Next field.

**Futex Wait**

A ConcAcl based futex wait will first check the futex table (2). An entry is allocated if necessary. If an entry cannot be allocated the OS implemented `futex_wait` (16) is used.
```c
void concal_futex_wait(faddr, val) {
    status = check_futex_table(faddr)
    if (status == NOT_FOUND) {
        __futex_alloc(faddr, val);
    }
    status = check_futex_table(faddr)
    if (status == PRESENT) {
        ticket = grab_ticket(faddr)
        waiter_struct = local_enqueue(faddr, ticket);
        if (*faddr == val) {
            waiter_struct.status = WAITING;
            resched_cpu();
        }
        resume()
    } else {
        __sys_futex_wait(faddr, val);
    }
    done;
}
```

Figure 3.2: ConCaI based Futex Wait

When an entry is available a thread can be suspended by acquiring a wake ticket (8-12) and allocating a private waiter-entry. Waiter entry simply contains a pointer to a thread structure, the futex address, and the value of the wake ticket. A successful suspension is concluded by rescheduling the CPU (13). Success is determined by whether the futex value at the time of suspension matches expectations. As an optimization, this check can also occur prior to allocating entries in the futex table and the local waiter structures. If the futex value does not match the expected value the suspension fails. The waiter_struct is left invalid and control is returned to the user application (14). Later, when the requisite number of wake operations is received, a failed wait is detected at the local core. The invalid waiter_struct can be garbage collected and the wake operation is repeated to wake the next thread in the suspension order.

<table>
<thead>
<tr>
<th>Next</th>
<th>Addr</th>
<th>NT</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Addr0</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Addr1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Addr2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Addr3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>Addr4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.3: Futex ticket lock table (kept in broadcast memory).
```c
void wake_tct(faddr, tct) {
    status = local_queue_check(faddr, tct);
    if(status == PRESENT) {
        int IDX = local_dequeue(faddr, tct);
        mark_rdy(IDX);
        increment_curr(faddr);
    } else if( status == FAILED_WAIT ) {
        local_dequeue(faddr);
        increment_curr(faddr);
        invoke(wake_tct, faddr, tct+1);
    } // No waiter structure locally found.
}
```

Figure 3.4: ConcAcl wake ticket lock implementation.

Futex Wake

The CS counter (currently-serving) indicates the last waiter to have been awoken. To launch a distributed wake operation a wake ticket is acquired by performing an atomic pre-increment on CS (++CS). Then, invoke is used to execute wake_tct across all ConcAcl contexts with the arguments being the futex address and ticket value. On receiving this request, each ConcAcl will execute wait_tct (Figure 3.4). First, ConcAcl will check its private, local waiter structures to search for the suspended thread (2). When the thread was found to be suspended locally it is dequeued from the local waiter queue and marked as ready (4,5). If a failed waiter struct is found, then it is dequeued, the CS counter is incremented, and another wake operation is invoked (8-10). Our approach has two key advantages. First, the elimination of the context-switch and the use of private-local structures enable low-latency wake. Unlike the OS based approach which takes many microseconds the ConcAcl approach takes fractions of a microsecond. Second, unlike the OS approach, which serializes operations on the futex queue, the ConcAcl approach allows parallelism. This is critical for highly concurrent event-driven applications where parallel wake operations targeting the same futex address are likely.

3.2.2 ConcAcl Agent Managed Scheduling

Overbearing context-switching penalties are the main driving force which motivate the use of user-threading in highly-concurrent applications. Reducing this penalty to an acceptable level can restore the viability of OS threads. This is a desirable outcome as implementing threads outside the OS threads is sub-optimal (see Section 2.1.1). In Linux, the “completely fair scheduler” (CFS) is the default scheduler and implements a policy of fairly dividing CPU time across tasks. The complications in OS scheduling stem from dealing with distributed
run-queues. Implementing load-balancing, task-stealing, and task-offloading in a performant manner is challenging due to the coordination intensive nature of these functions. With ConcAcl coordination is facilitated and these functions are simplified. Thus, the OS can leverage ConcAcl by delegating the responsibility of thread scheduling to the ConcAcl context. With this, the context-switch penalties are effectively eliminated and ConcAcl can rely on invoke to simplify coordinating the difficult load-balancing, task-stealing, and task-offloading functions.

Agent Managed Thread Selection (AMTS) allows the OS to delegate thread scheduling to ConcAcl. Thus, the preemption handler will be serviced by the ConcAcl context. This context will interact with three task queues to maximize resource efficiency and minimize QoS violations. These queues are the IDLE_QUEUE, DEADLINE_QUEUE, and the FAIR_QUEUE. The DEADLINE_QUEUE is prioritized and is intended to hold microsecond scale tasks whose deadline is less than the preemption period of the OS tasks. Figure 3.5 presents the AMTS scheduler code. The “next_queue” and “next_task” functions (lines 1-2) are helper tasks to set the current queue and next task. The next_queue procedure creates a prioritization of deadline tasks by returning the deadline queue whenever it contains task.

To match current system context-switching behavior, we implement three mechanisms which can trigger AMTS. These mechanisms are timer-preemption, sys_yield, and sys_futex (wait). The simplest mechanisms to trigger AMTS are the expiration of a scheduling quanta or the execution of a yield operation. These operations result in task selection followed by a context switch. To implement preemptive scheduling we assume the availability of special timer interrupt which is used to trigger the ConcAcl AMTS context. We assume the availability of back-door instructions (smt_stop, smt_cont, smt_move, and smt_load) which can allow ConcAcl to manage the state of SMT contexts.

The OS supervises the process in two ways. First, timer-interrupts continue to invoke traditional OS execution by interrupting one of the SMT contexts and switching to the OS interrupt handler. Second, a scheduling exception is raised if — when picking the next queue — the IDLE_QUEUE is selected and it is found to be empty. This will be a queue for the OS to either swap in a new batch of tasks — possibly by retrieving them from another machine — or to power down the CPU to save power. To match OS behavior, the resched_cpu operation which concludes a futex_wait operation will trigger AMTS. With this, futex_wait and futex_wake operations can both benefit from AMTS.

Idle Queue: The idle runqueue is replicated across the multicore using ConcAcls shared state. Thus, to pick the next task from the idle queue a core will claim it and then execute a pop_idle using invoke. Tasks are initially scheduling according to the fair timeslicing policy. To switch to idle scheduling, a task will execute an amts_idle operation. The arguments will specify the periodicity of the tasks. The common use-case of these tasks is maintenance.
```c
// Return next-queue.
TaskQueue next_queue();

// Select next-task from queue.
TaskStruct next_task(TaskQueue Q);

void amts(tid,op,arg1,arg2) {
  switch(op) {
  case TIMER:
    if (CHECK_TASK_STEAL) {
      ConcAcl_drain();
      if (Next_Queue() == IDLE_QUEUE) {
        ConcAcl_invoke(take_fair_load,CID);
        Default_Preemption_Timer();
        CHECK_TASK_STEAL = false;
      return;
    }
  case YIELD:
  case WAIT:
    break;
  case IDLE:
    idle_task = mk_idle_task(task(TID),arg1,arg2)
    ConcAcl_invoke(add_idle_task, idle_task);
  case DEADLINE:
    total_delay=sum(PRI_QUEUE);
    deadline=arg1;
    task_len=arg2;
    if (deadline < TOTAL_DELAY + task_len) {
      remove(queue(curr_task),curr_task);
      enqueue(curr_task,PRI_QUEUE);
    } else
      invoke(push_task_pri,task(tid));
  }
  next_queue = next_queue();
  next_task = next_task(curr_queue);
  if(next_task != None) {
    smt_ctx_swap(tid,curr_task,next_task);
    if(next_queue == IDLE_QUEUE) {
      ConcAcl_invoke(take_dl_load,CID);
      CHECK_TASK_STEAL=True;
      Short_Preemption_Timer();
      return;
    }
  }
  Default_Preemption_Timer();
}
```

Figure 3.5: AMTS: Algorithm to allow OS to delegate thread scheduling to ConcAcl context.
System daemons typically use this policy to limit the amount of resources consumed.

Deadline Queue: To switch to a latency-sensitive scheduling policy an application can execute a `amts_deadline` operation. This operation takes as operands a deadline (in microseconds) along with a task length (also in microseconds). From there, the implementation will examine the total delay in the prioritized runqueue and offload the thread to another is invoked to push the task to another CPU where it has a better chance of meeting the deadline (line 31). With this, ConcAcl can mitigate microsecond scale QoS violations efficiently by exploiting the fine-grained coordination capabilities of ConcAcl.

Scheduling Optimization: There are three ways that scheduling is optimized to improve performance and resource-utilization. First — when local queues are empty — AMTS will attempt to steal tasks from remote queues (lines 11,38). AMTS will first attempt to steal tasks from remote deadline queues by using `concall_invoke` to execute `take_dl_load(CID)` on all remote AMTS agents. After the invoke, AMTS will resume the idle task but will set a short preemption timer to callback into AMTS and check the results of the task stealing operation (lines 21-29). During this call-back, AMTS will synchronize with the pending ConcAcl functions using `concall_drain`. If no tasks were deposited next_queue will return the `IDLE_QUEUE` again. In this case, another task stealing attempt is launched (line 11) — this time for fair tasks (take_fair_load) — and ConcAcl will leave the idle task to complete. While the two task-stealing operations can be merged, it remains helpful to prioritize deadline tasks avoid taking fair tasks when prioritized work is available.

### 3.3 Evaluation

To evaluate ConcAcl we emulate it using software instructions on dedicated compute resources. We isolate CPUs from kernel scheduling and IRQ processing and dedicate LLC and bandwidth resources using Intel performance isolation features. We leverage Processor Trace (PT) to record the execution of benchmarks without introducing significant overheads. Processor Trace can be configured to capture timing information along with branch traces. To project performance benefits we post-process traces. During the processing, we can use the timing information which PT stores in the log to estimate the performance cost of the functions traced. This enables accurate estimation of performance gains possible when accelerating the functions of interest. To emulate ConcAcl we reserve one thread per core and isolate them from interference by preventing the kernel from scheduling tasks to them. These threads emulate ConcAcl OS threads and respond to requests for scheduling operations, futex operations, and TLB operations.
3.3.1 ConcAcl Model

Each core has an additional SMT context dedicated to the ConcAcl. In addition, 8KB of on-chip memory is added to each core. Half is private to each core while half is replicated and shared across all cores. ConcAcl contexts pass messages to each other by updating the shared state. A private, broadcast based interconnection network delivers these messages. This network has a throughput of $N \times 10^7$-msgs/s to enable a broadcast latency of one tenth of a microsecond. With this, the cost to invoke a function across all cores is three-tenths of a microsecond.

3.3.2 Benchmark Implementations

Strongly Atomic Software Transactional Memory: Transactional memory is used to manage concurrent access to shared data-structures in a safe manner. Accesses and modifications of transactional data-structures are guaranteed atomic by the TM implementation. To achieve this the implementation must detect and resolve read-after-write, write-after-read, and write-after-write conflicts. The TL2 library implements TM by constructing read and write sets to detect conflicts and by logging changes to data-structures until transactions are committed. A known deficiency of this approach is that it can only guarantee atomicity across transactions. This transactional guarantee is weak and can lead to bugs.

Strong transactional memory protects the atomicity of transactions against non-transactional accesses. This approach is more difficult and more costly to enforce. To provide eager transactional memory the OS provided copy-on-write can be used. During transactions shared pages are marked copy-on-write and writes will trigger an expensive copy. This copy enables roll-back when conflicts force a transaction abort. It is also possible to implement lazy transactional memory by utilizing page-permissions. Instructions which access memory in a manner disallowed by page permissions will raise a page-fault to the OS. Thus, limiting access to pages accessed during transactions enables the detection of conflicts. Pages modified by a transaction are given invalid page-permissions on all other processors. Read pages are given read-only permissions. To safely modify the permissions the OS must coordinate with all other processors to ensure the new permissions are being respected by all processors. This coordination process is needed to execute a TLB-shootdown, thereby ensuring that no TLBs contain stale translation. On modern processors, TLB shootdowns are conducted by interrupting all cores to execute TLB invalidation instructions.

We modify TL2 [143] and remove the cache-line locking as it is no longer necessary. We retain the redo/undo logging mechanisms to allow utilizing the available lazy and eager update strategies. Non-transactional code is prioritized and always aborts transactions. Conflict
resolution is based on processor ID and transaction age.

Event-Driven Concurrent Web-Server: Web-servers continuously face blocking conditions due to transmission delays, connection churn, and intermittent usage. Asynchronous execution patterns can avoid blocking and are critical for performance and efficiency. These techniques rely on event-synchronization [144] instead of polling to detect new requests and trigger their processing. This enables concurrency between fulfilling requests and waiting for their arrival. Thus, web-servers can scale to hundreds of thousands of intermittent connections being serviced by a single thread. This pattern does not preclude parallelism. With thread pools web-servers can process requests in asynchronous stages [145,146] to minimize delay.

