© 2018 Read Sprabery

CAPABILITIES FOR CROSS-LAYER MICRO-SERVICE SECURITY

BY

READ SPRABERY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Roy Campbell, Chair
Professor Carl Gunter
Assistant Professor Adam Bates
Dr. Hamed Okhravi, MIT Lincoln Laboratory

Abstract

Shared infrastructure computing has become ubiquitous; from the smallest start-up deploying on a multi-tenant cloud to the largest corporations whose separate branches all deploy to a shared private cloud. In both cases, the security challenges are similar and are unique from the legacy model of deploying monolithic applications on dedicated hardware. In the case of a multi-tenant cloud deployment, attacks can stem from other tenants who are not part of the same security domain, be that a different security-level within a single organization, or distinct organizations on a public cloud. In addition to nearly ubiquitous adoption of shared infrastructure, the rise of so called "micro-services" poses a set of unique challenges and advantages to security. The micro-service moniker stems from the idea of a Service Oriented Architecture (SOA) with a focus on having a small code base for each component of an application. The SOA approach is complimented by the DevOps movement in which software development practices are being applied to operations. These development and deployment techniques are here to stay as they enable more thorough testing, reliable deployment, and scalability that previous software architectures only supported with extensive rewriting. In this dissertation, we focus on providing security to this new paradigm of computing. These trends force us to face security challenges unique to cloud computing such as passive cache-based side-channel attacks. In addition to new challenges, this new paradigm also affords us better tools and services due to the well-defined behavior of micro-services. Here, we focus on mitigating security risks by leveraging the Principle of Least Privilege (PoLP) at every layer of the stack: the interface between the operating system and the hardware, the system call interface, and within individual applications. We implement the PoLP through layer specific capabilities by mapping the security challenges present in cloud computing to a Take-Grant relational model between subjects. We conceptually extend the notion of "subject" to include subjects at every layer of the cloud stack.

Additionally, we explore adding more trust guarantees to subject relationship monitoring. Finally, we explore fine grained memory operations within a micro-service that can impact a micro-service's relationships with other subjects in the system.

To my friends and family, for their love and support.

Acknowledgments

I would like to begin by thanking my advisor, Roy Campbell, for his guidance and support. His advice guided my development as a researcher and was invaluable throughout my graduate school career. These past five years have been filled with meetings which shaped not only this dissertation, but also my overall approach to research. Thanks for being a great mentor and friend, Roy!

I would also like to express my gratitude to my committee members, Carl Gunter, Adam Bates, and Hamed Okhravi, whose comments and questions helped me to contextualize my work. Their feedback and encouragement helped to shape this dissertation and I appreciate the time and ideas they have contributed to this work.

Many thanks are owed Zak Estrada for his encouragement as I delved into operating system development. Zak introduced me to the world of low-level development and provided technical guidance throughout my graduate career. Without him, this work would not have been realized.

I have also appreciated the insightful advice from Zbigniew Kalbarczyk. He encouraged me to find the core contributions of my work which has helped me provide more clarity in my writing and presentations. Thank you for your thorough reviews and the great advice.

Additionally, I would like to thank all of the wonderful people I worked with at MIT Lincoln Laboratory: Thomas Moyer, Rick Skowyra, Nabil Schear, and Hamed Okhravi. Their guidance on everything from low-level technical details to writing style has made the final portions of my dissertation possible.

Thanks to Thomas Morris who hired me as an undergraduate researcher in his security lab. Morris' lab provided my first exposure to security research and scientific writing. Without that first introduction to research, I might never have pursued graduate school. Thanks for the opportunity.

My current and former colleagues in the Systems Research Group deserve special mention. They have made my time here quite memorable and have been a constant source of inspiration and entertainment. I want to thank Imani Palmer and Hadi Hashemi for their honest feedback as I prepared various presentations. Sitting through an entire defense is tedious at best - and they did it twice! I'd also like to thank the other Ph.D. students Faraz Faghri, Mohammad Babaeizadeh, Shadi Noghabi, Xiao (Chris) Cai, and Hassan Eslami who have all been very supportive. All of you have helped to provide a Ph.D. support group of sorts - the program would have been a lot less fun without you! I'd also like to thank the numerous other SRG members I've worked with over the years including: Arian Azin,

Mohammad Ahmad, Shayan Saeed, Chaitanya Datye, Jigar Rudani, Konstantin Evchenko, Fangzhou Yao, Mayank Pundir, Kevin Larson, and John Bellessa. Each of you have made an impact on me as we worked on compiler homework, database projects, and enjoyed meals as varied as Thanksgiving dinner to traditional Russian meals.

Of course, no dissertation would be possible without tremendous help from the front office. Mary Beth Kelley was an invaluable source of knowledge concerning the complex landscape of academic approvals along with Maggie Metzger-Chappell, Holly Bagwell, and Kathy Runck. Prof. Campbell's admins including Laura Thurlwell, Andrea Whitesell, Tami Fazio, and Alice Needham have made sure I always get time with him - a challenging task!

I would like to thank my family for their support over the years. First, to my grandparents, Dale and Virginia Read, for their unconditional love and support. I'm grateful for all the time we've had together and know I would not be who I am today without them. I would also like to express my gratitude to my parents, Trev and Laurie Sprabery, for fostering a love of electronics and science. Thank you for laying the foundation for this Ph.D. years ago with ham radios and electronics kits.

Last, but certainly not least, I would like to thank my wife, Brittany Sprabery. She has provided a tremendous amount of support and encouragement throughout the program. She is always the first-cut on my presentations and is a great editor for papers - a skill she probably never wanted! Thank you for being there during the paper reviews and presentation practices and research conversations. Thanks for sharing this journey with me!

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11- 2-0084. My work was also supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE1144245. The views and opinions expressed in this article are the author's own.

Table of Contents

List of Tables ix

List of Figures x

LIST OF ABBREVIATIONS xi

Chapter 1 Introduction 1
1.1 Capabilities 5

Chapter 2 Capabilities to Access Stateful Hardware 9
2.1 Cache Capabilities 11
2.2 Background 13
2.3 System Model 19
2.4 Design of Capability Enforcement Mechanism 22
2.5 Implementation 29
2.6 Performance Evaluation 42
2.7 Cache Capability Enforcement Summary 48

Chapter 3 Trustworthy Monitoring and Intrusion Detection 49
3.1 System Capabilities 49
3.2 Goals of a Hypervisor-Based Trusted Log 50
3.3 Background 51
3.4 Attack Model Against the Logging System 55
3.5 Trustworthy Log Acquisition 57
3.6 Logged Events 61
3.7 Intrusion Detection for Micro-Services 64
3.8 Evaluation 67
3.9 Related Work 73

Chapter 4 Intra-Application Capabilities for Micro-Services 75
4.1 Fine Grained Capabilities 75
4.2 Background 76
4.3 Threat Model 78
4.4 System Overview 79
4.5 Implementation 81
4.6 Example Provenance Graphs 93
4.7 Related Work 95
4.8 Conclusion 97

Chapter 5 Summary 98
5.1 Cross-Layer Take-Grant 98
5.2 Conclusion 100

References 101

List of Tables

1.1 Subject Security 6

2.1 Cache Capability Definitions 12
2.2 Cache Based Side Channel Attacks 14
2.3 Per-Domain Thread Allocations 30
2.4 Terms used in Scheduling Algorithm 32

3.1 System Capability Definitions 49

4.1 Fine-Grained Capabilities Definitions 76
4.2 Terms used in Provenance Algorithms 88

List of Figures

1.1 A Functional Model of Take-Grant 8

2.1 Defense Architecture 23
2.2 Co-Scheduling Overview 29
2.3 Limitations of Default Scheduling Policy 30
2.4 Limitations of Default Scheduling Policy + Flushing 31
2.5 Under-Utilization 33
2.6 Work Conserving 34
2.7 Worst Case Under-Utilization 34
2.8 Side-Effects of Work Conserving Properties 35
2.9 Limitations of Best-Effort Co-Scheduling Policy 36
2.10 Strict Co-Scheduling Protocol 38
2.11 Strict Co-Scheduling Example 39
2.12 Tenant Observable Loss in CPU Time 43
2.13 Scalability of Isolation Mechanisms 45
2.14 Full Sharing vs. Selective Sharing (Redis) 47
2.15 Full Sharing vs. Selective Sharing (Tomcat) 47

3.1 Invocation Process for a Specific System Call Handler 56
3.2 Timing Constraints for Interrupt Attack (**A2**) 56
3.3 Event Driven Probe Architecture 58
3.4 Induced EPT Signature & Probe Insertion 60
3.5 Trustworthy-Log Driven IDS Architecture 65
3.6 Apache Bench and OpenSSL Overhead 69
3.7 Redis Benchmark Overhead 69

4.1 Provenance Across Fork Events 90
4.2 Provenance Engine Memory Layout 92

LIST OF ABBREVIATIONS

ACRONYMS

ASLR Address Space Layout Randomization

CFG Control-Flow-Graph

CFI Control Flow Integrity

CFS Completely-Fair-Scheduler

COTS Commodity-off-the-Shelf

DEP Data Execution Prevention

DFG Data Flow Graph

EPT Extended Page Tables

HAV Hardware Assisted Virtualization

IaaS Infrastructure-as-a-Service

CAT Cache Allocation Technology

IDS Intrusion Detection System

IPC Inter-Process-Communication

ISR Interrupt Service Routine

JOP Jump Oriented Programming

KSM Kernel-Same-Page-Merging

LLC Last-Level-Cache

MSR Model Specific Register

PaaS Platform-as-a-Service

PCI Payment Card Industry

PoLP Principle of Least Privilege

ROP Return Oriented Programming

SCS Strict-Co-Scheduling

SMT Simultaneous Multithreading

SOA Service Oriented Architecture

TDP two-dimensional page tables

UFS Union File Systems

VA virtual appliance

VM Virtual Machine

VMI Virtual Machine Introspection

VMM Virtual Machine Monitor

WFQ Weighted-Fair-Queuing

Chapter 1: Introduction

Current implementations of the cloud computing paradigm leave users vulnerable to attack. Efforts have been made to make individual components more secure by patching vulnerabilities [1] or modifying cloud management frameworks [2]. Despite individual efforts to secure specific components, services deployed on cloud architectures remain exposed to attack. We argue this is because the PoLP is not being applied holistically to the entire stack. For cloud deployments to achieve their desired level of security, a capability based implementation of the PoLP must be applied at every layer of the stack.