Apache web-server, for example, utilizes this approach. Threads monitor connections using epoll/kqueue and on request wake workers using futex operations. The client-server orientation of these applications make this approach to concurrency a natural one. Each client is tracked by a unique thread while resources for request processing are shared across clients. We implement a two-stage web-server using this approach. The first stage consists of a thread which monitors a unique buffer containing packets delivered by NICs using receiver-side scaling. Packets are pre-processed to identify which connection thread needs to be awoken to handle the request. This per-connection thread uses futex operations to suspend and resume its execution in an event-driven fashion. Connection threads inspect pending packets to determine which services must be invoked to fulfill the client request. Requests to these services will trigger the second stage which consists of service-providing threads which process client requests in batches. Maximizing performance requires balancing resources across these various stages.

3.4 Discussion

3.4.1 Microbenchmarks

TLB Shootdown: Figure 3.6 breaks down the cycle cost of TLB shutdown for each of the stages. Four bars present four configurations; the bottom two use ConcAcl. Each bar is broken down into the costs of the TLB shutdown discussed in Section 2.1. We further break down the costs of dispatch into dispatch (p) and dispatch (a) - or prep and address respectively. The major cost stems from the serial and linear scaling of the “prep” and “notify” steps. Linear scaling stems from preparing per-target structures and doing per-target bitmask operations to compute the set of remote processors. We assume a latency to trigger the coordinated step across agents equivalent to the IPI based signaling latencies we observed.
ConcAcl powered TLB shootdowns can provide a 64 core shootdown at a cost cheaper than the legacy 2 core shootdown.

To analyze the impact on system calls we instrumented the OS and libraries to trace the system calls and TLB shootdowns. We use sysfs interfaces to collect results and buffer everything in memory. With these traces, we can post-process them to re-evaluate the performance of a ConcAcl powered TLB shootdown. Figure 3.7 graphs the percentage of the ideal peak-throughput of mprotect system-calls as more cores are involved. The ideal peak throughput is computed by multiplying the throughput of a single core by the number of cores involved. When an mprotect system call is made a TLB shootdown is required, adding overheads and reducing the mprotect throughput. With the legacy TLB shootdown (dashed line) the throughput degrades to 50% of the expected peak once 16 cores incur the shootdown coordination overheads. With ConcAcl shootdowns (solid line) the throughput remains steady.

Producer-Consumer Delay: Figure 3.8a shows the impact of AMTS on producer-consumer scheduling delay. This figure plots the tail-latency (microseconds, log-scale) versus the number of concurrent tasks. Polling (i.e. spinning) is the mechanism through which producer-consumer delay is minimized. In this approach the consumer will poll for producer completion. As expected, with a concurrency that exceeds the available CPU count the tail latency growth begins to accelerate towards infinity. Event-synchronization can notify the consumer when the producer is complete –0 thereby preventing this jerk in tail-latency. Still, once the
concurrency exceeds the CPU count, contention in the OS on futex-related spin-locks will impose latency penalties. The more tasks, the more likely to block on spin locks. With ConcAcl based futex operations these spin locks are eliminated. This allows smooth growth in tail-latency as the machines over-subscription increases.

### 3.4.2 Benchmarks

**AMTS Event-Driven Web-Server:** To measure the performance benefits of AMTS we implemented an event-driven web-server. We compare the performance of the server tracks connections using OS threads, coroutines, and ConcAcl threads. Simply cycling between all coroutines repeatedly can maximize the performance — at the cost of large CPU utilization. At 30,000 requests/s this would waste a large amount of CPU time which can go to other workloads. To reclaim these cycles, an event-driven architecture is used. A few threads will poll packet-buffers and attempt to wake threads as requests arrive. Implementing this design using sys_futex and standard OS threads cannot satisfy the request arrival rates because of the cost of sys_futex operations and the cost kernel-driven context switches. ConcAcl solves both of these problems. With ConcAcl threads the performance of the web-server is within 10% while using a minimal number of cycles for the server.

Beyond 10,000 tasks the performance drops off significantly due to resource limitations. We save the register state in a contiguous memory buffer, which allows storing the index to the thread in the on-chip memory. Thus, if the private on-chip ConcAcl state was used
(a) Producer-consumer delay vs. concurrency in event-driven applications.

(b) Throughput of requests which meet QoS requirements.

(c) Web-server performance: AMTS vs OS vs. coroutines.

Figure 3.8: Performance benefits of ConcAcl AMTS on QoS-sensitive, event-driven web-server.
exclusively for holding waiter-structs, then roughly 630 threads can be suspended at each core and managed without OS involvement. For these reasons ConCAct closely tracks coroutines with performance dropping as OS involvement becomes frequent due to futex misses in the broadcast memory.

AMTS QoS + Batch Jobs: Intermittent client-server workloads create an opportunity for co-scheduling background tasks. These can be batch tasks like machine workloads. Unfortunately, for latency-sensitive servers this is not an option on modern OS due to the inability of the OS to react fast enough if a burst of requests were to arrive. We configure the load of the same event-driven server to exhibit a larger degree of intermittence to evaluate whether AMTS can enable fine-grained sharing of cores between latency-critical tasks and idle-only batch jobs. We generate a load of variable processing latencies in a manner which guarantees that at least 80% of the system cycles are always free for batch tasks. Requests are dispatched to random workers that would require much less than a single scheduling quanta to process the maximum number of requests. We also load the server with 80 CPU intensive batch jobs per CPU. We pin the workers and allow the batch jobs to migrate.

Figure 3.8b visualizes the throughput which meets aggressive QoS requirements of 1.2x the processing latency for requests. The figure visualizes the min, max, first quartile, third quartile, and the average number of requests processed which meet QoS requirements. The results are derived from 100 trials with each trial consisting of launching the server and workload generator and collecting 1000 sample request latencies.

OS scheduling performs the worst. OS threads are unable to preempt batch-tasks fast enough to avoid QoS violations. Workers poll for 500 microseconds before suspending on a futex. Any requests which arrive after the suspension will require waiting until the next preemption. The results eventually stagnate in part due the fact that requests are distributed to random workers. With a worker count equivalent to half the available cores, the chances of overloading a single worker create a rapid increase in QoS violation rates. In addition, the costs of migrating batch-tasks displaced by the higher-priority workers began to offset the potential gains of adding workers. This results in a web-server whose performance (under QoS consideration) stagnates at between 16% and 33% of the ideal throughput.

With AMTS, the low costs of context-switching, load-balancing, and task-migration help improve performance. Throughput and performance predictability are both improved as evident by the tighter candle-sticks. Ideal performance — linear scaling with worker count — is still not realized for the same reasons as OS threads. Again, without the ability to avoid QoS violations, performance tapers off once the worker count is equivalent to half the cores. By utilizing the AMTS deadline operation, AMTS can both detect and anticipate QoS violations and take measures to avoid them by using invoke. This results in the ideal
performance and allows batch tasks to claim over 90% of available cycles without significantly degrading the server’s QoS.

![Graphs showing overhead sources for strongly atomic STM](image)

**Figure 3.9:** Overhead sources for strongly atomic STM.

**Strong Software Transactional Memory:** Page-permission based STM implementations require TLB shotdowns to prevent stale TLBs. We select the genome, labyrinth, and vacation workloads from the STAMP [147] benchmark. These particular workloads spend a majority of their time executing transaction instructions. Figure 3.9 details the sources of slowdowns when operating TL2 in strong-atomicity mode. These costs come from the costs of committing and aborting transactions, dispatching load and store related shotdowns, and resolving page-faults from loads and stores. These component costs are normalized to the cost of the benchmark instructions themselves.

Page-fault costs are insignificant as transaction lengths are often small enough to only require on or two page-faults. The exception is labyrinth whose transactions are much larger
containing roughly 14x and 8x more memory accesses than genome and vacation. On 32 core runs, page-faults from loads add a 62% overhead while page-faults from stores add a 420% overhead. At 8 cores and 16 cores the total TLB shutdown respectively add overheads of 195% and 341% for genome, 731% and 992% for labyrinth, and 200% and 300% for vacation. TLB shootdowns for stores create the majority of this overhead, adding 2060% and 2500% overheads for labyrinth and vacation for 32 core runs. Writes dominate because TLB shootdowns are only needed to upgrade permissions. Since pages are initially shared, TLB shootdowns are rarely needed for load instructions — only for prioritized transactions which need to read from a page owned by some younger transaction.

These numbers come from a lazy-configuration of the TL2 library. Technically, this makes transaction commit more expensive than transaction abort despite the fact that transaction abort imposes much larger overheads. This stems from the fact that when TL2 is operated at the granularity of pages false-sharing becomes common. This can be eliminated by forcing memory allocators to return memory on page-boundaries. We opted to avoid this optimization.

Figure 3.9f shows the speedup gained by using ConcAcl for TLB shootdowns in this workload. Global TLB shootdowns on ConcAcl are a fraction of the cost of IPI based shootdowns. This eliminates the major cost of this workload resulting in 7-9x speedup for 64 core runs. It is worth noting that the performance of page-permission based transactional memory is much more expensive than the lock-based approach of TL2. These costs are shown for IPI and ConcAcl mechanisms in Figures 3.9b and 3.9d. To further eliminate these costs we need user-space mechanisms for changing page-permissions.
Chapter 4

RNR-SAFE: EXPLOITING CONCURRENT RECORD AND REPLAY FOR SECURITY

4.1 An RnR Security Framework

Figure 4.1 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 nondeterministic events of the execution in a software log. Recording adds only modest overhead — less than 15\% on average, according to Pokam et al. [29]. Note that we record at the VM level to also protect the operating system.

![Diagram of RnR-Safe organization]

The designer has augmented the hardware in the recorded VM (e.g., processor and memory system) with support to detect certain classes 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, a checkpointing replayer VM re-executes the workload natively. It uses the log to inject all the non-deterministic events. As a result, the execution deterministically follows the original one. If an alarm is found in the log, an alarm replayer VM characterizes the alarm, detecting either an attack or a false positive.
4.1.1 RnR-Safe Modes of Execution

In RnR-Safe, monitored recording consists of normal execution. Transparently to the recorded VM, all the nondeterministic inputs are being recorded by the hypervisor in a log, and the hardware and the hypervisor monitor 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 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 out of a recent (typically the latest) checkpoint. Old checkpoints and log entries are regularly 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.

4.1.2 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 from imprecise security hardware. The one requirement of the alarm mechanism is to avoid any false negatives.

Flexibility. RnR-Safe is flexible. As attackers devise new attacks, defenders can augment the recorded VM and its hypervisor 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. If desired, RnR-Safe also 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.
4.1.3 How to Use RnR-Safe

The RnR-Safe framework can be tailored to detect different attacks. For each attack, we first need a first-line of detection that targets that threat in the machine. The key advantage of RnR-Safe is that such a detection technique, 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 unclear which types of attacks would be best suited for RnR-Safe to target, Table 4.1 outlines three example attacks: ROP, jump-oriented programming (JOP) [148] and denial of service (DOS) [149]. For each, the table outlines the alarm trigger, possible first detection technique, and the role of replay. The first entry is ROP, which will be discussed in detail in the rest of this paper. The alarm trigger is a RAS misprediction. As we will see, the first detection technique is based on managing a multithreaded RAS, and using a whitelist of acceptable RAS mispredictions. The role of replay is to model a kernel-compatible shadow stack.

Table 4.1: Examples of potential RnR-Safe uses.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Alarm Trigger</th>
<th>Possible First Detection Technique</th>
<th>Role of Replay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP (this chapter)</td>
<td>RAS mis-prediction</td>
<td>Manage a multi-threaded RAS, use a whitelist</td>
<td>Execute a kernel-compatible shadow stack</td>
</tr>
<tr>
<td>Jump Oriented Programming (JOP)</td>
<td>Stray indirect branch or call</td>
<td>Table of begin and end addresses of the most common functions</td>
<td>Verify if the target is one of the less common functions</td>
</tr>
<tr>
<td>Denial of Service (DOS)</td>
<td>Kernel scheduler inactivity</td>
<td>Counter of number of context switches</td>
<td>Identify reason for low switching frequency</td>
</tr>
</tbody>
</table>

JOP is a related class of attacks, mounted by redirecting indirect branches or call instructions to execute the victim’s code. Preventing such attack requires preserving the CFI of call and branch instructions. A first detection technique against JOP can be a table of begin and end addresses of the most common functions. An indirect branch or call target is compared to the table and is legal if the target is the first instruction of a function. Indirect branch targets within the current function are also fine. Otherwise, an alarm is triggered, and the replay verifies the same conditions for 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 the replay analyzes the code that has dominated the system’s execution time.