Cloud computing emerged to reduce the costs of running services through the use of dynamic scaling and pay-per-use pricing models with per-hour billing, allowing a business or business units within a organization to treat computing resources as a utility [3]. NIST has defined *on-demand self-service*, *rapid elasticity* and *measured service* as 3 of 5 essential characteristics of cloud computing [4]. We highlight these characteristics because we believe they have played a major role in the adoption of cloud. These features make deploying scalable services easier and more cost effective than was previously possible. *On-demand self-service* allows a business or organizational unit to deploy new services without a lengthy equipment acquisition process. Combining this with *rapid elasticity* enables organizational units to deploy and scale their offerings without large amounts of management overhead. *Measured service* allows businesses to only pay for the time they use. This means that organizations can now iterate faster and with less risk without worrying about paying for the *possibility* of scaling. Organizational units will be charged more only when necessary, at a point when a service's value is likely to exceed the costs of scaling. The economics of scale afforded to shared-infrastructure cloud computing environments available today make offering such features a viable business [5]. These scalability advantages have driven the move to independently scalable components, or micro-services.

There are many shared infrastructure offerings for services targeting the public cloud,

private clouds, and hybrid clouds which leverage public clouds for scaling needs [4]. In this context, we use public/private/hybrid-cloud to mean Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) built on top [4, 6]. IaaS is most often used to deploy and scale Virtual Machines (VMs). Amazon's Elastic-Compute-Cloud [7] and Google's Compute Engine [8] are both examples of public clouds, though there are numerous more from various vendors [9–11]. More discussion on PaaS is in Section 2.2.4. In the context of a public cloud, the shared-infrastructure is multi-tenant. A VM from one organization may be co-scheduled on a single physical host as another organization's VM.

Private clouds initially seem to solve the security risks with multi-tenant public clouds by allowing an organization to host shared infrastructure for different business units company wide. Large companies can host their internal cloud utilizing a variety of private cloud stacks and make resources available to each business unit. OpenStack, co-developed by NASA and Rackspace, is one of the leading examples of software being used to deploy private clouds [12]. The same features are available in a private cloud: business units can spin-up virtual resources without a separate acquisition process, can dynamically scale-out, and fine grained measurements are still taken on "customer" resources usage. "Customer" here refers to a business unit within the larger organization. This allows a company to track the spending of various branches and possibly pass the costs through to different cost centers. In a private cloud, only a single company's VMs will run on the shared infrastructure. This does not eliminate the security risks associated with shared-infrastructure as multi-tenancy has been traded for multi-level clouds. Organizations divide their data and compute jobs based on risk and compliance needs. One business unit may be processing Payment Card Industry (PCI) data and thus need to meet or exceed PCI Data Security Standards [13]. Other organizations may be serving the needs of competing clients who may have services deployed on the shared-infrastructure of the firm. One could envision a situation in which one client tried to disrupt the service of another by leveraging the shared nature of the service-providing firm's private cloud. Government units need to provide various classification levels to data

and resources. The Department of Defense's Trusted Compute System Evaluation Criteria (TCSEC) (*i.e.,* the Orange Book) defines 4 such levels, each of which is further broken down into categories. When utilizing shared-infrastructures, even in the form of private-clouds, governments and organizations still need to be able to provide security isolation between services running on the infrastructure.

The rise of cloud computing has been accompanied by a growth in SOA's in the form of so called "micro-services" - pieces of software intended to provide the smallest granularity of functionality in the scope of a larger application goal. While no official definition of a micro-service exists, we will use the community definition popularized on the web [14] and from Sam Newman's book on building micro-services [15]. Micro-services are developed in a way that favors horizontal scaling and can be deployed independently of one-another regardless of whether or not they depend on data from another micro-service. It is common to deploy a micro-service oriented cloud application using containers or virtual appliances (VAs), VMs tuned for a single service. Micro-services are being used by small organizations and startups and large established companies such as Netflix [16]. The micro-service approach compliments a new trend toward programmatic operations units in a movement known as "DevOps". The goal is to make deployment of services as repeatable as possible by utilizing the same development and testing techniques used in software development. Micro-services often follow what is known as the "12-factor" approach to software development [17]: they tend to be side-effect free, rely on external configuration and scale horizontally. The DevOps movement, the ability to treat computing as a utility through shared-infrastructures, and micro-services are clearly here to stay and are driving the adoption of both VAs and containers to quickly deploy and scale isolated services. Security considerations must be made that take into account the challenges and advantages afforded applications developed using this new paradigm.

Computing on shared infrastructure poses numerous challenges to security. We separate these security challenges into two categories: 1) Those stemming from stateful hardware (*e.g.*,

side-channels) and those stemming from active attacks on micro-services (*e.g.*, an exploit for a particular vulnerability). The latter category can be divided into coarse grained attacks and fine grained attacks. Coarse grained attacks are those focused on execution of near-normal behavior, such as opening a configuration file containing application secrets. On the other hand, fine grained memory attacks allow Turing-complete execution within the compromised binary. We explore capabilities for PoLP enforcement at every layer of the stack by adding capabilities to access stateful hardware to mitigate passive attacks, adding more trust to relationship monitoring for coarse grained attacks, and by producing instruction to memory access capability lists for fine grained attacks.

For side-effect based attacks, we extend the concept of a capability for containers, a popular deployment target for many of today's micro-services, to include the shared cache used by a given service. Direct attacks can be detected using a variety of methods. We begin by exploring the limitations of existing dynamic probing mechanisms that can be used to probe relationship creation functions inside of a guest-kernel on top of which a micro-service is deployed. We add more trust to subject-relation monitoring systems. We then produce an Intrusion Detection System (IDS) that leverages the minimized nature of micro-services to enable manageable whitelists - enforcing capabilities based on observed behavior.

Despite being able to detect capability violations by leveraging probing mechanisms, there are still a large number of attacks that stay intra-process that will go undetected. Consider a micro-service that handles credit card data. Using a probing based approach, we could detect that the service was maliciously writing customer information to a file to be transferred to the attacker, a capability it was not granted. On the other hand, the attack would go undetected if the attacker siphoned customer information out of the service by embedding it into otherwise legitimate requests. This is more typical of what is known as an Advance Persistent Threats a type of malware that resides in a system over a long period of time. To combat these kinds of attacks, we build fine-grained capability lists through the use of a novel provenance engine for type unsafe code bases.

We argue that all of these issues can be addressed by utilizing the PoLP, but that it must be applied at every layer. In this dissertation we apply the PoLP at the hardware level, the systems level, and at the application level to mitigate both direct and indirect attacks through the use of capabilities. We enforce the PoLP to eliminate side channel attacks on the cache at the hardware level through the novel use of hardware enforced spacing separation and software enforced temporal separation of micro-services to provide a cache-access capability. We add additional trust for capability monitoring at the system level. The reduced behavior of micro-services is utilized to improve the manageability of whitelists to enforce the PoLP at the interface between applications and the operating system. This can stop certain attacker payloads and detect misconfigurations along with insider threats. The systems boundary is coarse-grained. Advanced exploits require that the PoLP be applied within the micro-service (intra-application). We build a provenance engine so that capabilities can be produced for memory regions within an application.

1.1 CAPABILITIES

Capabilities were first introduced by Dennis *et al.* [18] and further refined by Jones [19]. Jones discusses only giving subjects the minimal rights required to accomplish a given programmed goal - today we call this concept the Principle of Least Privilege (PoLP). In discussing capabilities, Jones explores the notion of taking and granting capabilities for subjects to access objects. Jones *et al.* go on to formalize this subject-object relationship in "A linear time algoirhtm for deciding security" [20]. This model of security was refined by reducing the entity types from subject-object's to subjects by Lipton *et al.* [21] in what became known as the Take-Grant model of security. Lipton *et al.* explore the notion of relationships between subjects as highlighted by Table 1.1. Bishop *et al.* extended the notion of Take-Grant to capture capabilities on *de facto* information transfer [22]. More recently, the Take-Grant model of security has been used to formally verify the seL4 micro-kernel, demonstrating the

scalability and applicability of the approach to production quality code bases [23]. Recently, there has been efforts to transition commodity operating systems to include stricter notions of capabilities through sand-boxing techniques [24]. Instead of focusing on just capabilities at the operating systems level, we focus on a cross-section of capabilities spanning the entire stack leveraged in cloud-deployment of micro-services. We have modernized the notation used in [21] and denote the group of subjects accessed by subject α as $\mathbb{S}_\alpha$. Note that our usage of the word "accessed" is imprecise - the term in this context simply means there exists a relationship between two subjects. The nature of that relationship is dependent on the exact application at hand. It may mean reading a file, writing a file, executing a binary, accessing a given memory region, memory page, or other operating system construct. In a web application, "subjects" may be application level users and they may have relationships with other application level constructs - all of which may be represented in an application database. In this dissertation, we explore extending the concept of "subject" to simultaneously include stateful hardware, coarse grained system actions, and fine grained memory accesses within an application.

Table 1.1: Subject Security

	Definition of Term
$\mathbb{S}_\alpha$	Subject Domain for process α
$\mathcal{R}$	A relation between two subjects
$\alpha\ \mathcal{R}\ \beta$	Process α is related to β through relation $\mathcal{R}$

To further clarify our usage of the term "subject-domain" consider two subjects α and β related to one another through relation $\mathcal{R}$ as shown in Table 1.1. Afterwards, $\alpha \in \mathbb{S}_\beta$ and $\beta \in \mathbb{S}_\alpha$. Capabilities are built around relationships; for example, a subject may have the capability to use R to modify its security domain $\mathbb{S}$.

We leverage a functional model of micro-services that treats the service as a function whose execution produces side-effects on the function's subject-domain. We model a micro-service as a function $g(x,\ y)$ on program arguments x and the environment in which it executes, y.

The side-effects on the subject-domain are relationships $\mathcal{R}_1,\ \mathcal{R}_2,\ ...,\ \mathcal{R}_n$. Capabilities are applied at the relationship level. To address passive attacks, we extend the notion of a subject to include cache regions and grant the capability to access a given region to a single security domain at a time. Mechanisms to monitor relationships stemming from system actions for VAs already exist - we explore making these monitors more trustworthy. To conclude, we address the limitations in coarse-grained subject-relationship monitoring by extending the definition of subjects to include individual memory instructions and the memory regions they access to produce fine grained capabilities lists to be applied within $g(x,\ y)$. Figure 1.1 highlights our model. Relationships are made as a side-effect of execution of the micro-service, leading to changes on the micro-service's subject-domain. Capabilities are security policies on these changes. If there exists a transition from a trusted subject-domain to one containing subjects that violate a policy or information flow, then the transition should not be allowed. For fine-grained attacks, we want to identify the memory regions that can impact $\mathbb{S}_g$, identify the instructions that should modify those regions, and then enforce memory access capabilities guaranteeing those are the only instructions modifying those regions.

To enforce security in cloud computing environments, we must consider stateful subjects within the hardware (cache lines), add additional trust to relationship monitoring, and finally, explore tracking of memory that impacts the micro-service's relationship with subjects in the system. Having formally added stateful hardware and the relationship with it to our security model, we can look at mechanisms within the operating system to add capabilities to the relationship between micro-services and shared caches. The system call boundary is the primary means a service has for relationships creation between subjects, and is a logical monitoring point from within a Virtual Machine Monitor (VMM). Finally, the arguments to these system calls are dependent on the capability of individual memory modification instructions to modify memory being passed into system calls that impact a given service's security domain, and are the target of modern fine-grained attacks [25, 26]. Building capa-

bilities lists at the instruction granularity can address these attacks.

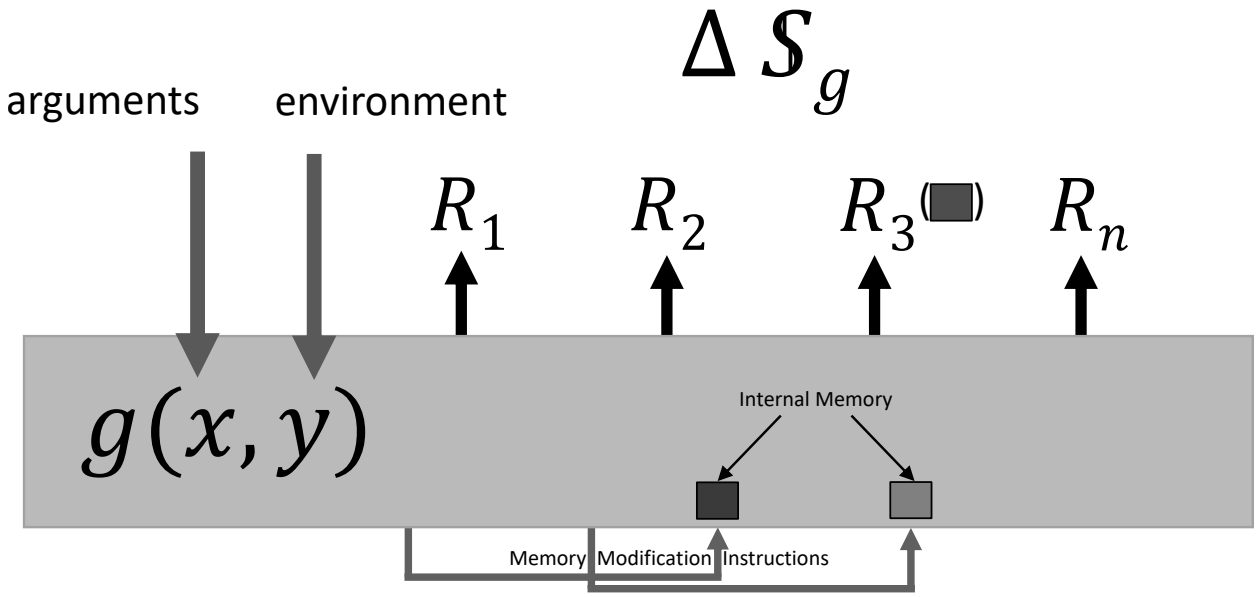


Figure 1.1: A Function Model of Take-Grant: Micro-Service $g(x,\ y)$ changing its subject domain $\mathbb{S}_g$ as a side effect of execution through relationships $\mathcal{R}_1 - \mathcal{R}_n$. Some relationships are functions of internal state that is modified by fine-grained "subjects" (instructions) within the service.

Chapter 2: Capabilities to Access Stateful Hardware

Cache-based side-channel attacks (*e*.g., [27–29]) are a threat to computing environments where a diverse set of users share hardware resources and are a leading example of how stateful hardware can be leveraged for passive information flow. Such attacks take advantage of observable hardware side-effects due to the execution of software programs. A number of these attacks focus on differences in timing while accessing shared processor caches. Recently, researchers have adapted these cache-based side-channel attacks to cloud computing environments, especially IaaS clouds [30–33], and showed that secrets and sensitive information can be extracted across co-located VMs. Container frameworks such as Docker [34] are even more susceptible to such attacks since they share the underlying operating system kernel [35].

Initial cache-based side-channel attacks focused on gaming schedulers at the OS and VMM layers [28, 29, 31]. Such approaches focused on resource sharing of L1 and L2 caches within a single processor core via Simultaneous Multithreading (SMT) [27]. Multicore processors introduce cache-based side-channels via the Last-Level-Cache (LLC), thus making defenses much harder [33, 35].

Many defenses against cache-side-channel attacks in cloud environments have been proposed [36–50]. Existing solutions are insufficient in the following ways. Shannon's noisy-channel coding theorem states that information can be transmitted regardless of the amount of noise on the channel [51]. While probabilistic defenses (*e*.g., [36–38]) may decrease the bit-rate of attacks, they cannot fully eliminate them. Defenses that *eliminate* such attacks, rather than frustrate techniques employed by the attacker, are more desirable. SMT must be disabled for some solutions [38], impacting performance and utilization. In addition to a guaranteed defense, the solution must not severely impact *(i)* the performance of the applications or *(ii)* utilization of the machine. Defenses must minimize the performance impact of enforcing hard isolation to remain practical. Disabling hyperthreading (*e*.g., SMT) can

have a significant impact on machine utilization. To the best of our knowledge, every cloud provider enables hyperthreading. Costly application rewrites may be required for other defenses [39–41].

Solutions must be *easy to adopt*. History has shown that solutions requiring additional development time (or significant changes to existing applications) are less likely to be adopted (as shown in the Return Oriented Programming (ROP) community [52]). Thus, solutions that require developers make application level changes [40, 41] may be challenging to apply to existing workloads. Hardware based approaches are plagued by similar problems – they are difficult to deploy as they require vendor support and fabrication of new chips [42, 43]. Violating x86 semantics by modifying the resolution, accuracy, or availability of timing instructions can frustrate attacks, but consequently require changes to all applications running on the machine [44–46]. Global compiler and page-coloring cache-partitioning [47–50, 53] transformations introduce high overhead. JIT techniques allow for local optimization, but performance remains problematic [54].

In this chapter, *we present a hardware-software framework to add capabilities for cache-access, mitigating side-channel attacks in cloud computing systems that use multicore processors with a shared LLC*. The proposed framework uses a combination of *Commodity-off-the-Shelf (COTS) hardware features* along with *novel scheduling techniques* to enforce a cache-access capability, defending against cache-based side-channel attacks. In particular, we use Cache Allocation Technology (CAT) [55] which allows us to *partition last-level caches at runtime*. CAT, coupled with *state cleansing* during capability transfer between security domains and selective sharing of common libraries removes the source cache-timing-based side-channel attacks between different domains. We implement a novel scheduling method as an extension to the commonly-used Completely-Fair-Scheduler (CFS) in Linux to reduce the overheads inherent due to any such cleansing operation and to enforce our cache-access capability by limiting execution to only those processes holding the capability to access the

a given region. Our solution provides a *transparent*[1] way to eliminate cache-side-channel attacks while still working with *hyperthreading enabled* (SMT) systems. It works with containers, kernel-based virtual machines (vCPUs), and any other schedulable entity that relies on the OS scheduler[2]. To the best of our knowledge, this work is the first to provide transparent protection of applications without disabling hyperthreading.

In summary, we make the following combined contributions via a capability based approach that:

C1 Can eliminate cache-based side-channel attacks for schedulable units

C2 Allows providers to exploit hyperthreading,

C3 Requires no application level changes, and

C4 Imposes modest performance overheads

2.1 CACHE CAPABILITIES

Consider groups of processes belonging to a given security domain. These processes may be threads in the same micro-service or many distinct micro-services being run by the same parent entity or security domain. Each process has a capability list of resources it can access. These terms are outlined in Table 2.1. Operating systems today have the ability to grant individual schedulable entities the capability to access a given memory region m_r through the virtual memory subsystem. Operating system schedulers also have the ability to grant a process the capability to access a limited set of CPU resources r_c. We want to extend this notion of capability to access a given hardware resource to include cache regions, c_l.

[1]From the perspective of the application developer/user.

[2]For ease of exposition, in the rest of the chapter, we will describe our framework using containers.

Table 2.1: Cache Capability Definitions

	Definition of Term
$\mathbb{C}_{\mathcal{P}}$	Set of Capabilities for process $\mathcal{P}$
$\mathbb{P}$	Set of processes in an isolated region
$\mathbb{P}_A$	Set of processes in security domain A
m_r	Memory Resource Capability for memory r
r_c	CPU Resource Capability for CPU c
c_l	Capability to access cache region l
$\mathbb{D}$	Set containing all security domains running on the system

Let $\mathbb{P} = \mathbb{P}_A \cap \mathbb{P}_B$, the union of processes from two security domains. Note that in Table 2.1, we introduce the notion of an "isolated region". This is necessary for cache-access capability enforcement and is expounded upon in Section 2.4. Consider subject (process) $\alpha \in \mathbb{P}_A$ and process $\beta \in \mathbb{P}_B$. Traditional capability mechanisms available in operating systems allow for the creation of capability lists of the form: $\mathbb{C}_\alpha = \{m_\alpha,\ r_\alpha\}$ and $\mathbb{C}_\beta = \{m_\beta,\ r_\beta\}$. In this chapter, we extend these capability lists to include the capability to access a given cache region, c_l, that can only be held by a single security domain at a time. The final capability list for a process β could then become: $\mathbb{C}_\beta = \{m_\beta,\ r_\beta,\ c_l\}$. For capability enforcement, we want to ensure that if a process within a security domain holds the capability to access a given cache region, then no other process in any other security domain can hold the same capability. Formally, $\forall\ \lambda \in \mathbb{P}_\Lambda;\ \forall\ s \in \mathbb{D} - \Lambda,\ \forall\ \delta \in \mathbb{P}_s$ then $c_l \notin \mathbb{C}_\lambda \cap \mathbb{C}_\delta$. The mechanisms presented in this chapter enforce this invariant while allowing for capability transfer between security domains to ensure the machine can still be shared between domains while balancing the performance impact. We use the word "transfer" to mean revocation by the operating system followed by a subsequent grant of the cache-access capability to another domain.