47
4.2 Example: Targeting Kernel ROPs

4.2.1 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, and is not accessible by software; not even the kernel. 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 mispredictions, making false negatives impossible. Furthermore, for the RAS to be useful in RnR-Safe, false positive alarms should be infrequent. In practice, there are a few major sources of false positives in the basic RAS operation. We explain these sources with Linux kernel examples, when relevant.

First, there is the effect of multithreading. In a multithreaded environment, on a context switch from Thread i to Thread j, the RAS may still retain some entries belonging to Thread i. 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 when i executes again. Hence, Thread i will also mispredict.

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 a matching address, the RAS will not contain the 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.

Finally, 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 events occur relatively more frequently—specifically, in the setjmp/longjmp calls, typically used in exception handling.

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. Hence, RnR-Safe adds support for the first two sources of false positives: multithreading and non-procedural returns in the kernel. However, to completely eliminate false positives would require major hardware and software changes. Hence, RnR-Safe raises alarms for the last two sources of false positives (underflow and imperfect nesting), and relies on the replayer to flag the events as false positives, and discard them. We discuss each of these cases in the next few sections.

4.2.2 Basic Design

As shown in Figure 4.1, the workload (applications plus kernel) runs in a recorded VM. As the workload runs, the hypervisor creates an input log that is sent to and is consumed by the checkpointing replayer VM. If the workload executes a return instruction and the hardware finds a mismatch 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 an alarm replayer, starting from the most recent checkpoint. The alarm replayer determines whether the alarm is a false alarm or a real ROP.

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

4.2.3 Supporting a Multithreaded Environment

In a multithreaded environment, as a thread is descheduled, it may leave return addresses in the RAS. Such addresses may be popped by subsequent threads, which will suffer mispredictions. Further, when the original thread is rescheduled, it may pop RAS entries belonging to other threads, which will also cause mispredictions. The result is false ROP alarms.

To address this problem, RnR-Safe augments the microcoded virtualization hardware so that it additionally performs the following operations on a context switch. First, it saves the current RAS into a safe memory area outside of the reach of the kernel. Then, it restores the RAS state as needed for the upcoming running thread. To know the correct memory area to move data to and from, the hardware uses a new hardware pointer that the hypervisor sets.

The structures are shown in Figure 4.2. The software structure in memory is an array of backed-up RASes (BackRAS array). Each entry in the array 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 reloaded 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.

![Diagram of Memory Structure and Processor Structures](image)

Figure 4.2: Structures used to support multiple threads.

Figure 4.3 shows the logic used. On a context switch, as part of the transition to the hypervisor, the hardware saves the RAS to the BackRAS 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 BackRAS entry for the new thread. Finally, as part of the transition back to the guest, the microcoded hardware loads the correct BackRAS entry into the RAS, taking another 200 cycles.

![Algorithm and timeline](image)

Figure 4.3: Algorithm and timeline to handle multiple threads.

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 4.3.2 explains how this is done without modifying the guest kernel.

With this support, when a thread is scheduled, it finds its correct state in the RAS, thus eliminating many false alarms.
4.2.4 Supporting Non-Procedural Returns

Sometimes, the kernel 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 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, depending 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. There is a single-entry return whitelist (RetWhitelist) with the PC of the instruction with the non-procedural return, 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 PC 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 generated, 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 4.3.1.

4.2.5 RAS Underflows and Imperfect Nesting

For false positives that occur very infrequently, RnR-Safe’s approach is to avoid complicated hardware in the recorded VM and, instead, raise alarms and rely on the replayer VM to handle them. RnR-Safe uses this approach for the remaining two cases: RAS underflows and imperfect nesting.

The first case occurs when the kernel or an application executes enough nested procedure calls to cause the RAS to evict an earlier return address. Later, when the hardware tries to pop that entry from the RAS in a return instruction, it will find the RAS empty. This is a RAS underflow, and is flagged as a RAS misprediction.
RnR-Safe handles this case as follows. When a RAS entry is about to be evicted in the recorded VM, the hardware triggers a VM exit and dumps the about-to-be-evicted RAS entry to a special memory that is out of the kernel’s reach. Then, the hypervisor places this data as an Evict record in the log. Later, when RAS underflow causes a RAS misprediction, the hardware will raise an alarm and the hypervisor will place an alarm record in the log.

The second case is imperfect procedure nesting, such as that caused by setjump and longjump. Imperfect procedure nests occur rarely, but require very complicated hardware to handle transparently [91]. Consequently, when an imperfect procedure nest causes a RAS misprediction, RnR-Safe simply places an alarm record the log.

The replayer has enough information to handle both cases. The replayer will be able to match evict records with alarm records due to RAS underflow. Further, the replayer will be able to identify setjumps and longjumps easily and fix its software RAS. Overall, while these false positive alarms will slow down the workload execution when they happen, they are rare enough to have a negligible total performance impact. The proven effectiveness of the RAS to predict return targets suggests that the RAS will soon re-synchronize with the program’s true return addresses.

4.2.6 Replaying Platform

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

Checkpointing Replayer

To understand the operation of the checkpointing replayer (CR), we first describe the contents of a checkpoint. Figure 4.4 shows three checkpoints. Each checkpoint has three components. The first one is all the VM state. It includes all the memory pages of the VM, a page with the processor state at the time of checkpoint (PC, stack pointer, and the rest of the registers), and the virtual disk image contents. The latter is the state that the VM has written 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 and blocks that have been modified since the previous checkpoint; for each unmodified page or block, it keeps a pointer to it in the latest checkpoint that modified it.

The second component of a checkpoint is a pointer to the input log buffer (InputLogPtr). The pointer points to the next input log record to be processed after the checkpoint. Finally, the third component is the BackRAS at the time of the checkpoint. We will see in Section 4.2.6
why it is needed.

With this background, we can describe the CR operation. The CR re-executes the workload deterministically. When the time since the last checkpoint becomes higher than a certain threshold and a VM exit occurs, a checkpoint is taken. In a checkpoint, first, the hardware automatically saves the RAS into the BackRAS. Then, the CR dumps the processor state (PC, stack pointer, and all registers) into a memory page. Then, the CR creates the checkpoint by saving: (1) the page with the processor state, and all the memory pages and disk blocks modified since the prior checkpoint (together with pointers to the unchanged ones), (2) the current BackRAS, and (3) the current InputLogPtr. Then, the CR marks all pages copy-on-write, and resumes execution. During execution, when a page is written for the first time since the last checkpoint, a copy is made and used from then 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.

The replay takes place in a safe platform, where the hardware’s ability to trigger ROP alarms is disabled. This is because replay does not create alarms. However, the hardware still dumps the RAS at context switching points. This ensures that, at the point of a checkpoint, the CR can reconstruct the up-to-date state of the complete BackRAS to stash in the checkpoint.

Alarm Replayer

When the CR encounters an alarm, it launches an alarm replayer (AR) VM starting from the checkpoint immediately preceding the alarm. The AR will determine whether the alarm is caused by a ROP or is a false alarm. The AR starts by initializing the VM state using the checkpoint. It marks all the pages and blocks in the checkpoint as copy-on-write to avoid modifying the initial state. Then, it reads the checkpoint’s BackRAS into a software data
structure that it uses to simulate the RAS. Next, it loads the saved processor state from memory into the processor registers. Finally, it starts execution, reading from the log starting from the checkpointed InputLogPtr.

The AR executes the original workload natively, in a deterministic manner, consuming the input log until it reaches the alarm marker. The AR traps at every call and return instruction, inducing VM exits and transferring control to the hypervisor. There, an unbounded RAS is modeled in software, with our extensions for multithreading and non-procedural returns. 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.

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, the AR provides the state of the system at the point of the ROP attack. An expert programmer can study the state to glean information about the attack. Moreover, the AR can be re-run multiple times, with increasing levels of instrumentation, or starting at different checkpoints, to fully characterize the attack.

The case of evict records in the input log and the resulting underflow alarm records in the input log is handled in a special manner. While it can be handled by ARs, it is simpler if the CR handles this special case itself. Specifically, as the CR processes the input log and finds evict records, it saves them in a software structure. When the CR finds an underflow alarm record in the input log, it compares it against the latest evict record from the same thread. If they match, the alarm was a false one, and the CR removes the corresponding evict record from its structure. Otherwise, the CR launches an AR to handle this alarm.

Section 4.4 shows an attack. Note that our design allows running multiple ARs concurrently, to analyze the same or different ROP alarms in parallel.

### 4.3 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.

#### 4.3.1 Hardware Support

The hardware support required by RnR-Safe includes a replay platform, small extensions to the RAS hardware, the BackRASptr register, and the two whitelist tables. The requirement
for RnR can be considered the most substantial addition. However, RnR is well understood and accepted as a useful primitive for debugging and program analysis. Also, the RnR infrastructure can be reused for a large variety of debugging and security analyses. The RAS extensions consist of triggering an exception when the RAS is about to evict an entry, and microcode to dump into memory and restore the BackRAS array, and to dump into memory the evicted RAS entries. There is also microcode to maintain the BackRASptr and whitelist tables.

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.

4.3.2 Hypervisor Support

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 pointing 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 transferred to the hypervisor, the latter can introspect the state of the guest kernel 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.
Recycling BackRAS Entries

In Linux, threads can be 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. 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. A similar approach is followed when a thread is created and its BackRAS entry needs to be allocated.

4.4 Mounting A Kernel ROP Attack

We built and mounted the ROP attack of Figure 2.3. In the recorded VM, as the workload calls the Vulnerable procedure of Figure 2.3c, the hardware pushes into the RAS the return address at the call site (call it Return Address). This is the same address that is stored above the buffer in the software stack of Figure 2.3e. After the malicious string copy, the software stack becomes Figure 2.3f. As the program executes the return of the vulnerable procedure, the hardware uses the RAS to predict that execution will transfer to return address. In reality, the target of the return is resolved to be the address of gadget G1, as shown in Figure 2.3f. 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 can analyze 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 launch additional alarm replayers 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 simulated 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 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 [150]. This information is easy to get now since the workload is not running.

4.5 Evaluation

4.5.1 Experimental Setup

Goal of the Evaluation

In this chapter, 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 mount only the kernel ROP attack that we describe in Section 4.4. Such attack is representative of ROP attacks, as they all use similar gadget-based patterns. Collecting and analyzing multiple real-world kernel ROP attacks is left as future work.

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 [151], 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 4.3 shows the system configuration we use for our performance evaluation, and Table 4.2 shows our benchmarks. The benchmarks are: fileio and mysql from SysBench [152], apache, make (a compilation of the Linux kernel), and radiosity from SPLASH-2 [153].
Table 4.2: Benchmarks executed.

<table>
<thead>
<tr>
<th>Host machine</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Xeon E3-64bit,4-cores,3.1GHz</td>
<td>Memory: 8 Gbytes</td>
</tr>
<tr>
<td>OS: Ubuntu, Linux kernel 2.6.38-rc8</td>
<td></td>
</tr>
<tr>
<td>Guest machine</td>
<td></td>
</tr>
<tr>
<td>CPU: uniprocessor</td>
<td>Memory: 1 Gbyte</td>
</tr>
<tr>
<td>OS: Debian, Linux kernel 3.19.0</td>
<td>Disk: 32 Gbytes</td>
</tr>
</tbody>
</table>

Table 4.3: System configuration for performance evaluation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>-n100000 -e20</td>
</tr>
<tr>
<td>fileio</td>
<td>-file-total-size=6G -file-test-mode=rndrw</td>
</tr>
<tr>
<td></td>
<td>-file-extra-flags=direct -max-requests=10000</td>
</tr>
<tr>
<td>make</td>
<td>linux-4.0 config with all-no</td>
</tr>
<tr>
<td>mysql</td>
<td>-test=oltp -oltp-test-mode=simple</td>
</tr>
<tr>
<td></td>
<td>-max-requests=500000 -table-size=4000000</td>
</tr>
<tr>
<td>radiosity</td>
<td>-p1 -bf 0.005 -batch -largeroom</td>
</tr>
</tbody>
</table>

Handling Non-Deterministic Events

Synchronous Non-Deterministic Events: Instructions such as rdtsc (read time stamp counter) or rdrand (read random number generator) return non-deterministic results. Accesses to memory regions like Memory Mapped IO (MMIO) are also non-deterministic. The VMCS controls when the processor will perform a VMExit. During recording, we configure the controls to synchronously trap these non-deterministic 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 Non-Deterministic Events. Asynchronous events are more challenging to replay. They 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.
Figure 4.5: Execution time of recording setups (a) and breakdown of the Rec overhead over NoRec (b).

Each step will suffer the overhead of a VMExit ($\approx 1,000$ cycles).

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.

4.5.2 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.
4.5.3 Discussion

A breakdown of user/kernel instruction counts is shown for the evaluation workloads in Table 4.4. Table 4.5 presents data on the number of points requiring instrumentation and hypervisor involvement during replay while Table 4.6 shows the alarm performance during recording.

Table 4.4: Workload characteristics (dynamic counts).

<table>
<thead>
<tr>
<th>Workload</th>
<th>Total Instrs</th>
<th>Kernel Instrs</th>
<th>Ratio (K/T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>2.62E+10</td>
<td>1.78E+10</td>
<td>0.68</td>
</tr>
<tr>
<td>fileio</td>
<td>1.63E+09</td>
<td>1.49E+09</td>
<td>0.92</td>
</tr>
<tr>
<td>make</td>
<td>2.72E+11</td>
<td>2.46E+10</td>
<td>0.09</td>
</tr>
<tr>
<td>mysql</td>
<td>1.45E+11</td>
<td>1.35E+10</td>
<td>0.09</td>
</tr>
<tr>
<td>radiosity</td>
<td>9.06E+10</td>
<td>4.05E+09</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 4.5: Replay+RIIV sources of overhead (per 100,000 instructions).

<table>
<thead>
<tr>
<th>Workload</th>
<th>Context Switches</th>
<th>Calls</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>5.26</td>
<td>2457.10</td>
<td>2462.34</td>
</tr>
<tr>
<td>fileio</td>
<td>8.02</td>
<td>2566.22</td>
<td>2574.11</td>
</tr>
<tr>
<td>make</td>
<td>3.08</td>
<td>2350.24</td>
<td>2353.23</td>
</tr>
<tr>
<td>mysql</td>
<td>9.59</td>
<td>2394.68</td>
<td>2404.25</td>
</tr>
<tr>
<td>radiosity</td>
<td>1.11</td>
<td>2486.74</td>
<td>2487.85</td>
</tr>
</tbody>
</table>

Table 4.6: Recording alarms suppressed (per 100,000 instructions).

<table>
<thead>
<tr>
<th>Workload</th>
<th>whitelisted</th>
<th>RBM Suppressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>3.04</td>
<td>13.76</td>
</tr>
<tr>
<td>fileio</td>
<td>6.79</td>
<td>29.09</td>
</tr>
<tr>
<td>make</td>
<td>0.26</td>
<td>0.73</td>
</tr>
<tr>
<td>mysql</td>
<td>1.41</td>
<td>5.29</td>
</tr>
<tr>
<td>radiosity</td>
<td>0.04</td>
<td>0.15</td>
</tr>
</tbody>
</table>

4.5.4 Recording

Our recording scheme generates the log and also saves and restores the RAS at context switches. Recall that we require hypervisor mediated I/O, which prevents the use of paravirtualized (PV) network drivers. Figure 4.5a 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 saving and restoring the RAS at context switches (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 successfully applied to PV drivers [88]; 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 and restoring the RAS (RecNoRAS) takes 24% longer than NoRec. These overheads are modest, and will decrease in a reasonably optimized implementation of recording — e.g., Pokam et al. [29] measure that their implementation of recording adds only 13% overhead.

To understand the source of overheads, Figure 4.5b 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 and restoring the RAS at context switches.

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 and restoring the RAS induces only 4% overhead on average.

Figures 4.6a and 4.6b 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 of recording network packet contents. Also, the bandwidth to save and restore the RAS at context switches is very small. Overall, the impact of the architecture on the memory system is modest.

4.5.5 Minimizing False Alarms in the Kernel

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 4.8 shows the number of kernel false alarms, broken down into those suppressed with the whitelist and with the BackRAS, and those reported to the replayers (FalseAlarm). The figure shows the number per million instructions.
Figure 4.6: Input log generation rate (a) and bandwidth to save and restore the RAS at context switches (b).

Figure 4.7: Execution time of checkpointing replay setups (a) and breakdown of the RepChk1 overhead over Rec (b).
Since the number of false alarms passed to the replayers is so small, the FalseAlarm category cannot be seen, and we put the number on top of the bars. All the benchmarks except Apache pass practically no kernel false alarm. Apache passes a few false alarms, which are RAS underflows. They are caused by the deep procedure nesting of the network driver code under extreme loads. Both the whitelist and the BackRAS are very effective at supressing false alarms.

![Graph showing false alarms](image)

Figure 4.8: Kernel false alarms suppressed (Whitelist and BackRAS) and reported to the replayers (FalseAlarm).

4.5.6 Replaying

Checkpointing Replay

Figure 4.7a 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 easily 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 back pressure to temporarily slow down recorded execution.

The figure also shows that increasing or decreasing the checkpoint period changes the replay speed. However, even without checkpointing, replay already takes on average 48% longer that Rec.
To understand these effects, Figure 4.7b takes the slowdown of RepChk1 over Rec and breaks it down into its sources. The sources are the events that we saw in Figure 4.5b for the recording, plus creating checkpoints (Chk).

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 4.5.1, it requires single-stepping VMExits over several instructions. This is the reason for the overhead of Figure 4.7b. It also explains that replaying without checkpointing (RepNoChk) already has significant overhead over Rec.

Alarm Replay

We now consider the speed of an alarm replayer that checks for ROP attacks in the kernel. Figure 4.9 compares the execution time of such 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 kernel 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 when analyzing the replay. For apache, it takes 50x. On the other hand, for radiosity, with its modest kernel activity, it takes 2.8x.

![Figure 4.9: Execution time of alarm replay checking for ROP attacks in the kernel.](image-url)
4.5.7 Time Window to Respond to an Attack

The 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 recorded execution logged the alarm. Such time window and the size of the resulting log that was generated in between the two times depend on multiple factors. Two of the most important ones are the workload characteristics and the number of machines dedicated to replay. For our system, we measured that the time window is on average a few seconds, and the log size several MBs (Figure 4.6a).

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. In RepChk1, 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 plus two — 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 the entire history recorded. The user can be motivated to do this as the recorded history can be used for forensics or to audit prior executions to detect intrusions.
Chapter 5

REPLAYCONFUSION

Our goal is to design a detector that can detect various attacks — including the parallel ones. Recall that in Section 2.9.2, we noted that the attack protocols target specific subsets of the cache in order to improve their protocol qualities and attack effectiveness. Based on this, we highlight that these attacks are highly sensitive to the configuration of the cache they were designed for. Our proposal, Replay Conflict Confusion (RCC), builds upon this observation and designs a very effective detector.

The main idea behind RCC consists of the following:

- Record the program along with its cache profile during the original execution.
- Replay the same program under different cache configurations and regenerate the cache profile.
- Compare the two cache profiles to obtain the cache miss difference profile.
- Analyze the cache miss difference profile for patterns that indicate covert channel attacks.

Note that the cache configurations are carefully designed to specifically disrupt the contention patterns that are manufactured by covert timing channel attacks.

We describe the details of our design, cache modifications and classification in the following sections.

Threat Model: We assume a strong adversary capable of running code on the victim machine. The typical attack scenario is shown in Figure 5.1. An application has an embedded trojan with access to security sensitive information such as passwords, databases, and security keys. To avoid detection by the OS, the trojan cannot directly leak these secrets to the spy process through traditional inter-process communication (IPC).

As such, the trojan will be limited to indirectly leaking the information bit by bit through the LLC based covert channel. This is the covert channel attack where the trojan is the sender and the spy process is the receiver. This work is therefore not concerned with protecting
against an adversary whose sender and receiver can synchronize or communicate through files, shared memory, or pipes.

We restrict the threat model to the LLC. The LLC is shared by all processors making it the largest and most accessible attack surface for shared cache timing channels. It is accessible even by adversaries conducting attacks in virtualized environments.

We assume that Os or hypervisor is able to secure the record and replay logs against corruption by the adversary.

Finally, we assume the attacker has full knowledge of the cache configuration within the system. Note that although we describe the sender and receiver as being processes, our scenario extends to virtual machines. In that scenario, the RnR framework will be built into the hypervisor. The sender and receiver reside in different virtual machines that are scheduled to cores belonging to same physical processor.

5.1 Overall Design

Figure 5.2 presents the overview of our design, highlighting the components added by RCC to a traditional system, namely, (i) timing deterministic replay (TDR) unit, (ii) cache profile manager, (iii) cache configuration manager and (iv) cache address computation unit.

The operating system is extended with a timing deterministic replay (TDR) [138] to record the application’s non-deterministic inputs along with the system schedule information. This allows the replays on the same architecture configurations to be reproduced with minimal deviation.

A cache profile manager is added to the OS to obtain the cache miss trace and securely save it to storage system for later processing. All required cache information can be obtained from performance monitor unit (PMU), that is common in modern processors.

The configuration of cache address computation unit in the hardware can be characterized
by the cache distribution function it used, \( F = (f_{set}, f_{sli}) \). Modern processors use only one set index function and slice hash function and do not modify them. We propose to make these two functions programmable by exposing a software interface to the cache configuration manager.

During the original execution, the cache is configured to \( F_{def} \) — the default function. During the replay phase, the cache is configured according to a new function \( F_{new} \) — different from \( F_{def} \). A different \( F_{new} \) is used for each process — ensuring that each process will have a different profile for each function \( F_{def} \) and \( F_{new} \).

### 5.1.1 Design \( F_{new} \)

As discussed in Section 2.9.2, most attacks are customized according to specific cache distribution functions, i.e. \( f_{set} \) and \( f_{sli} \), to do group division or achieve controllable conflicts between sender and receiver. Cache behavior of an attack will significantly change when it is executed under a configuration that is different from the one it is tailored for. We illustrate this using a toy example in Figure 5.3.

This example shows the cache misses encountered during the communication of a simple...
two bit message ‘11’. The protocol uses two groups $G_0, G_1$. Recall that a group is a collection of cache sets (Section 2.9.2). Delayed access by the receiver to $G_0$ represents reception of a ‘0’ bit while delayed access to $G_1$ represents a ‘1’ bit.

Plot A shows steps for a sender and a receiver using the same distribution function, $F_{def}$. In step 1, the receiver will access groups $G_0, G_1$ in order to prime the cache. Then, in step 2, the sender will communicate the first bit by accessing the addresses that will displace the receiver addresses from $G_1$. The receiver will measure the time to read both groups in step 3. It will receive a ‘1’ because it detects delayed time to access $G_1$. Steps 4 and 5 repeat the prior two steps to communicate the second bit of the message.

In plot B the sender or the receiver uses different distribution functions. Our techniques work best when the receiver’s accesses to the group result in large differences in the cache profile when $F_{new}$ is applied. This is shown in the figure, where, during the communication periods in step 3 and 5, the receiver’s miss profile is inverted. In practice, the functions do not need to perfectly disrupt the profile during communication.

Plot C shows that when the difference between these two signals is computed, the communications periods are emphasized. Note that implicit communication occurs when the receiver accesses a group that the sender did not flush. Our technique can still detect such implicit communication.

As stated earlier, $F_{new}$ should be designed so that it has negligible effect on a benign program while causing significant disruption to the cache behavior of malicious programs. In the following section, we describe two hardware techniques that enable a programmable $F_{new}$.

Set Index Scramble

The default set index function in modern hardware is

$$\text{index} = (\text{phys}\_\text{addr}/\text{block}\_\text{size})\%(\text{set}\_\text{count})$$

Our set index scramble is to either flip bits or swap bits within the computed $\text{index}$. As a result, the conflicting addresses will still be in conflict, but in a different set within the cache. For example, if we flip the last bit of $\text{index}$, all the addresses previously mapped to an odd set are now mapped to an even set and vice versa. If the new $f_{set}$ is applied to all the processes in the system, the number of cache misses should remain the same. But, if different $f_{set}$ are applied to different processes, cache interference between processes will be different.

A benign program is written in a manner oblivious to the existence of other programs in the system. Hence, cache interference between them will not be affected significantly by set index scramble. However, in an attack, conflicts are generated intentionally based on the
assumption that the same \( f_{\text{set}} \) is used by both the sender and the receiver. This assumption is broken in replay, leading to big cache behavior changes.

Slice Rehash

Recall that the slice hash function maps physical addresses to LLC slices (Section 2.9.5). The default slice hash function in modern system is an xor function on selected bits in the physical address. With the “Slice Rehash” technique, we switch the address bits used by the hash function. We must be careful to keep the hash fair so that the cache behavior of benign programs is not impacted. One of the ways to ensure this is by only changing the original \( f_{\text{sl}} \) slightly. For example, the new function can swap a few bits with the adjacent ones.

Since benign programs are oblivious to \( f_{\text{sl}} \), a well-designed \( f_{\text{sl}} \) will not introduce significant cache behavior changes. A cache line distribution aware attack might leverage knowledge of \( f_{\text{sl}} \) in order to define groups in particular slices. It is for those kinds of attacks that \( f_{\text{sl}} \) is effective. By changing the mapping of the addresses to slices, many conflicts between the sender and receiver will be eliminated causing large deviations from the original execution.
5.1.2 Classification Method

Figure 5.4 details the full detection process of our system. In ①, the cache configuration
manager configures the computation unit to use \( F_{\text{def}} \). When a process is scheduled during ②, TDR starts recording and the profile manager begins constructing its cache miss profile by
reading PMU. Concurrently or later, the log is transferred to a different machine.