The capability approach presented here is the only defense that can defend against Prime&Probe cache-based side channel attacks without modifications to the tenant application while also enabling hyperthreading for the machine at hand. Every cloud provider today leaves hyperthreading enabled to drive up machine utilization. We evaluate multiple workloads under varying conditions to explore viability of a strict cache-access capability enforcement mechanism. Our changes are implemented on top of the Linux kernel.

2.2 BACKGROUND

In this section, we highlight existing attack mechanisms and defenses. We then review the cloud systems we target due to current deployment trends. Finally, we provide background on the scheduling mechanisms we extend for cache-capability enforcement.

2.2.1 Cache Side Channels

Cache side channel attacks take advantage of the shared nature of processor resources, in particular the processor L1, L2, and Last-Level-Cache (LLC). Prime&Probe attacks were first explored across VMs by Osvik *et al.* [27] and shown to be practical in cross-core attacks via the LLC by Liu *et al.* [33]. Modern clouds are driving up machine utilization by offering container based platforms. This both drives revenue [5] while providing more performance to tenants [56, 57]. Cache side channel attacks are already emerging on container based infrastructure [35]. Flush+Reload attacks [32] are a real threat to cloud computing security and have been successfully deployed on public infrastructure [35]. We outline the attack types in Table 2.2. All of the attacks fall under the broader class of *access based* side-channels in which an attacker can tell whether or not a given resource has been accessed by a victim in a given time period.

Table 2.2: Cache Based Side Channel Attacks

Cache Level	Attack Type	Placement
L1 & L2	Prime&Probe	Single Physical Core
L1 & L2	Flush+Reload	Single Physical Core
L1 & L2	Prime&Probe	Single Physical Core, SMT
L1 & L2	Flush+Reload	Single Physical Core, SMT
L3/LLC	Prime&Probe	Cross-Core
L3/LLC	Flush+Reload	Cross-Core

Prime&Probe Attacks: Originally designed in [27], Prime&Probe attacks derive their namesake from a two phase approach of priming a cache line and then probing it after a short period of time. The attack begins with the attacker priming a cache line and then measuring the time to access the same line to establish a base line for the probe phase. The attacker will then probe the same cache line and measure the access time. If the access time is higher than the established base line, then it means that a victim process also accessed the cache line causing the attacker's data to be evicted. This probe phase also acts as a subsequent prime. This measurement loop is repeated until the attacker has sufficient data on the cache access patterns of a victim process. It is important to note that this technique is highly dependent on victim code base and requires a cache access pattern be established for the software package being run by a targeted victim. Methods such as Cache Template Attacks [58] can be used to generated these patterns. This attack has been demonstrated on L1 and L2 on a single physical core via hyperthreading [27]. Neve *et al.* show a very similar attack that can be carried out on a single core without relying on hyperthreading by taking advantage of the operating system scheduler [28]. Ristenpart *et al.* demonstrate that a variant of Prime&Probe can be used to determine cache usage on a host VMM [30] while Zhang *et al.* demonstrated that the Prime&Probe method could be used to extract private keys across VMs [31]. It is important to note that the authors had to add extra

layers of processing to reduce the additional noise and scheduling impacts of the VMM. When working with containers, these complications are eliminated, making the attack easier to carry out. It was initially thought that simply scheduling processes on different cores might eliminate the security risk because it would be too difficult to carry out the attack on the LLC. Liu *et al.* were the first to show that the technique could be successfully executed on the LLC [33]; their work was carried out in a VM environment. Again with this attack, it will be easier to carry out in container environments as the noise and overheads associated with the VMM are removed.

Flush+Reload Attacks: Flush+Reload attacks rely on shared memory between two executing processes and were first used in [29] to derive an AES key from a victim process, though not under the Flush+Reload name. The attack utilizes similar probing to the Prime&Probe attack to determine if a given cache line was accessed with the requirement that memory must be shared between the attacker and victim. The nature of shared memory allows an attacker to determine which branch within a shared library was taken by a victim process. Gullasch *et al.* show that `clflush` can be used on x86 platforms to flush cache lines of libraries shared between processes on the same host on a single core [29] and then probes can be used to produce an access based side-channel. While they do not show the attack with hyperthreads, the single physical core case will be very similar. Flush+Reload works by first flushing a piece of memory using `clflush` and then measuring the time to reload the flushed memory location. If the reload phase takes the same amount of time as a previously established baseline measurement, then the victim process accessed the same piece of memory. The Flush+Reload term was first used by Yarom *et al.* to attack RSA when the authors extended the technique to multi-core, leveraging the LLC. The authors tuned the attack to work with multiple VMs [59] sharing pages due to memory de-duplication processes running within the VMMs. Shortly afterwards, Irazoqui *et al.* use Flush+Reload to successfully attack AES in a cross-VM scenario with the attack completing in under a minute [32]. Yarom *et al.* has also used Flush+Reload to attack the Elliptic Curve Digital Signature

Algorithm in OpenSSL on a single quad-core host [60]. This more closely represents the container-like deployment in which processes all run on the same host kernel. With the rise of containers and interest in PaaS clouds, focus has shifted to practical deployments of the attack. Zhang *et al.* showed a Flush+Reload attack deployed on the public PaaS DotCloud which uses container technology as the underlying deployment mechanism [35]. In container deployments, shared memory stems from the fact that image base layers are shared and are loaded from the same location on disk implying that they use the same page-cache. This is similar to the way shared libraries work in a traditional computing environment.

2.2.2 Existing Solutions

Some solutions are probabilistic [36–38] which is insufficient as they cannot fully eliminate the source of cache-based side-channel attacks. Existing approaches achieve single core isolation by disabling hyperthreading [38]. Disabling hyperthreading reduces the throughput of not only tenants requiring extra isolation, but the entire machine. Such approaches are untenable as the economic model behind cloud computing dictates high per machine utilization [5]. Cutting whole-machine utilization by even 20%, the impact of hyperthreading in 2005 [61], is too high for cloud computing. We argue that while for some tenants, a 20% overhead may be a reasonable trade-off for increased security, there is little value in forcing all tenants to pay that performance penalty. StealthMem is able to enable hyperthreading, but the authors do not sufficiently address cross thread scheduling issues [40] and require application developers to make code changes. CATalyst [41] follows a similar defense model to StealthMem but uses Intel's CAT technology to assign virtual pages to sensitive variables instead of software-based page coloring. Application rewrites are necessary for these approaches [39–41], increasing the cost of adoption. Costly hardware changes [42, 43] face similar adoption challenges. CHERI [53] takes a similar capability based approach to cache-access, but at this time is not available to cloud providers in COTS systems as it requires custom hardware. Some hardware approaches change timing instruction behavior which

changes the semantics of the architecture [44–46], again forcing application changes. Cache coloring [47–49] approaches have impractically high overheads.

CACHEBAR [38] defends against Flush+Reload attacks by duplicating memory pages on access from separate processes, a scheme it calls Copy-On-Read. Since Kernel-Same-Page-Merging (KSM) does de-duplication of pages at regular intervals, it also modifies the behavior of KSM to achieve Copy-On-Read. To defend against Prime&Probe attacks, CACHEBAR modifies the memory allocation based on which memory regions are cacheable so that the attacking process loses visibility into the victim process. This technique provides only probabilistic guarantees for defense against a Prime&Probe attack while also disabling hyper-threading. Other solutions that rely on the scheduler [62] focus on uniprocessor systems, thus do not have to face the challenges faced by caches shared by multiple cores. The solution presented by Godrey *et al.* [63] suggests scheduling patterns for VMs can decrease the cost of cache-flushing. Their solution only works in hypervisors. Our work is more generic in that it applies to any schedulable entity, be it a container or a vCPU supporting a KVM VM. Other ideas in [63] are similar: they use cache coloring techniques while we use hardware cache partitioning, but they rely on infrequent context switches between vCPUs and do not content with the need for a form of "strict" co-scheduling.

Finally, solutions like Nomad [37], while probabilistic, complement our approach. Nomad works in the cloud scheduler to reduce the co-residency of different security domains. Our solution could be used in conjunction with Nomad to provide hard isolation when co-residency is not possible.

2.2.3 Intel's Cache Allocation Technology

We utilize Intel's CAT to isolate a portion of the cache as a "secure partition" so that a capability to access this isolated region can be produced. CAT allows a processor's LLC to be partitioned into 4 segments on our test machine (this number will likely increase in future generations). The partitions are configured using a bitmask that is written into an Model

Specific Register (MSR) on Intel's Broadwell and later CPUs. CAT provides isolation for a given cache by not-allowing evictions on that partition from cores belonging to another partition. It is important to note that cross-partition cache hits are still allowed even with CAT enabled. A configured cache partition can be used by a select number of cores. Each core has a core specific MSR dictating which "class-of-service" (the Intel nomenclature for cache partitions) it is running in. When discussing an isolated region, we are referring to both the cache partition and the set of CPUs assigned to that cache partition. The size of the cache partition can be reduced to limit the cost of flushing the cache during capability transfer. The reduced cache size has little impact on cloud based workloads [64]. Using this feature alone as a defense would dictate that the capability to access a secure partition be limited to a single core so that cross-thread, cross-domain Prime&Probe attacks would be prevented, but would not prevent Flush+Reload attacks.

2.2.4 Platform-as-a-Service

Containers have been popularized by Docker [65] and have better performance when compared to VMs [56, 57]. Containers are the main building block for PaaS as they allow for fast, repeatable, and scalable deployments of the web-serving workloads that PaaS target. Ristenpart *et al.* show that it is possible to maliciously collocate tenant workloads on VM based cloud infrastructure [30]. Ristenpart's collocation approach has been extended for container based infrastructure in [35] for usage in deploying a side channel attack on DotCloud, a popular PaaS. Containers run as isolated processes running on top of the same kernel, allowing for many more tenants per provider-machine. Original cache based side channel attacks [27, 66] focused on the cross-process single-kernel environment - containers exhibit the same properties. Single kernel side-channels attacks are well understood, thus the risk to container based deployment is higher than for VMs. Existing VM based defenses do not have to deal with the scale of the per-process isolation required to defend against side-channel attacks in PaaS.