Eventually the system is ready to replay the processes and check for possible covert channel
attacks. During the replay machine, in ③, the cache configuration manager configures the
address computation unit to use a different \( F_{\text{new}} \) for each processes scheduled. This \( F_{\text{new}} \) will
forever be associated with that process (and all its children threads). The profile manager will
begin to construct the cache profile for this process under the \( F_{\text{new}} \) configuration. Then, in ④, TDR replays whole system by consuming the log while the cache miss profile is constructed. Eventually – after a target number of processes are replayed – in ⑤, we apply our analysis
methodology to the profiles collected in order to detect covert channel communication.

The entire process described above is flexible. For example, the system administrator can
create a rule to check every untrusted process. Alternatively, one can check only the processes
with suspicious activity, such as frequent execution of timer read instructions or access to a
specific system file containing physical address information.

After obtaining the profiles, we process each of them as follows. The difference between the
two profiles from record and replay is computed as a new profile, called \( P_{\text{diff}} \). For a benign
program, the modified cache configurations will have negligible impact, and hence, most
values in \( P_{\text{diff}} \) should be close to 0. Some “jitter” may be observed due to system noise or
TDR imprecision, but such “jitter” will appear randomly. However, for a malicious program,
a consistently high value in \( P_{\text{diff}} \) will be observed during the decoding phase of receiver, and
small values during the other phases of execution. Our classification algorithm is designed to
look for repeating patterns of consecutive high values in \( P_{\text{diff}} \).

We employ the autocorrelation technique [154] to identify such patterns. We compute the
autocorrelation of \( P_{\text{diff}} \) and generate its autocorrelogram (a chart of the auto-correlation
coefficients with varying lag values). An oscillatory pattern in the autocorrelogram indicates
a periodic pattern in the signal being processed. In our case, we classify a program, whose
\( P_{\text{diff}} \) generates an oscillatory pattern in the autocorrelogram, as an attack.
5.1.3 Implementation Details

Flexibility of Cache Address Computation Unit

The cache address computation unit in LLC is the only hardware unit modified by RCC. As discussed in Section 5.1, the modification is required to increase the flexibility of address computation and add a software interface. In order to support the “Set Index scramble”, we add a special register, \( R_{\text{flip}} \), which can only be configured by cache configuration manager. The index bits that are inverted will be set to 1 in \( R_{\text{flip}} \), while the other bits are set to 0. To support “Slice Rehash”, we add another special register, \( R_{\text{sel}} \), which is used to configure the bits in physical address that are used to compute slice id. The cache configuration manager ensures that the correct \( F_{\text{new}} \) is applied when the corresponding process is scheduled during the replay. Note that these modifications have negligible performance impact on the system as they are quite simple.

Program Execution Correctness

During the modification of cache configurations, the OS must ensure that program executes correctly. In particular, shared memory programs must be handled carefully. Shared addresses from different threads must use the same computation function to correctly resolve to the same cache line.

Hence, during the record phase, the operating system needs to keep track of the shared memory activities. After shared memory allocations are performed in a thread, the kernel will tag that thread and its parent. During the replay, all threads that share the same tag will be assigned the same \( F_{\text{new}} \). Thus, we can ensure that the execution is replayed correctly.

The final issue that needs to be considered is the safety of data while modifying the address computation unit. Since data will be cached, it is possible to compromise coherence. To avoid this scenario there are two possible solutions. Either the cache is flushed and disabled before changing the function or the address computation unit will always use \( F_{\text{def}} \) when executing privileged code. We opted for the latter solution to avoid destroying the cache profile.

5.2 Evaluation

5.2.1 Experimental Setup

In this section, we describe our experiment setup and the benchmarks used for evaluation.
Table 5.1: Record and replay cache configurations.

<table>
<thead>
<tr>
<th>( f_{set} )</th>
<th>( (pa/64) % 4096 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{set} )</td>
<td>( p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31} )</td>
</tr>
<tr>
<td>( f_{set} )</td>
<td>( p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{29} \oplus p_{31} )</td>
</tr>
</tbody>
</table>

Replay \( F_{new} \)

<table>
<thead>
<tr>
<th>( f_{set} )</th>
<th>( \text{swap}((pa/64) % 4096) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{set} )</td>
<td>( p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{30} \oplus p_{31} )</td>
</tr>
<tr>
<td>( f_{set} )</td>
<td>( p_{18} \oplus p_{20} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} )</td>
</tr>
</tbody>
</table>

For the purpose of the evaluation, we assume RnR is available to be used as a system primitive and exclude its evaluation from this work. The design and performance details of RnR implementations have already been discussed and thoroughly evaluated in literature [29, 88, 138, 155]. We instead focus on conducting a thorough evaluation of the RCC detection techniques we propose.

We evaluate our idea using MARSSx86 [156], a full system simulator. We boot Ubuntu 10.4 with a 4GB main memory. The simulator models 2 in-order cores, a 32KB private L1 cache, and a 2M shared L2. The L2 is organized in 4 slices, is 8-way associative, and contains 4096 sets in total.

Table 5.1 shows the different configurations for the cache address computation unit in the record and replay executions. The replay hash function is carefully chosen in order to ensure it has a distribution similar to that used during the record. We verify this by allocating 2M addresses using 512 4K-pages, and get the cache line distribution for both functions in Figure 5.5. In the 4-slice configuration, each slice should get exactly 25% of cache lines if a perfect hash function is used. Since we use a small number of addresses, a slight unbalance is observed. That is, each slice gets 20% to 30% lines using original function. Our replay hash function achieves a similar distribution.

We leverage the check-pointing support in Qemu [87] (the front-end of MARSSx86) and the deterministic feature within program to simulate the effect of record and replay. For unavoidable non-determinism, we ignore it if it is small enough as it does not affect detection precision. Any non-determinism is manually eliminated through slight program modifications.

First, the same checkpoint is used to ensure that the system and applications start with the same state. Second, we make sure the scheduler variance in different runs has a minimal effect on the relative execution progress between programs. Since the SPEC [157] benchmark is deterministic, we count instructions for each time interval to verify minimal deviation between record and replay exists. However, the attack code itself is not deterministic. For example, the receiver spends more time in synchronization if cache hits are decoded instead of cache misses. To make sure they execute in the same relative way as record, we manually
insert synchronizations in both sender and receiver to keep them executing at the same pace as record.

5.2.2 Discussion

We first evaluate how the two cache changes, i.e. set index scramble and slice rehash, impact the behavior of both benign programs and attacks. Then we evaluate the effectiveness of our classification algorithm. Lastly, we evaluate possible approaches to evade detection.

5.2.3 Effects on Benign Programs

In each experiment, we run two benchmarks simultaneously to see how they interact with each other under the specific cache configurations. For this testing, we run bzip2 alongside h264ref, gobmk alongside sjeng, and two stream instances together, both configured using array size similar to the LLC size. During record and replay, every 100,000 cycles, we obtain the program’s instruction count, LLC access count and LLC miss count. The data is processed by grouping every 5000 data access instructions together as a sample, and the cache miss rate per sample is computed as a cache miss profile. In this way, we make sure that the cache miss rate in the replay is computed at the exact same program execution points as the original execution. Cache miss difference, which we call CMD for short, is the absolute value of subtracting the miss rate of the original execution by the miss rate of the replay for each sample.

In Figure 5.6, we show the distribution of CMD for benign programs. The benchmarks bzip2, h264ref, gobmk and sjeng all have clear consistent trends – more than 90% of the samples have CMD of less than 0.2. The cache miss profile of sjeng is shown in Figure 5.7a.
Figure 5.7: Cache miss profiles of benign programs and attacks.
Figure 5.8: Cache miss distribution of attacks using parallel protocol.

Some mismatches between the original execution and replay exist, but the difference is small and the overall behavior is consistent. This is because set index scramble does not impact a program’s self-interference in cache, and the modified hash function creates a similar distribution of addresses to LLC slices as the original one. Similar cache profiles are obtained for bzip2, h264ref and gobmk.

The only exception is the stream benchmark, which shows a higher CMD than the other benchmarks. This is due to the benchmark characteristics and effect from set index scramble. The stream benchmark allocates multiple large arrays and scans them linearly. Figure 5.7b shows 200 samples from cache miss profile of the stream benchmark. We launch two identical instances of stream. During the original execution, since they have the same cache access behavior, they are tracing each other in cache accesses. Data accessed by one process are quickly evicted by the other, thus high miss rate is observed. However, during replay, the set index scramble breaks up this tracing effect – reducing the conflicts observed. This will not be classified as an attack because there is no periodic pattern within this CMD.

5.2.4 Effects on Attacks Using Different Protocols

We evaluate our cache configuration modification’s effect on attacks using different protocols as follows. We first evaluate the attacks using parallel protocol, shown in Figure 5.8.

Multiple Groups

We implement “2g_aware”, the same attack used by Xu et al. [39]. In this attack, the whole cache is divided into odd and even groups based on the last bit of the set index. The sender uses a differential encoding of two groups to send 1 bit each time. We compute the CMD for only the communication part by excluding the idle time, where very few LLC accesses are
We see that 70% of CMD is above 0.4. The cache miss profile in Figure 5.7 shows clear “HL” (high-low) pattern in the original execution, which is flattened out during the replay. Since the attack leverages set index function to divide groups, and two groups have distinctive cache behavior, all conflicts will occur in one of the two groups during the original execution. During the replay, the set index scramble will mix up the two groups, causing conflicts to spread in between them.

Set Index Aware vs Slice Hash Aware

We implement 2 single group attacks, “1g_set_aware” and “1g_slice_aware”, which obtain controllable conflicts based on either the set index function or slice hash function (but not both). The group size is 1/4 of the cache size in both attacks. The sender flushes the whole group by sending bit value ‘1’, and leaves receiver’s data cached to send bit value ‘0’. In “1g_set_aware”, both the sender and receiver choose addresses mapped to the first 1024 sets in the cache out 4096 sets in total; in “1g_slice_aware”, they use all 1024 sets within slice 0.

We show cache profile of “1g_slice_aware” in Figure 5.7c. When the sender is sending bit 0, the receiver gets no cache misses in both the original execution and replay. When the sender is sending bit 1, the receiver gets all cache misses in the original execution, but only one fourth in the replay. It is because in replay, slice rehash redistributes the addresses of one process to different slices. Slice 0 only gets 1/4 of the addresses, resulting in 1/4 conflict misses compared to the original execution. This explains the CMD distribution in Figure 5.8, where for both attacks, half of the CMD is close to 0, and the other half is above 0.5.

Cache Distribution Unaware

In a cache distribution unaware attack, both the sender and receiver allocate a 2MB buffer (the same size as a LLC slice). The sender accesses all of the addresses within the buffer to send bit ‘1’, and idles to send bit ‘0’.

This attack does not leverage any knowledge of the cache configuration. As such, no group division is possible. In Figure 5.7d, unlike the distribution aware attack, the receiver gets around 40% cache misses even though sender does not touch the cache. These misses come from self-invalidation, which the attacker is unable to control. We observe a slightly lower CMD for the cache distribution unaware attack compared to other cache distribution aware attack. However, we still can detect it because these small CMDs will a show consistent repeated pattern within $P_{diff}$.
Round Robin vs Parallel

Our detection methodology works for both the round robin and parallel protocols. Figure 5.9 shows the CMD distribution of round robin attacks using the same spatial features as the parallel protocol. The alternating pattern within round robin results in a clearer measurement. Therefore we are able to observe slightly higher CMD for 2g attacks.

5.2.5 Detect Attacks Using Smaller Number of Sets

One commonly used approach to evade detection is to reduce the group size (the number of sets used in attack). We show by setting the monitor granularity appropriately, we are able to detect attacks with very small footprint.

In Figure 5.10, we show CMD distribution for attacks using different group sizes with a varied monitor granularity. We use a parallel, 2 group and set index aware attack since it is commonly used and highly reliable. In addition, in order to hide high miss rate, the sender and receiver constantly access a group of cache lines which only belong to themselves. This creates a large number of LLC cache hits.

In Figure 5.10a, where the monitor granularity is 2000 data access instructions, 40% CMD of attacks using group size 1024 and 256 are above 0.4, while attacks using 64 sets have a
Figure 5.11: Cache miss difference distribution of benign programs.

lower CMD in the range of 0.2-0.3. This is because the receiver issues a smaller number of cache accesses if group size is small, and the high number of LLC hits lowers cache miss rate in the original execution (in Figure 5.7f). The set index scramble works to bring miss rate down to 0. But the CMD computed is still lower than 0.3.