2.2.5 Scheduling

Because of the popularity of Docker [65] on Linux, we limit our discussion to Linux. Our capability approach remains generic. Scheduling in Linux is motivated by Weighted-Fair-Queuing (WFQ) [67] originally developed for improving the Quality of Service of packet networks. Early attempts at scheduling in Linux focused on reducing the algorithmic complexity of the scheduler's decision making process. This lead to the so-called $O(1)$ scheduler, the namesake of which is also the algorithmic complexity of the algorithm. The $O(1)$ scheduler performed poorly on I/O bound workloads on desktop systems as any latency sensitive workload, like those suitable for scale-out cloud deployment, had to wait for proceeding processes to exhaust their time-slice. To improve responsiveness, the community moved to the WFQ-motivated CFS algorithm. This brought with it the introduction of per-thread *vruntime*'s which track per-thread weighted time on the processor. Over a given stable "epoch", a time frame for which no threads are entering or leaving the system, CFS can guarantee fairness and ensure that I/O bound processes are not starved. In our implementation, we leverage a co-scheduling algorithm to ensure proper capability transfer between domains. The algorithm is implemented within the Linux CFS scheduler, but only operates on cores which have been designated as being isolated. A detailed explanation of how this is done in Section 2.4. For a complete discussion on scheduling in Linux, we refer the reader to Love's book [68].

2.3 SYSTEM MODEL

In this section, we outline the hardware requirements of our systems. We describe our specific machine used for testing and evaluation. Finally, we introduce the threat model which drives the design decisions behind cache-access capabilities.

2.3.1 Cloud Environments and Commodity Hardware

We assume cloud workloads like those presented in [64] are being run. We consider public Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) cloud environments. Such environments allow for co-residency of multiple computational appliances such as containers or VMs belonging to potentially different security domains. We assume that the cloud computing infrastructure is built using commodity-off-the-shelf (COTS) components. In particular, we assume that the servers have multi-core processors with multiple levels of caches, some of which are shared. We also assume that the servers have a runtime mechanism to partition the last-level shared cache as described in Section 2.2.3.

In this chapter, we use an Intel Haswell series processor that has a three-level cache hierarchy: private level 1 (L1) and level 2 (L2) caches for each core (64KB and 256KB respectively) and a last level (L3) cache (20MB) that is shared among all the cores. For cache partitioning, we turned to Intel's *Cache Allocation Technology* (CAT) [55] that allows us to partition the shared L3 cache. The CAT mechanism is configured using MSRs. This can be achieved at runtime dynamically using software mechanisms. On this specific processor model the maximum number of partitions is limited to *four* but newer generations support more [69]. Intel CAT technology has been available on select Haswell series processors since late 2014 and continues to be available on select processor lines belonging to the Broadwell and Skylake micro-architectures that succeeded the Haswell micro-architecture.

While our implementation and evaluation uses an Intel Haswell processor with Intel CAT technology, the proposed approach is generally applicable to any multi-core processor with a hierarchical cache, shared last-level cache, and a mechanism to partition the shared last-level cache.

2.3.2 Adversary Model

We assume that the attacker can easily collocate a container next to a victim workload as shown in literature [30, 35] with the goal of carrying out a cache-based side-channel attack. We assume that container base layers are shared across tenants by default, introducing the risk of Flush+Reload attacks. The attacker can sleep, allocate memory using memory allocation system calls and otherwise try to game the scheduler or memory allocation policies.

We consider both cross-core (*i.e.,* attacker and victim running on different cores on the same processor) and same-core (*i.e.,* attacker and victim running on the same core) side-channel attacks. Same-core attacks, as the name indicates, require the attacker achieve co-residency on the same core and be able to preempt the victim. They typically focus on higher-level caches (L1 and L2) that are specific to the core. Cross-core attacks on the other hand only require the attacker achieve co-residency on the same physical server. If multi-socket systems are used, we assume the attacker can achieve co-residency on the same socket as the victim. Their limitation is that they only allow the attacker to observe the victim's activities through the LLC which is shared by all cores on the chip and thus is noisy. As discussed in Section 2.2.1, it has been shown that such limitations can be overcome [33].

We assume the attacker is capable of achieving co-residency with a victim, can allocate an arbitrary number of resources, and game both the cloud level scheduler (placement) and the operating system level scheduler (preemption). However, we assume that the cloud infrastructure provider is trusted. That is, while the cloud scheduler may be gamed, we assume that both the attacker and victim are authenticated with the cloud provider (e.g., for billing purposes). Additionally, we assume that the host kernel, running either KVM or container frameworks, is trusted.

2.4 DESIGN OF CAPABILITY ENFORCEMENT MECHANISM

Our solution contains cross domain contamination through strict cache-access capability enforcement. We implemented cache-access capabilities via software based flushing, hardware based cache isolation, and per-domain duplication of container base layers. Flushing costs are reduced through the novel use of hardware cache size reduction and software based temporal partitioning through the use of a co-scheduling capability enforcement algorithm. The solution allows schedulable entities to be grouped into security domains and ensures there is no leakage across domains. Unlike existing approaches, we can enable hyperthreading while also being completely transparent to guest applications. There is no need to manually rebuild an application to indicate the parts most vulnerable to side channels as was done in [41]. Additionally, there is no implied performance impact to tenants with less strict security requirements outside of a reduced cache size.

Our Flush+Reload defense has little impact on CPU utilization, but will increase disk and memory usage. Existing approaches for Flush+Reload such as the one produced by Zhou *et al.* [38] rely on dynamic memory duplication. This not only drives up memory usage, but may impact the performance of micro-services by making the shared memory portion of the kernel more complex.

Intuitively, flushing the cache during a context switch between entities belonging to different security domains defends against any attack that uses cache based timing to derive information from L1 and L2 caches. Flushing alone will not defend against LLC attacks as the LLC is shared across cores, thus per-core cache partitions must be created. "Secure" workloads then must be limited to a single core on either a software based partition (cache-coloring) [40, 49] or hardware based cache partitioning, which until recently has not been available on COTS hardware.

Our framework logically partitions a host server into an *isolated region*[3] and a *shared region* as illustrated in Figure 2.1. Tenants are required to indicate to the cloud provider whether or

[3]It can be easily extended to multiple isolated regions.

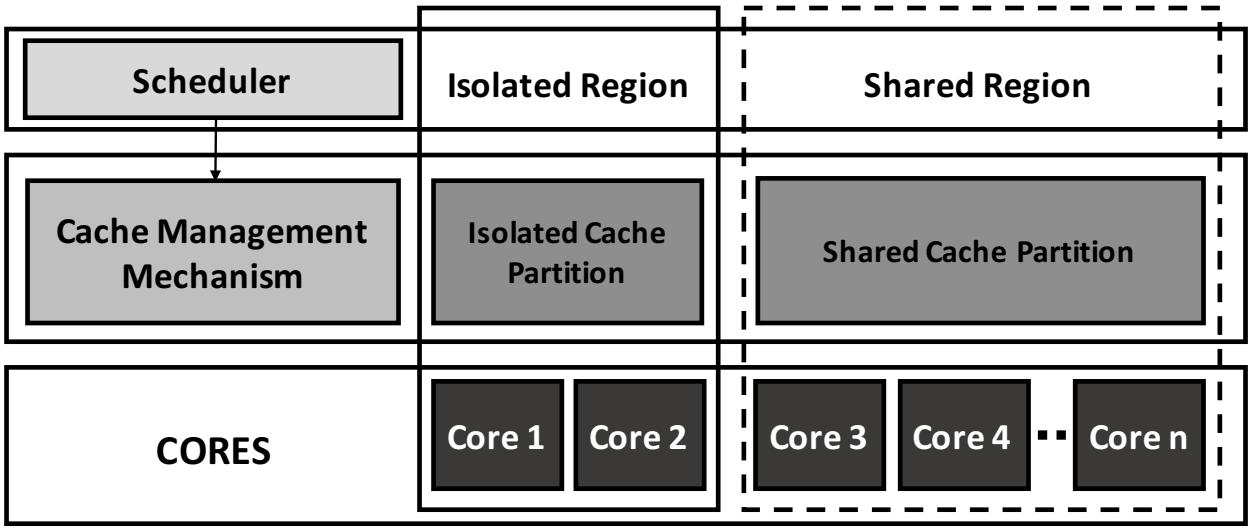


Figure 2.1: Defense Architecture

not their containers need isolated execution. Entities that require isolated execution will be executed while their parent domain holds the capability to access a separate cache partition on the host server; all other entities will be executed with a shared cache partition. The "isolated execution" designation guarantees that processes within the designated containers will not share cache resources with *(i)* processes from any container belonging to another security domain or with *(ii)* processes belonging to any container that is not designated for isolated execution irrespective of their security domain. Consider the cloud deployment of a web service using a micro-service based architecture. We envision a system in which the tenant indicates that the load-balancer (usually an HTTP reverse proxy) should run in the isolated region. The load-balancer is the micro-service often responsible for encrypting connections and thus may contain sensitive functions and data which could be the target of a cache-based side-channel attack.

Our design leverages *(i)* Intel CAT, processor affinities, and selective page sharing to provide *spatial isolation* and *(ii)* co-scheduling with state cleansing to provide *temporal isolation* for designated containers, granting the capability to access the isolated cache region to a single security domain at a time.

2.4.1 Hardware-Assisted Capability Enforcement for Spatial Isolation

Intel's CAT [55], currently available in COTS hardware in Intel's Xeon series processors, is designed to improve the performance of latency sensitive workloads by allowing the LLC to be partitioned into distinct regions. Each processor core is thus restricted to allocating cache lines into a specified cache partition. Consequently, a processor can only evict cache lines within its own assigned LLC partition, thus reducing the impact of processes running on other cache regions and vice versa. In particular, note that the ability to allocate cache lines, priming in Prime&Probe attacks, in a cache shared with the victim and the ability to evict cache lines being used by the victim process, flushing in Flush+Reload attacks, are key steps in cache-based side-channel attacks. Ensuring potential victim and attacker processes run on cores associated with different LLC cache partitions defeats many cache-side-channel attacks. Specifically, cross-core Prime&Probe attacks on a victim process are eliminated between the isolated and shared regions since we use different cache partitions. An additional source of side-channels, shared memory, is discussed in Section 2.4.2.

Cores in the system are associated with partitions such that each core is assigned to one and only one LLC partition. We refer to these partition-core combinations as *isolated* and *shared regions*. The maximum number of cache partitions available with Intel CAT is fixed for a given micro-architecture. The machine used for our testing allows for up to 4 distinct partitions, but newer machines have 16. The configuration of size, number of active partitions, and core to partition assignment occurs in software and can be adjusted based on the demand for isolated execution and the needs of the expected workloads. If there is no demand for shared execution then the host server could be partitioned into two (or more) isolated partitions. In this chapter, we evaluate a single isolated region and a single shared region, though the technique works equally well with multiple isolated regions. The cache-access capability is a capability granted to a domain to access a single isolated region.