With a coarser monitor granularity, the CMD is reduced for each attack. If the granularity goes up 10000 data instructions, the CMD of attacks using group size 256 is around 0.1-0.2, and attacks using 64 sets have a CMD below 0.1. Low CMD makes detection more difficult. However, it is also difficult for the attacker to tolerate noise introduced by the other applications within the system. Theoretically, the finer the monitor granularity, the better detection coverage can be achieved.

5.2.6 Classification Algorithm

Figure 5.11 shows an example of how our classification algorithm works to distinguish attacks from normal programs. We use bzip2 as the example benign program. The cache miss profile during the original execution is shown in Figure 5.11a. We observe repeated patterns of high cache miss rates in bzip2. Such patterns are common in programs that contain alternating memory-intensive and computation-intensive phases. The attack here uses a round-robin protocol. It uses a cache distribution aware mapping strategy to pick 2MB of addresses, and divides them into 2 groups. The cache profile of the attack shows a pattern similar to that of bzip2. There are 30 samples with a high miss rate, followed by another 30 samples with a low miss rate. This is where the measurement takes place – the receiver gets all misses in one group and all hits in the other group of cache sets. We observe the oscillatory pattern in
autocorrelogram of cache miss profile in both programs. Therefore, cache miss rate alone is not sufficient to be used to classification.

Figure 5.11b shows the cache miss difference between the original execution and replay of two programs. For bzip2, most samples have very low CMD, in range 0-0.2, indicating our cache changes do not affect benign programs’ cache behavior. There exists a small number of samples with 0.1-0.2 CMD. Such differences are introduced by record and replay interference, system noise, or cache configuration changes. The autocorrelogram does not show any pattern of CMD. For attacks, we get around 0.5 miss rate difference during communication, i.e. the exact execution point where “HL” pattern appears in record trace, and almost 0 CMD during the synchronization time. Since communication happens recurrently, we get a clear oscillatory pattern in the auto-correlation graph. The value to auto-correlation is approximately 1 at lag 70, which is the bandwidth of the communication. This means the attacker transmits a symbol every 70 samples.

In summary, our detection works because attackers communicate by repeatedly creating conflicts, and such conflicts will result in a high CMD after cache configuration changes. Similar patterns are observed for all the attacks we construct.
Chapter 6

REPLAYENDURANCE

Vulnerabilities in hardware are among the most concerning for system security. Hardware forms the basis upon which systems are built — any vulnerability in hardware undermines the security of the entire system. Furthermore, to rectify the vulnerabilities, patching hardware is required — a process complicated by the rigidity of hardware. Since the emergence of the hardware vulnerabilities in Spectre [47] and Meltdown [48], a steady stream of transient execution attack vectors [139, 141, 142, 159–162] have been uncovered. The predominant strategy to mitigate the threat of these vulnerabilities has been, thus far, to plug the leaks in speculative architectures as they are discovered [50, 52]. Unfortunately, new attacks which undermine prior mitigations have continued to emerge.

As visualized in Figure 6.1, the known attack surface for Spectre-style attacks is large. Attacks have been demonstrated which exploit the pattern history table (PHT), branch target buffer (BTB), and the return stack buffer (RSB). Unfortunately, the complexity of the modern out-of-order core makes it likely that even more attack vectors will be discovered in the future. Alternatively, processors can be extended to verify that transient instructions did not leak secrets. Assuming it is possible to measure this leakage, distinguishing between those which maliciously leak and those which inadvertently leak is difficult. Instead of trying to squash these vulnerabilities as they are discovered — an uncertain process — we take a different approach whereby we concede that speculative architecture can leak and focus on detecting the leakage.

![Figure 6.1](image-url)

Figure 6.1: A comprehensive view of the known variants of Spectre-style attacks. [158]
We propose a new security requirement which can be readily enforced: transient execution fungibility. Transient execution should be fungible as instructions whose effects are discarded should never be able to influence those which are enduring. Using transient execution fungibility as a safety condition allows us to prevent transient execution attacks. The difficulty is in measuring the fungibility of transient execution.

We call our approach ReplayEndurance — a defense which verifies that enduring instructions are not influenced by transient instructions and is robust to the emergence of new and presently unknown attacks. ReplayEndurance works because it is safe to conclude that transient instructions should be impotent. Instructions which are discarded should never affect retired program state. Thus, any enduring effects of transient execution can be considered a security violation. To measure the fungibility of transient execution we exploit the concurrency of record and replay. Our approach can detect a fundamental signature which will necessarily manifest during any attempt to recover secrets leaked through Spectre-style covert channels. This signature is a divergence in the control-flow of the recorded attack and the replayed attack. We construct a prototype implementation — which we call ReplayEndurance — to protect embedded JavaScript runtimes and evaluate its performance and security.

Threat Model

We address a threat scenario where attacker and victim code are physically co-located and share a virtual address space. The attacker and the victim share a runtime which schedules their execution and enforces fine-grained memory isolation checks. The runtime is correctly implemented modulo a vulnerability to Spectre v1, v2, or v4 based transient execution attacks. We also assume that the attacker has source of timing information with micro-second scale precision. This assumption is not necessary, as it is possible to exploit by simply increasing the measurement counts to overcome the imprecise timers [163].

6.1 Detecting Enduring Effects of Transient Instructions

Software instructions interact with architectural state (i.e. memory and registers) to implement their programming. Beyond these architectural side-effects are sub-architectural — or micro-architectural — side-effects. These include changes to hardware buffers and caches such as the translation lookaside buffer (TLB) or the last-level cache. Software execution cannot directly observe the micro-architectural state. When a processor executes instructions due to mis-speculation the CPU state is rolled back to the correct state. Only the architectural

\[1\] A fungible quantity can be readily substituted for something of equal value.
side-effects are rolled back. Thus, transient instructions can be leveraged by attackers to leave traces of data in the micro-architectural state.

Since program instructions cannot directly observe microarchitectural state, we argue that transient execution ought to be fungible. Substituting arbitrary operations in place of transient execution is inconsequential to benign software as such software is oblivious to the microarchitectural state. This does not hold for transient execution attacks. These attacks systematically induce transient execution to leak secrets into microarchitectural states. Later, the residual effects of transient execution on the microarchitecture are measured to recover secrets. These recovery instructions are necessarily sensitive to transient execution. For these instructions transient execution fungibility does not hold.

Transient execution attacks rely on instructions which attempt to recover the secret value leaked by the transient instructions. The inherent non-determinism in these instructions violates the fungibility. To visualize this, two control-flow traces of Spectre attacks are presented in Figure 6.2. Time flows along the horizontal axis and in the vertical axis are the potential branch targets. The first trace (a) was captured under ideal conditions and the second (b) under noisy conditions.

These traces are from left to right. First, the microarchitectural state is initialized in the setup and some branches occur. After setup, malicious transient execution occurs. Next, the recovery instructions are executed to compare cache timings and record whether timing-miss or timing-hits occur. These operations are vulnerable to noise. If the recovery instructions and the transient instructions are separated by a large amount of time then some false-hits

![Graph showing control-flow during Spectre attacks]
Figure 6.3: Transient execution attacks leverage the side-effects of transient instructions (1) to recover secrets (2). Modifying the behavior of transient instructions will trigger a different recovery (3). Comparing the original and modified control flow can be used to detect software which is maliciously using transient behaviors to recover secrets (4).

will be registered. This is what causes the divergence between the two traces. In trace (a) one timing hit is registered while in (b) four hits are registered.

To measure the fungibility of transient execution we propose using multi-variant execution. We estimate fungibility by measuring the control-flow divergence between the original execution and a variant in which transient operation was modified. Figure 6.3 overviews our strategy. Transient operation (1) is replaced with alternative transient instructions in (3). Thus, when the original recovery instructions (2) are compared to the new ones (3) control-flow divergence can be detected (4).

```plaintext
for(...) //To denoise, repeat attack
  init_uarch_state(); //Train branch, init cache
  idx=calc_leaky_index(&Secret);
  vuln_code(input); //Trigger transient execution
  for (...) {
    latency = time_access(Buffer, i)
    if (latency < DRAM_LATENCY)
      counts[i]++ ; //secret might be i
    if(counts[i] > MAX) MAX = counts[i]
  }
  if(MAX > THRESHOLD)
    break; //Stop attack once confident in secret
}
secret = find_best_candidate(counts);
```

Figure 6.4: Spectre attack simplification. Highlighted statements modify control flow based on transient behaviors.
6.1.1 Detailed Example

To understand why fungibility works we can take a detailed look at Spectre. Consider the code in Figure 6.4. The highlighted statements are control flow statements which test timing results to determine the result of the Spectre attack (i.e. the recovered secret). The dynamic behavior of the branch instructions which implement these statements is determined by the transient instructions. For example, consider line 7. The compiled code will contain a conditional branch to skip the remaining loop statements for the cases where the observed latency corresponds to a cache hit. This condition will manifest if and only if transient instructions touch this memory location. Whether this occurs will depend on the value of the secret being extracted by the transient instructions.

Finally, consider lines 9, 11 and 12. These conditionals work together to test when the attack can be terminated. If the environment is noisy, the attack will need many — possibly tens of thousands — of repetitions to succeed. In more ideal conditions, the outer loop (line 1) can be terminated early. Figure 6.2 visualizes these control flow patterns by presenting branch traces from a Spectre-style attack on real hardware.

The x-axis represents time and the y-axis provides descriptions of the function of the targets. Both traces start with some setup code followed by the transient targets. The divergence can be seen in the loop which iterates across possible secret values while performing timing tests. With 6.2a, the attack yields a clean reading with a single secret value yielding a timing hit. On the other hand 6.2b shows a case where a noisy environment rendered three additional candidates — any one of which may be the actual value. To denoise the attacker is forced to repeat the attack until one candidate stands out.

6.1.2 Measuring Fungibility on Commodity Hardware

Benign execution should never be influenced by transient execution. For this reason a sensitivity to prior transient operations can be taken as a tell-tale signature of transient execution attacks. Based on this principle we propose a new strategy to detect transient execution attacks. Our strategy is visualized in Figure 2.7. Transient execution attacks access secrets during transient execution (1). To recover the secrets, retired instructions use side-channels to observe the residual microarchitectural effects of transient execution (2). To measure the sensitivity of an execution we can execute a variant of the program which modifies transient behaviors. This modification heavily impacts transient execution attack-code and results in the recovery of a different secret (3). This divergence in outcome reveals the presence of non-fungible transient execution. The most immediate manifestation of this divergence can be seen in the control-flow divergence (4). By tracing a program’s
control-flow we can detect these divergences.

6.1.3 Enhancements via Hardware Support

Enhancing this technique is possible. One option is to use “Speculative Store Bypass Disable” support on Intel processors. Another would have processors expose speculation as a capability which the OS can grant to applications. This would greatly simplify the implementation of our techniques as it would allow us to compare program traces with and without speculation. It is trivial for the processor to support a mode whereby speculation is disabled. If this mode is enabled, the processor will issue a fence operation with every conditional branch. The fence will execute prior to the conditional branch test. With such a mode, the overall process is simplified. To generate variant branch profile software can simply enable this mode and repeat the JavaScript program while tracing control flow.

6.2 Software Prototype

The availability of software mitigation strategies and branch-tracing support allows us to prototype our ideas using software. We call our prototype ReplayEndurance. Our execution environment is modeled after serverless environments (e.g. AWS Lambda or Google Cloud Functions). This environment has runtimes which host many cloud functions from various applications. The runtime schedules these functions as they are triggered. The functions are implemented using languages like JavaScript, Java, and Python. All communication across functions proceeds through IO operations such as persistent storage or network communications. These functions most often persist their results using cloud APIs. We build our runtime using the Mozilla SpiderMonkey JavaScript engine.

6.2.1 Variant Generation

We generate variants to detect Spectre v1, v2, and v4. These are the Spectre attacks which enable the functions of a serverless runtime to attack other functions. To enable the measurement of transient execution fungibility the variants must alter the behavior of transient instructions.

For Spectre v1 (bounds check bypass) there are a couple of variants which we can use. One could, for example, generate a variant where all memory that is not associated with the function is corrupted. Alternatively, one could prevent the speculative bypass of bounds checks by using fence instructions. We opt for the second approach. We build a clang
source-to-source compilation pass which adds a 3 byte NOP instruction in the body of every if/else block. To enable or disable the variant these three bytes can be replaced with the op-codes of a fence instruction or reverted to a NOP. This fence effectively creates a variant execution where the transient instructions are turned into NOP instructions. For Spectre v4 (speculative store bypass) we leverage the Intel mitigation aptly named “Speculative Store Bypass Disable”. By disabling this form of speculation we can create a variant which will not transiently bypass stores. In this manner, no leakage can occur during transient execution.