As previously discussed, hardware-assisted spatial partitioning protects the containers running in the isolated partition against cross-core Prime&Probe style attacks from contain-

ers running in the shared partition. However, cross-core side-channel attacks across cache partitions are not entirely eliminated. Intel CAT, primarily designed to improve fairness of cache sharing and performance of latency sensitive workloads, allows cache hits across partition boundaries to maximize the benefits of shared memory such as the `libc` shared library. In particular, if the victim and the attacker processes share memory, because of layered file systems used in container frameworks for example, an attacker can carry out a Flush+Reload attack. Since the attacker previously flushed the cache lines, a cache hit indicates that the victim executing in a different core (and LLC partition) has used or is using the library. While CAT limits the granularity of information an attacker can glean across partition boundaries, timing observations are still possible and hence the side-channel is not entirely eliminated while using CAT. Furthermore, attacks within an isolated partition continue to be viable. These will be addressed in the following sections.

The partial protection against cache-side-channel attacks obtained through spatial partitioning comes at the cost of reduced LLC cache size and the associated potential reduction in performance. Fortunately, reduction in cache size has been shown to have relatively little impact on modern cloud workloads [64]. In particular, minimal performance sensitivity to LLC size has been reported for cache sizes above $4-6$MB with modern scale-out and server workloads (see Section 4.3 and Figure 4 in [64]) that are typical in cloud environments.

2.4.2 Selective Page Sharing

To ensure the cache-access capability is enforced properly, we must identify all sources of cross-domain cache-access to enforce the invariant described in Section 2.1. As previously discussed, hardware-assisted spatial partitioning does not eliminate cross-core Flush+Reload style attacks when the attacker and the victim share memory pages. Modern container deployments have one primary source of shared memory, causing cache-lines to be shared across domains. We limit our discussion to Docker as it is one of the most popular choices for building container images and running them on Linux platforms, but these concepts also

extend to other container frameworks. Docker uses storage drivers that are built on top of Union File Systems (UFS) so that a process inside of a container can access a file system composed of a stack of layers. Several different containers may use the same versions of libraries and other base components, thus a way was needed to reduce disk and memory usage of the common building blocks. Docker addressed the need to share base components by uniquely identifying each layer by its cryptographic hash and sharing common layers between all containers built using a given layer ID.

Often there are multiple containers running the same image which causes them to share every layer except for the upper-most writable layer. For example, two Apache Tomcat servers running on the same Docker host using the same image would share all binaries including the Java Virtual Machine (JVM), Apache Tomcat, GnuPG, and OpenSSL among others. Only the top most layer, containing writable elements such as the Tomcat log file, differ between containers.

To enforce the cache-access capability in container deployments, we eliminate cross-domain page sharing through selective layer duplication. That is, for containers requiring isolated execution, our system allows sharing of layers only among containers belonging to the same security domain but not otherwise. This is a reasonable trade-off as it enables isolation between different tenants while limiting the increase in memory usage. In particular, the increase in memory usage will be a function of the number of tenants running on the server rather than the number of containers. We do not prevent traditional sharing of layers for containers running in a shared partition.

For VMs, the kernel same-page merging (KSM) module in Linux, used for memory de-duplication, is the main source of shared pages. However, KSM and memory de-duplication in general come with their own security risks (*e.g.*, [70–73]). For instance it has been shown that KSM can be leveraged to break ASLR [70], enable Rowhammer [74] attacks across VMs [72], and create a timing side-channel that can be used to detect the existence of software across VMs [73], much like the Flush+Reload style attack discussed previously.

Given the serious security concerns surrounding the use of KSM, we leave it disabled. Same page merging is disabled by default on commercial products from VMWare as well [75].

Note that selective page sharing, combined with hardware-assisted spatial partitioning, eliminates cross-core cache-side-channel attacks *across partitions* by allowing cache-access capabilities enforcement at the cache-partition boundary. Selective page sharing removes the ability for an attacker to measure interference after flushing a given address and CAT partitioning removes the ability for an attacker to prime a victim's cache across partition boundaries. Cache-side-channel attacks from within an isolated partition due to multi-core and SMT continue to be a threat and will be discussed next. A way is needed to transfer the capability to access a given cache-partition to another security domain to allow an isolated region to be shared without leaking information.

2.4.3 State Cleansing During Capability Transfer

Even with containers running in isolated partitions, an attacker allocated to the same isolated partition as the victim might be able to *(i)* observe the victim's LLC usage if scheduled to run on a different core than the victim but associated with the same partition and *(ii)* even observe the victim's L1 and L2 usage if running on the same physical core as the victim [31]. In the latter case an attacker observes the cache usage of the victim by managing to frequently alternate execution with the victim process, or via SMT.

To thwart these attacks *we propose to cleanse the cache state when transferring the cache-access capability between schedulable entities belonging to different security domains.* That is, if a process from one security domain, SD_1, runs on a core, then processes belonging to another domain, SD_2, must either run on a core assigned to a separate partition or state-cleansing of the shared caches must be performed on the partition during the transfer of the cache-access capability from SD_1 from SD_2. There currently exists no hardware instruction for per-partition cache invalidation. More detail on how state cleansing can be achieved is in Section 2.5. However, state cleansing alone does not prevent attacks from an attacker

process that is running in parallel with the victim either on the same-core through SMT, or running on a different core but in the same partition.

A naïve solution for capability transfer within a single isolated region would be to assign a single core to the isolated partition, disable hyper-threading and perform state-cleansing on every context-switch. The performance cost of such an approach is unattractive. A mitigation would be to create multiple isolated partitions with a single-core assigned to each one. However, the number of cache partitions is finite, 4 in our case, and such an approach would further fragment the LLC and hamper performance for the shared partition. Furthermore, many cloud workloads are multi-threaded and leverage additional cores when available. Thus, a mechanism is needed to enforce cache-access capabilities while assigning multiple logical-cores to an isolated region allowing for more scalability within an isolated region.

2.4.4 Co-scheduling for Temporal Isolation

To address the aforementioned threat, we use a *novel scheduling technique for temporal separation of security domains*. Co-scheduling container processes belonging to a given security domain across multiple processors amortizes the cost of state cleansing during capability transfer, but introduces additional complexity as discussed below.

Scale-out workloads with many threads, those commonly deployed on cloud infrastructure, motivate this approach. As thread counts for a security domain increase, the number of threads able to run per domain at any given time will be high. This allows us to drive up utilization of cores assigned to a partition and *only flush the partition when transferring the cache-access capability to the next domain*. The complexities stem from needing to synchronize all isolated cores during capability transfer, thus any implementation of co-scheduling has to guarantee an exclusion property. No task belonging to security domain SD_X can run on an isolated processor while a task from another domain, SD_Y, is running in a processor associated with the same isolated partition to enforce the invariant in Section 2.1.

Additionally, before SD_X can be granted the cache-access capability, a state cleansing event must occur. As shown in Figure 2.2, multiple cores can be utilized at once within a security domain. However, state-cleansing must be performed as every core assigned to a given partition context switches to the security domain for which the cache-access capability has been transferred. The next security domain cannot run on any isolated processor until this process is complete.

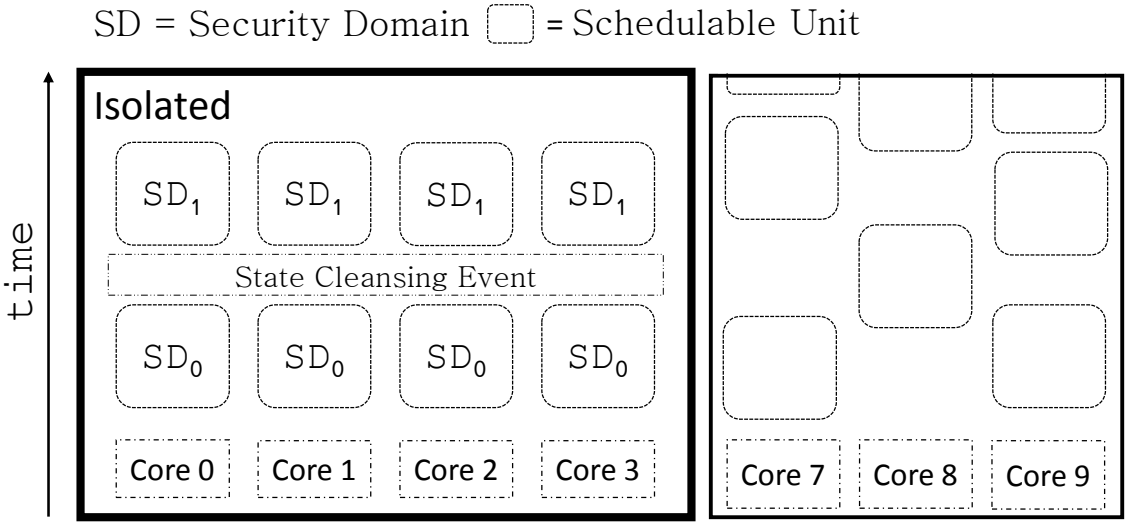


Figure 2.2: Co-scheduling Overview - The isolated environment is on the left. It consists of an isolated cache partition along with the processors assigned to that partition. Co-scheduling is used to group tasks belonging to the same security domain and state cleansing events occur when changing domains. Regular tasks are on the right in a separate cache partition. Tasks on the right, those in the shared region, have no scheduling restrictions.

2.5 IMPLEMENTATION

Partitioning the LLC and associating cores with each partition does not require changes to the kernel or the operating system. It can be done by a system administrator as part of the machine configuration. Here we focus on the implementation of our co-scheduling and selective-sharing mechanisms.

2.5.1 Capability Enforcement through Strict Co-Scheduling

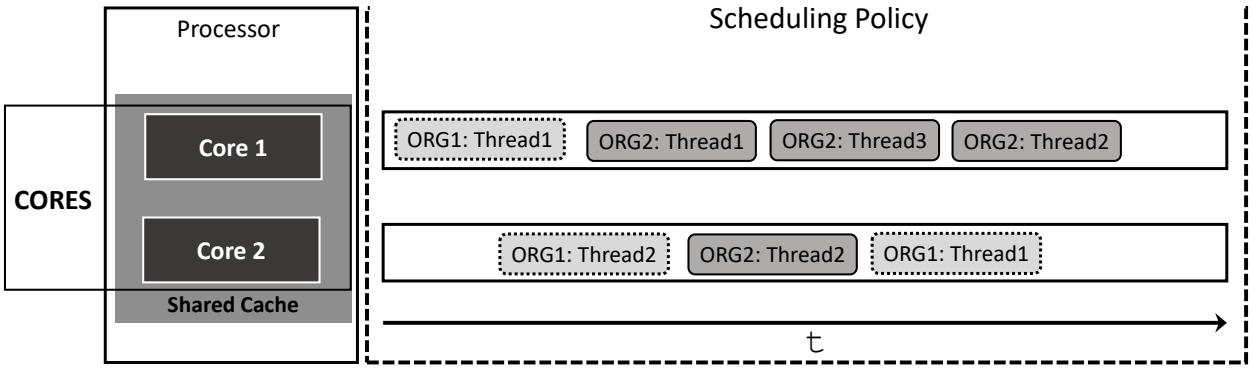


Figure 2.3: Limitations of Default Scheduling Policy

Default Scheduler: Consider containers from two security domains or organizations with thread configurations as shown in Table 2.3 running on two cores. Simply executing on the cores requires allowing the processes to access the L1, and L2 caches belonging to those cores along with the entire LLC. The defining characteristic in our example is the shared cache. For hyperthreaded cores, this is the L1, L2 *and* LLC. In the case of two physical cores, the shared cache is only the LLC. The cores 1 and 2 in Figure 2.3 *are not* cores on two separate sockets on the same motherboard. Figure 2.3 shows an example schedule that might result from the default scheduler in Linux.

Table 2.3: Per-Domain Thread Allocations

Domain ID	Thread Count
ORG1	2
ORG2	3

Even if the scheduler flushes between scheduling different containers, the other organization has the ability to carry out a cache-base side-channel attack. This process is shown in Figure 2.4. Consider the flushing events f_1 on $Core_1$ and f_2 and f_3 on $Core_2$ as shown in Figure 2.4. Despite the flushing event f_1, attacks can be carried out across containers belonging to different domains during Δt_1. This limitation is repeated at flushing event f_3 for a period of Δt_2. It is clear that enabling hyperthreading or assigning multiple cores

to a single isolated region poses an additional set of challenges to cache-access capability enforcement.

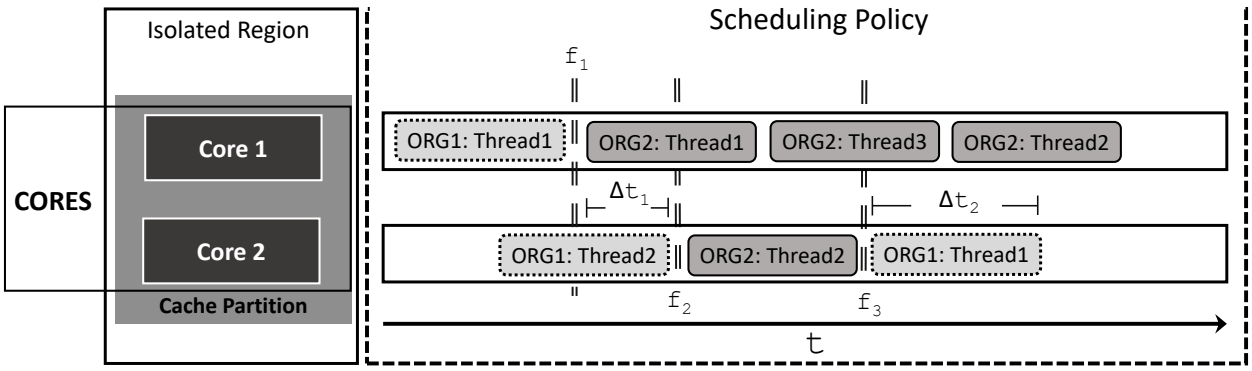


Figure 2.4: Limitations of Default Scheduling Policy + Flushing

Traditional CFS in Linux was born out of the need to reduce the impact to latency sensitive jobs. We introduce a Strict-Co-Scheduling (SCS) algorithm for cache-access capability enforcement. Our SCS implementation aims to reduce the cost of transferring the capability to access a given cache region due to flushing while remaining favorable to latency sensitive tasks by utilizing CFS within a security domain.

The terms outlined in Table 2.4 are used to describe the SCS algorithm we introduce. Like the default Linux scheduler, if no work is available the processor will idle. There are two main changes to the default scheduler class that are not shown here. When choosing a process to run, the scheduler will always choose from $P_{PrivilegedDomain}$. Additionally, the default scheduler will not schedule a process for less than $MinRuntime$. To ensure our SCS algorithm remains as work-conserving as possible, a thread may be preempted after only running for a fraction of the $MinRuntime$ where necessary. More discussion on this below.

Algorithm 2.1 highlights how the SCS *nextDomain* function is implemented in Linux. Once the next next domain is chosen, cores schedule threads only from $P_{PrivilegedDomain}$ or idle until work is available. This approach introduces little additional algorithm complexity. Because the $O(1)$ scheduler in Linux, which is round-robin, performs worse for latency sensitive tasks, we would expect this to remain the case when using round-robin as a basis

for SCS. We mitigate this impact by only transferring the cache-access capability in a round robin fashion, while utilizing the CFS scheduler when scheduling processes from the domain holding the cache-access capability.

Table 2.4: Terms used in Scheduling Algorithm

	Definition of Term
$SDCList$	A circularly linked list of security domains.
i	The index offset in to $SDCList$.
P	The queue of *runnable* processes in the system sorted from least to greatest `vruntime`
P_{DOMAIN}	The queue of *runnable* processes belonging to $DOMAIN$ sorted from least to greatest `vruntime`.
$PrivilegedDomain$	The security domain currently holding the cache-access capability.
$MinRuntime$	The minimum runtime for which a thread should be scheduled.

Algorithm 2.1 Strict-Co-Scheduling (SCS) Domain Selection

```
function NEXTDOMAIN
    i ← i + 1
    i ← i mod SDCList.size()
    domain ← SDCList[i]
    while size(P) > 0 AND size(P_domain) == 0 do
        i ← i + 1
        i ← i mod SDCList.size()
    end while

    domain ← SDCList[i]
    return domain
end function
```

The downside with SCS stems from the system's inability to remain work-conserving under certain situations. In a work-conserving system, the processor never idles if there

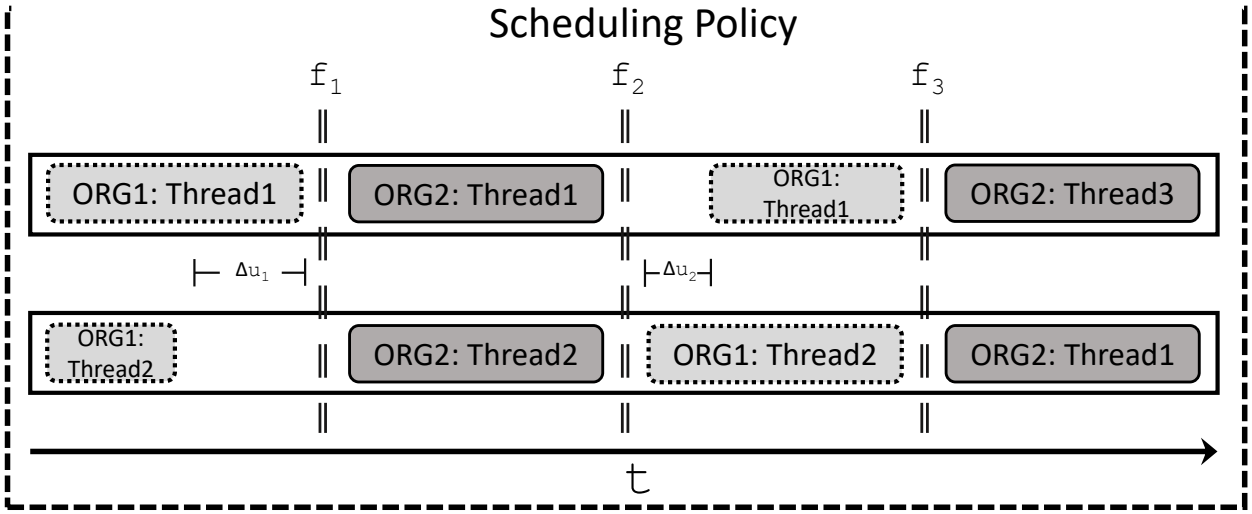


Figure 2.5: Under-Utilization

is work that can be done by *any* process in the system. By very definition, we can not consider all processes during the isolated run of a single group of tasks belonging to the same security domain. We highlight three different cases which demonstrate an inability to remain work-conserving using the configuration as outline by Table 2.3.

Figure 2.5 shows the situation in which $ORG1 : THREAD2$ finishes before the *PrivilegedDomain*'s ($ORG1$) *MinRuntime* is up, leading to a Δu_1 underutilized time span The system must idle until the next capability transfer occurs, assuming work is available for the next domain. Once $ORG1$ is scheduled again, $ORG1 : THREAD1$ may be waiting on input for a period of Δu_2. Again, the system will remain underutilized for a period of time and $ORG1 : THREAD1$ will not receive the full *MinRuntime*.

In order to mitigate situations of low utilization such as the one presented in Figure 2.5, we choose to violate the *MinRuntime* guarantee provided to processes in Linux. Consider the situations in which the number of processes for the *PrivilegedDomain* exceeds the number of cores, as is the case for $ORG2$. As shown in Figure 2.6, we can schedule $ORG2 : THREAD3$ if $ORG2 : THREAD2$ does not use its full *MinRuntime*. This means that $ORG2 : THREAD2$ will not be scheduled for the full *MinRuntime*, potentially introducing more context switching.