The retprobe mitigation for Spectre v2 (branch target injection) can be used to generate a variant which mitigates the attack. This mitigation transforms indirect calls into instruction sequences which kill speculation. While more involved, the approach to generate this variant is fundamentally identical to our v1 approach. Instead of a three byte NOP at least 19 bytes are required to transform a 6 byte indirect call into the instruction sequence implementing the retprobe.

6.2.2 Multivariant Execution

We can benefit from our serverless execution environment to simplify our multi-variant execution model. Prior to executing the triggered JavaScript function, we use fork() to clone the runtime process. The child will then activate branch tracing, enable its variant capabilities, and proceed to execute the function. The parent will also go on to activate branch tracing and proceed to execute the function. Once the functions are executed the branch traces can be processed and analyzed for online detection or stored for future analysis. Our v1 and v2 variants require code instrumentation. This can be achieved via a kernel driver. The v4 variant will make a specialized prctl() call to flip the SSBD switch.

6.2.3 Runtime Overview

An overview of our architecture is presented in Figure 6.5. We present the four relevant layers of the platform: the application, runtime, system, and hardware.

In hardware, we assume the availability of chicken-bits which can disable various forms of speculation. These bits are flipped using model-specific registers. In our experiments we emulate these capabilities using a combination of software and system support. One bit will disable speculation beyond conditional branches. If set, hardware will implicitly issue a fence following every branch instruction – regardless of the target.

In the system layer we place a lightly customized branch tracing driver. We modified our driver to optimize its performance for online-analysis use-cases. We assume our runtime is hosting ephemeral cloud functions that are scheduled in event-driven fashion. This
simplifies our implementation, allowing us to rely on fork() to run the variants. We wrap our JavaScript scheduling code and context management code with functionalities related to security policy enforcement and trace analysis.

6.2.4 ReplayEndurance Operation

We can leverage our fungibility estimations as a detection primitive around which to build security systems. When a function is triggered the runtime will prepare for execution. After some setup, the cloud functions will be ready for execution. Prior to execution fork() is used to create a master and child process. Both processes will activate branch tracing. The child process will activate variant execution. At this point, the process states are equivalent modulo the state of memory-mappings and the configuration of transient-operation. The cloud function will begin execution on both the child and master process. Throughout execution the function may invoke some runtime-implemented operations (e.g. to interact with cloud services). Any output (e.g. Line 10 from Figure 2.2) is buffered until the function’s execution was deemed safe. Once execution completes, the runtime will measure the fungibility of transient execution by comparing the two branch traces for divergence. If an attacker attempted to recover secrets during the function he will fail the fungibility test.

There are two attack strategies available for attackers targeting serverless applications. The first involves an attacker performing all attack steps within his malicious function. This means the attacker attacks the vulnerable code, collects microarchitectural timing variations, and processes them to interpret the secret. Alternatively, an attacker can improve his stealthiness by streaming timing measurements off-machine for offline post-processing. The first attack strategy is significantly less expensive to mount in a sustained and widespread fashion. The second attack offers the advantage of avoiding any detection technique built around observing
the secret recovery step in speculative execution attacks. For this reason, ReplayEndurance will discard the outputs produced by the vulnerable execution in the case that the attacker may have observed time. The outputs from the variant which mitigates transient execution attacks can be used.

Potential Circumvention

ReplayEndurance is built on monitoring the execution of an attacker in two environments and monitoring for divergent behavior. Circumventing ReplayEndurance necessitates recovering the residual effects of transient execution without creating any observable state-transitions. This can be accomplished only if the attacker can avoid creating divergence in any execution state that is being monitored. For example, if ReplayEndurance monitors control-flow the attacker can use predicated execution (e.g. conditional mov instructions). This is not a viable approach because the attacker only controls the JavaScript and not the compiler. The compiler can simply prohibit predicated instructions to prevent this form of circumvention.

It is possible for the attacker to stream the timing results for offline analysis. This is viable as the attacker could recover the secret on an unmonitored machine. The drawback is that this approach is tremendously expensive. Even in the most ideal conditions over one kilobyte of timing data is needed to recover 32 bits of secret data using Spectre v4 (speculative store bypass). Even if we ignore the effects of noise the attacker must overcome imprecision in the location of the secret. The secret could reside in any virtual memory location on any runtime. If the attacker could narrow down the location to 1000 possibilities it would still cost over $100 to leak a 96byte RSA private key. It is more likely for the attacker to have to consider millions of potential locations making this attack economically unsustainable for most threat models. To further defend against this threat defenders can configure ReplayEndurance to discard the outputs which originate from the vulnerable machine whenever the attacker may have access to timing information.

Dealing with False-Positives

Our strategy of detecting the attacks via detecting control flow divergence could implicate benign applications. The potential for non-deterministic behaviors in applications – e.g. non-malicious testing for external service response time-outs – could create benign control-flow divergence. This is problematic in serverless computing environments since millions of function invocations for an application are not uncommon, making even a highly infrequent imprecision a source of frequent false-alarms. For this reason, we designed our security policies to contain programs which are implicated by our detection scheme. The worst-case outcome
benign application is private execution (policy 1) or longer latencies (policies 2 or 3). In this way we can avoid the burden of being forced to remove all imprecision in our defense.

6.3 Evaluation

6.3.1 Experimental Setup

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>box2d</td>
<td>2D Physics engine</td>
</tr>
<tr>
<td>cube3d</td>
<td>3D Physics engine</td>
</tr>
<tr>
<td>crypto</td>
<td>Encryption (int heavy)</td>
</tr>
<tr>
<td>hashmap</td>
<td>Runtime-persisted hash-map</td>
</tr>
<tr>
<td>mandrel</td>
<td>Bullet physics (277k lines of JS)</td>
</tr>
<tr>
<td>morph3d</td>
<td>3D CAD benchmark</td>
</tr>
<tr>
<td>pdfjs</td>
<td>Mozilla PDF reader (33k lines of JS)</td>
</tr>
<tr>
<td>raytrace</td>
<td>Ray tracer benchmark</td>
</tr>
<tr>
<td>raytrace3d</td>
<td>Ray tracer benchmark (3d)</td>
</tr>
<tr>
<td>recursion</td>
<td>recursion microbenchmark</td>
</tr>
<tr>
<td>regexp</td>
<td>Regular expressions benchmark</td>
</tr>
<tr>
<td>Richards</td>
<td>OS kernel simulation benchmark</td>
</tr>
<tr>
<td>splay</td>
<td>Splay-tree microbenchmark (GC heavy)</td>
</tr>
<tr>
<td>sieve</td>
<td>Prime numbers generation</td>
</tr>
</tbody>
</table>

Performance Evaluation

To evaluate the cost of adopting multi-variant execution to protect against JavaScript attacks we implement an embedded JavaScript engine using Mozilla SpiderMonkey. We choose to use an embedded JIT as interpreted languages like JavaScript, Python, Java, C#, and Ruby are common in serverless applications. Furthermore, the multi-context runtime is exactly the computing environment most vulnerable to Spectre-style attacks. Table 6.1 lists the benchmarks we used to evaluate our work. Eight of the benchmarks are pure JavaScript benchmarks which originate from (V8, WebKit, and SpiderMonkey). These benchmarks allow us to estimate the performance cost of our software defenses and project the potential improvement in the case hardware acceleration is added.
Security Evaluation

To evaluate the potential for multi-variant execution we generated multiple binaries for Spectre v1 and v2. For Spectre v1, for the variant we add an lfence within every conditional block (both if and else). We implement a clang source-to-source compiler to achieve this. For Spectre v2 we leverage retpoline to generate a mitigated binary. For Spectre v4 we leverage hardware support which allows software to disable speculative store bypass. This functionality was added by Intel via micro-code patches. To trace branches we use Intel Processor Trace and a lightly modified kernel driver to reserve buffers.

6.3.2 Discussion

![Graph 1](image1)

**Figure 6.6:** ReplayEndurance component costs. Record and Replay are needed for multi-variant execution and branch tracing is used for divergence detection.

![Graph 2](image2)

**Figure 6.7:** Overall performance degradation suffers most from logging overheads necessary for multi-variant execution.
Multivariant Execution Costs

Figure 6.6 presents the online costs of a Multi-Variate Execution implementation based on Record and Deterministic Replay. We use Intel Processor Trace to trace the retired branches. Our recursion benchmark represents an extreme challenge for PT due to the dominance of control-flow instructions. In this case, a 30% overhead is registered for an application which is dominated by call and return instructions. These instructions add additional packets to the trace to disambiguate the branch targets. We record the results of system calls and the results of any changes to runtime-maintained data-structures. Hashmap is a benchmark which stresses the runtime logging, adding an overhead of nearly 33%. Cube3d and raytrace3d stress arrays and floating-point math in a relatively short-running code. This inflates the overheads of MVE.

Policy Enforcement Costs

Figure 6.7 presents application level performance when ReplayEndurance is used to generate branch traces and test them for divergence. Each plot presents the performance of the applications versus the number of runtimes running. Each runtime is single-threaded and enforces one of four policies. The data is normalized to a single runtime.

The points corresponding to the “none” line represent baseline operation where ReplayEndurance is not involved. The points from “logging” add the branch tracing, system call recording, and initial fork for the checkpoint creation. No security online security analysis or policy enforcement is performed with logging. The overall costs of logging can vary and depends on the nature of the workload. For instance, the richards benchmark results in waiting behavior due to the device tasks. These tasks simulate waiting for data from an external device. This slack leaves CPU time available for variant execution analysis and logging overheads which, in turn, soften the cost of the relaxed and safe policies. Without this slack, CPU intensive workloads like box2d and morph3d will struggle to close this gap – despite the lower overheads (Figure 6.6).

Impact on Benign Applications

ReplayEndurance exposes the sensitivity of malicious retired instructions to transient behaviors. Retired instructions in benign applications are affected by the register and memory state and not the microarchitectural state. Malicious transient execution attacks retire instructions which are influenced by the microarchitectural state. Figure 6.8 presents divergence in branch targets across two executions of bzip2. The executions are on the same input and each bar presents the number of diverging branch targets for an individual branch trace.
Figure 6.8: Deviation across variants during benign execution (bzip2). Each bar corresponds to a single branch. Most branches suffer no deviations.

Aside from transient execution, the two executions are identical. The first set of comparisons (Spec-NoSpec) is for executions where one execution proceeds with mitigations for Spectre v1 in place while the other execution proceeds without mitigating Spectre v1. We instrument the mitigation using a clang source-to-source compiler. The traces of the majority of branches (over a thousand) are equivalent despite the mitigations. The 15 diverging targets (across 6 branches) can be attributed to non-bzip2 branches which occur at load time and in shared libraries (e.g. glibc). These same exact deviations are present when comparing two runs in which neither execution mitigates Spectre v1. The same profile manifests when mitigating Spectre v2 (via retpoline) and Spectre v4 (via Speculative Store Bypass Disable).

Detecting Spectre Attacks

Figure 6.9: Deviation between variants during Spectre v1 attack, 25 branch traces are compared across 2 executions — one with mitigations for Spectre v1 and one without.

Spectre Control-flow Patterns: We detect Spectre attacks by measuring the sensitivity of the application to transient behavior. Transient execution attacks which test the microarchitecture for traces of leaked secrets will experience control-flow differences if transient execution is modified. Figure 6.9 presents divergence counts for a Spectre v1 attack when comparing branch targets across two executions. In the first execution (Spec) speculation is allowed and the attack is generally successful. By re-executing after instrumenting fence instructions in place of 3 byte NOP instructions, the speculative leakage will never occur. This causes
(a) Deviation by branch  (b) Branch counts (by SRC PC)  (c) Branch counts (by DST PC)

Figure 6.10: Spectre v4 branch profile. Nine branches implement the recovery phase of the Speculative Store Bypass (SSB) exploitation. These branches test the observed latencies and count the number of delayed instances. The cumulative deviation can mask the attack as seen (b) and branch destinations (c) when SSB is disabled or enabled is largely identical. To detect the deviation the time-varying behavior needs to be monitored.

Figure 6.11: Control-flow when repeating the execution of a deterministic function will not deviate from the original execution. For Spectre v4 attacks, deviations are plenty due to sensitivity of the attack to transient instructions.

diverging outcomes in terms of the recovered secret and diverging branch targets. Two branches exhibit over 1000 instances of diverging branch targets and, in addition, seven branches have hundreds of deviations. Since both executions start from the same initial state these deviations can only be attributed to the altered transient execution. This is the tell-tale sign of speculative execution attacks.

Detecting Spectre Through Deviation:  The deviations can manifest through a small number of branch instructions. For Spectre v4 attacks three instructions cause all the divergence. Some of the divergence is seen in the instruction performing the branch (Figure 6.10b) or the different branch destination (Figure 6.10c). Altogether, when non-application branches are filtered, a clean signal can emerge. Figure 6.11 presents a slice of time from a Spectre v4 attack. After recording the attack, execution is repeated with speculative store bypass disable
toggled to mitigate the attack. The results are presented — a steady stream of deviated branches by both source and destination.
Chapter 7

RELATED WORK

7.1 Security Defenses

Control Flow Integrity: Enforcing Control Flow Integrity (CFI) [94] 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 [114] 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 [93] re-purposes a multibyte NOP instruction to mark valid destinations for indirect branches. Other approaches [96, 97] define validity via the proximity of the branch destination with respect to function boundaries. In general, such approaches fail to completely eliminate gadgets [114, 164], permitting ROP payload construction. Moreover, shadow stack integrity and longevity of CFI are additional points of concern for CFI [165, 166].

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 [119–121, 167]. 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 [168–170]. 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 for Security: Bezoar [171] 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 [22]. 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.

Unlike RnR-Safe, Aftersight assumes that replay analysis in full 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.

Instead, RnR-Safe’s architecture presents the key practical aspects of online RnR security analysis. They 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 the analysis.

Memory Safety: Guaranteeing memory safety can entirely eliminate code reuse attacks. A memory access is safe when it is conducted through a valid memory reference, and when it does not violate the bounds of the original allocation. Ensuring the first property requires tracking the lifetime of every pointer, detecting any dereference of a pointer that has been freed. Ensuring the second property requires checking every pointer dereference to ensure it is within the boundaries of the structure [172, 173]. Recently, Intel has begun to include in its processors hardware accelerated bounds checking via the Memory Protection eXtensions (MPX) [9]. While MPX is the most significant step towards providing memory safety on commodity systems, it does not protect against the first property mentioned before. Without this property, the use-after-free attack surface remains. Moreover, it has at least two usability concerns. One is that bounds updates can race with memory accesses. This may lead to time-of-check-to-time-of-use vulnerabilities in multithreaded codes. Another is that a compromised kernel can bypass MPX by disabling or corrupting its checks. Hence, it is difficult to depend on MPX for kernel protection.

We have discussed CC-Hunter in Section 2.9.4. Our work is different from CC-Hunter in the following ways. CC-Hunter relies on alternating patterns of conflicts between sender and receiver. This only occurs for round-robin based protocols. We instead focus on the spatial aspects of protocols for our detection methodology. Our detection coverage is larger than that of CC-Hunter as we can also detect parallel protocol based attacks.

Chen et al. [138] propose to use timing deterministic replay (TDR) for network covert channel detection. The application with a potential attack is recorded, and the input is replayed using a known clean implementation of the application in the replay. Significant timing deviation observed in network packets indicates a possible attack. Our work is different from theirs in two ways: First, we focus on a covert channel in cache. Their approach does not detect cache-based covert channels, because the timing deviations introduced by variations in
cache behavior may not be significant enough to be observed by the detector — resulting in either a high false positive or false negative rate. Second, we do not require a clean version of application, which is not always available as a reference. We record and replay the same binary and detect attacks by the attacker’s high sensitivity to cache configurations.

Two classes of mitigation techniques have been proposed to prevent cache covert channel attacks: cache partitioning or run-time diversification. Cache partitioning will isolate an attacker’s cache activity. Static partitioning [174] causes cache under-utilization and creates, in turn, significant performance degradation. Stealthmem [40] proposes cache partitioning against side channel attacks, using dynamic page coloring. The system manages a set of locked cache lines which will never be evicted. Secure sensitive applications can request using these locked lines to hide their cache activity from other processes. Catalyst [44] proposes a hardware-software hybrid approach by leveraging Cache Allocation Technique (CAT) introduced in recent Intel processors. The LLC will be dynamically divided into a SW managed secure part and an insecure part. PLCache [41] supports fine grained cache locking using special load and store instructions. While these approaches can work towards side channels with tolerable performance overheads they cannot efficiently eliminate covert channel attacks due to the requirement of isolation across a large number of processes in different VMs.

Wang and Lee [41] propose to dynamically randomize memory-to-cache mapping in L1 so the attacker cannot create controllable conflicts. This works for side channel attacks which require precise cache mapping information, but as discussed in Section 2.9, cache distribution unaware covert channels can still be constructed. Moreover, an efficient design for LLC is not explored due to its large capacity.

Fuzzytime [43] and Timewarp [42] disrupt the timer by adding noise to reduce the accuracy of the system clock. This approach cannot prevent covert channel attacks as attackers can simply use a larger number of sets for communication which is robust to the added timer noise. Furthermore, this approach will affect benign programs which require a high-precision clock.

A key contribution of this dissertation is mechanisms for detecting Spectre-style attacks against commodity systems. As far as we know, this is the first detection technique proposed to exploit an approach based on record and replay based multivariant execution strategy to expose malicious dependencies on transient execution. Other approaches in literature focus on mitigating these vulnerabilities by eliminating the leakage of speculative architectures.

Transient execution attacks use microarchitectural state changes during transient instruction as a covert side-channel. To protect against Spectre-type attacks effort must be taken to either reduce the accuracy of these covert channels or to prevent them entirely. Tracking the flow of information between speculative and non-speculative states in order to prevent
the flow of information from transient instructions to enduring ones is another fundamental defense. By tainting any speculative results the pipeline can ensure that speculative results are not prematurely encoded in any microarchitectural state.

Another approach is to focus on isolating the effects of speculative execution to prevent all impact on non-speculative execution. This approach has been more popular in processor architecture literature [50–52] as it can prevent the attacks without difficult changes across the microarchitecture. SafeSpec [51] takes this direction by proposing the replication of microarchitectural structures. During speculative execution the replicas are exclusively used, thereby preventing any pollution during transient execution. In the case of misspeculated branches the pipeline will revert microarchitectural state changes which manifested during the transient instruction execution. SafeSpec applied this approach to prevent channels which flow through the cache and TLB attacks; other attacks – such as those which rely on port contention [140] or the return-address stack [139] – will remain possible.

Buffering the results of speculative operations until they become non-speculative is a strategy proposed by InvisiSpec [52]. In the case that the operation becomes unnecessary due to misspeculation the buffered results can be discarded without impacting microarchitectural state. InvisiSpec [52] used this approach to make speculative loads invisible thereby eliminating the potential for load instructions to be used during transient execution based covert channels. To completely safeguard against transient execution attacks this strategy buffering must be applied across all locations where microarchitectural state-change can transiently occur. DAWG [50] partitions the cache across its ways to mitigate cache-based covert channels by isolating cache-hits within protection domains. With protection domains that isolate on a cache hit, cache miss and metadata level, cache-based covert channels are mitigated.

Other directions call for embedding CFI principles into the branch prediction decisions [175]. These approaches offer the potential for a fundamental solution which can enable limited fencing to secure against Spectre-style attacks. While we believe that such solutions will eventually close the attack surface of transient execution attacks, we still believe approaches like ReplayEndurance are valuable. First, they offer the potential for securing against unknown threats. There will always be a potential for new attacks and as such having complementary software defenses will create more robust security. Second, approaches like ReplayEndurance establish a security system built around actively probing untrusted code to determine maliciousness. Enabling these approaches with hardware acceleration can lower their costs and increase their feasibility.
7.1.1 Software Defenses

Different projects use different techniques to mitigate Spectre. WebKit replaces array bound checks with index masking which prevents out-of-bounds accesses from even occurring during transient loads. WebKit also uses a pseudo-random poison value to encrypt pointers to ensure that attackers cannot construct predictable transient execution from mis-speculation during type-checking branches. Isolating security contexts across processes is the approach adopted by Chromium with site-isolation. With this approach, successful transient execution attacks are limited to reading attacker data or the runtime meta-data necessary to host the attacker context. The Google Chrome browser takes this approach to mitigate Spectre. For multitenant software built using Chromium, this approach can reduce performance by orders of magnitude due to the increased overheads of multiprocess designs.

As an alternative to complete defenses, one approach is to make it difficult for attackers to recover transient results. By reducing the precision of timing mechanisms it becomes difficult to distinguish changes in microarchitectural state. This approach has been taken by all modern browsers. To successfully attack using low-precision timers attackers are forced to repeat their measurements. With enough imprecision the repetition counts are beyond what is feasible.

7.2 Concurrency Management

TLB Shootdown Hardware: The application that we evaluated is the TLB shootdown, for which there have been many hardware and software based proposals for minimizing the cost. HATRIC [176, 177] proposed maintaining TLB coherence by piggy-backing coherence protocols with new flows for the TLB. HATRIC tracks sufficient state in the LLC to precisely identify all tasks which share a page precisely identify. The primary goal of this design was to improve the handling of nested pages. SITE [177] takes a similar approach but instead tracks the mapping between TLB entries and physical addresses of the PTE entries. From there, any invalidation can be looked up in the PCAM to determine if TLB entries require invalidation. Unlike HATRIC, this approach cannot support nested page tables. TLB shootdowns can compromise the benefits of having fast on-chip DRAM in 3D stacked systems [178], so the authors propose microcoding the entire process. The microcode sequence can detect a special interrupt vector and proceed to process the shootdown request at a cost of approximately 2500 cycles for a 16 CPU shootdown. ConcAcl TLB invalidation commands do not require the pipeline serialization and are able to acknowledge them faster. DiDi [179] adds DiDi — a centralized structure to track which processors have accessed an address space and thus may have dirty TLBs. ConcAcl is not centralized.
TLB Shootdown Software: There are ways to improve the TLB shootdown process without changing the hardware. Improving the precision of the target set construction will reduce the number of cores needing interrupts – which can improve the cost of initiating and completing the shootdown as well as reduce the impact on the system [180, 181]. This is accomplished by enabling the OS to determine if a processor never translated an entry by privatizing the page tables and using accessed bits [180] or by emulating SW managed TLBs to track when pages get accessed and when they become shared in SW. Other techniques include batching shootdowns [182] and lazy-tpb invalidation [183, 184] which delay the initiation and completion of shootdowns for improved system performance. Barrelfish [7] uses topology aware multicast messages to implement the shootdown by having certain cores forward the shootdown message, eliminating inter-socket contention costs.

OS Scalability: Barrelfish [7], an experimental OS based on the multikernel approach, aims to avoid inter-core sharing and instead relies only on explicit communication via message passing. They propose using remote procedure calls, in a manner similar to what is done in distributed systems, with a full-fledged name and monitoring service. Without HW support they were forced to implement their RPC channels using cache-coherent memory. At best the latency for recognizing one of their messages is roughly 68ns, but this requires constant polling by software. This is one of the reasons why the authors anticipated that eventually dedicated HW channels would be designed to route and process the messages via programmable peripherals.

Hardware Broadcast Primitives: The WiSync architecture extends the cores of a many-core processor with two antennas and 16KB of replicated memory to house the broadcast variables [185]. The two wireless antennas support two broadcast channels to update data and synchronize. Such hardware can be used to implement the interconnection network of ConcAcl.
Chapter 8

CONCLUSION

In this thesis we proposed techniques for exploiting concurrency for performance and security. We presented ConcAcl, an accelerator for concurrency management. ConcAcl prevents scalability bottlenecks by accelerating coordination with dedicated hardware resources that simplify cross-core coordination. Furthermore, by reserving a single intermittently activated hardware context and providing it with fine-grained control over SMT threads, the OS is able to offload latency-critical functions onto the ConcAcl context. Using these capabilities empowers the OS kernel to implement concurrency management functions in a non-intrusive manner. On the security side, we proposed a framework for achieving robust security—by recording system execution and exploiting concurrent replay in a technique we call RnRSafe. Leveraging the RnRSafe methodology we defend against indirect attacks which violate program security. Our framework enables implementing shadow-stack security to prevent return oriented programming attacks at modest overheads. With ReplayConfusion, we protect against covert channels which exploit the last-level cache to communicate. We detect these attacks in a sound manner by perturbing the mapping between the physical-address to cache-addresses during replay. This technique betrays covert channels by creating large divergence in the cache miss patterns between the original recorded attack and the replayed attack. In ReplayEndurance we propose a robust mechanism for detecting transient execution attacks based on measuring a security property which we name “transient execution fungibility”. As transient instructions are discarded and ignored by the processor we argue that transient execution should be fungible and arbitrarily replaceable. We measure the fungibility of transient execution by recording execution and altering the behavior of transient instructions during replay. The replayed execution is compared to the recorded counterpart to assess the fungibility of transient execution. Control-flow divergence between the two executions indicates that transient instructions were not fungible as they should have been. For both ReplayConfusion and ReplayEndurance, the concurrency between record and replay is critical to enable online security. With ReplayEndurance we are able to exploit scheduling by leveraging the multi-context nature of the JavaScript runtime to hide any delay in the security analyses.
BIBLIOGRAPHY


[93] Intel Corporation, Control-flow Enforcement Technology Preview, June 2017, no. 253669-033US.