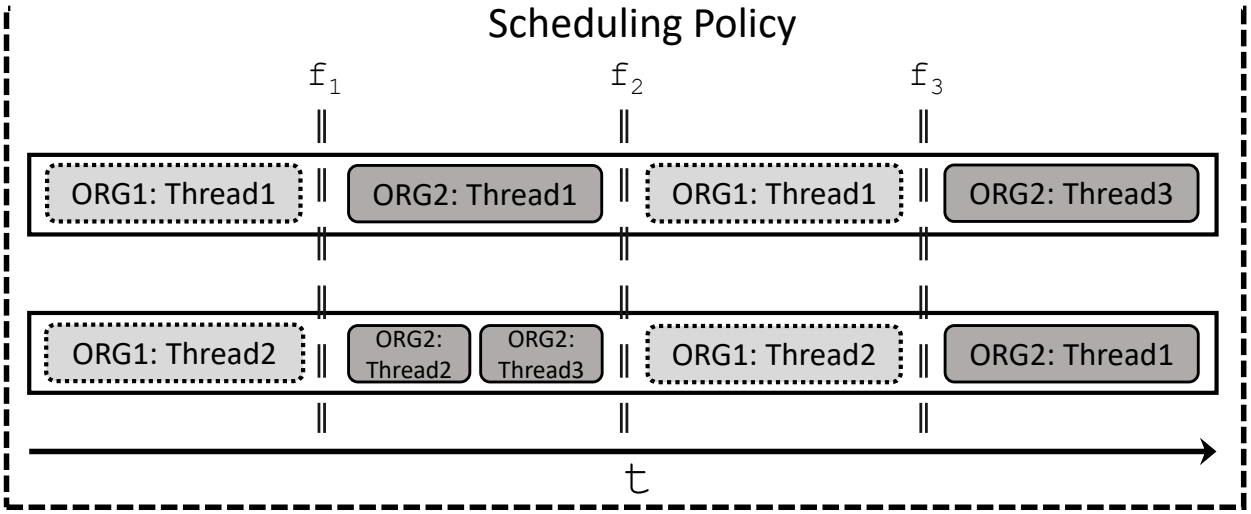


Figure 2.6: Work Conserving

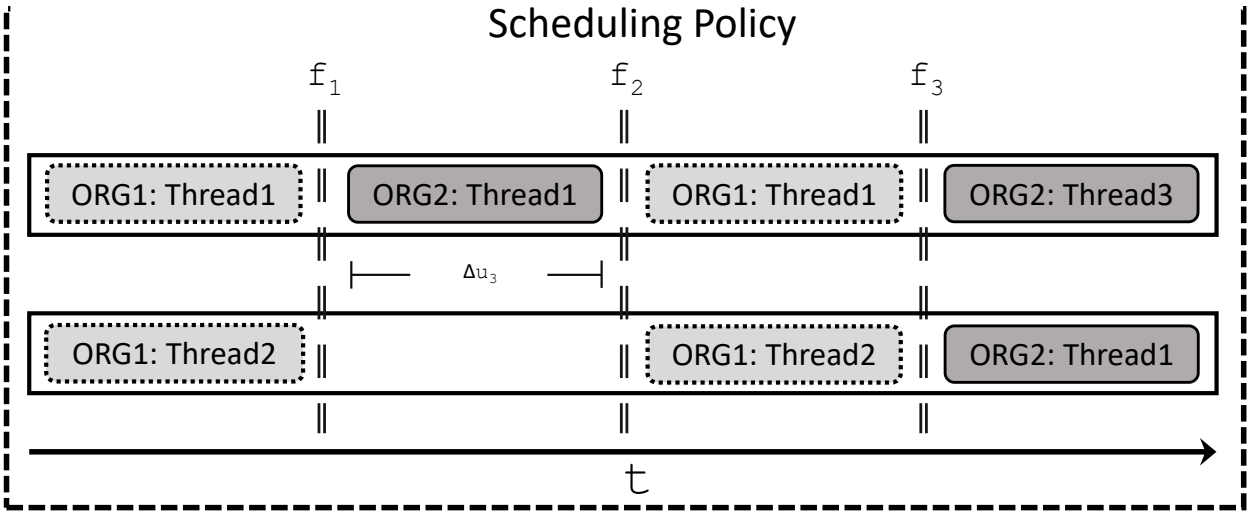


Figure 2.7: Worst Case Under-Utilization

Figure 2.7 demonstrates a worst case scenario upon granting $ORG2$ the cache-access capability. If only one thread can do work, then one core will be underutilized for Δu_4 which will be equal to the full $MinRuntime$. The performance of interactive workloads with long inter-arrival times of requests could lead to situations like this. Note that this is the worst case because if there are no runnable tasks in a domain, we will not grant the domain the cache-access capability.

The goal of violating the $MinRuntime$ guarantee afforded to processes in Linux is to drive up utilization. This can lead to unintended consequences depending on workload. Consider

the case presented in Figure 2.8. Our approach is to revoke $ORG1$'s cache-access capability and then grant it to $ORG2$ if both threads for $ORG1$ finish before the $MinRuntime$ is reached. Upon switching to $ORG2$, consider an example in which work is only available for a single thread. Partway through the execution of $ORG2 : THREAD1$ work may become available for $ORG2 : THREAD2$, so it is scheduled. This processes might be repeated for $ORG2 : THREAD3$, leading to three distinct times of under-utilization, Δu_6, Δu_7, Δu_8, the sum of which is guaranteed to be strictly less than $MinRuntime$. This situation should be rare in practice as long as the mean processing time for I/O bound processes is greater than the $MinRuntime$, but can come into play when multiple threads have been scheduled for many $MinRuntime$'s and are nearing the end their current workload. If this case were to occur frequently in practice it may drive up the amount of time spent underutilized and for which transferring the cache-access capability to another security domain might have been more appropriate. This can be minimized if the proper number of threads are chosen for a given workload. Correct thread allocation or dynamic optimization of thread count is a separate research problem.

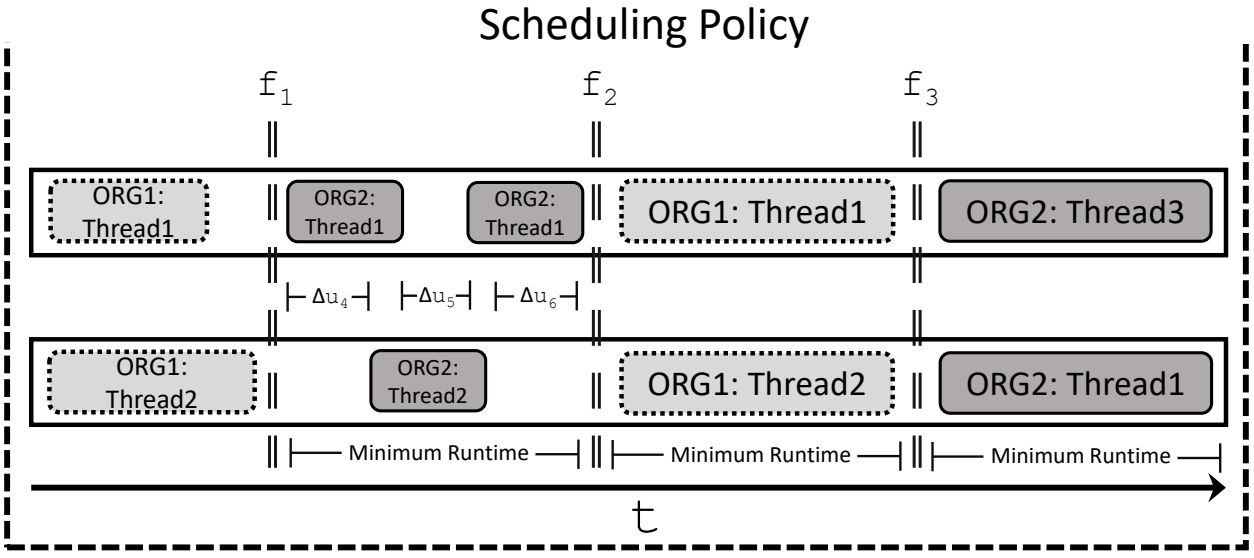


Figure 2.8: Side-Effects of Work Conserving Properties

2.5.2 Linux Kernel Modifications

Co-scheduling *can* enforce proper capability transfer between security domains, but any implementation must be precise. By precise, we mean that any form of "loose" or "lazy" co-scheduling is unacceptable. Co-Scheduling attempts in Linux have been focused on VMs. Existing patches are "best-effort" co-scheduling [76]. These "soft"-co-schedulers try to find a thread to run that belongs to the same group of processes as the thread just scheduled. If no thread is found, there is no guarantee that one from another security-domain will not be scheduled. This leads to, at best, situations like the one presented in Figure 2.9.

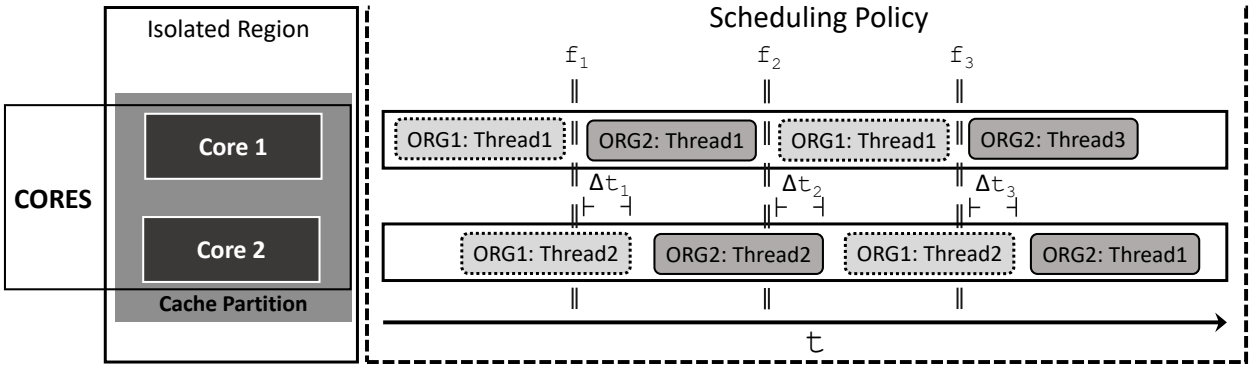


Figure 2.9: Limitations of Best-Effort Co-Scheduling Policy

Figure 2.9 is a schedule instance of the configuration as described in Table 2.3. The example is a situation in which 2 cores are associated with an isolated partition and are running containers belonging to two security domains. These cores may be two physical cores or one physical core presented as two to the operating system as is the case with SMT. The defining characteristic in our example is the shared cache. For hyperthreaded cores, this is the L1, L2, *and* LLC. In the case of two physical cores, the shared cache is only the LLC. The cores 1 and 2 in our example are cores on the same socket.

Consider a situation in which $Core_1$ initiates capability transfer before scheduling a thread from a conflicting domain, `ORG2:THREAD1` in this example. Even if the scheduler invokes a flushing event, f_1, there remains a Δt_1 during which cross-core, cross-domain attacks could be carried out. This is seen again after $Core_1$ schedules `ORG1:THREAD1` and `ORG2:THREAD3`

leading to durations Δt_2 and Δt_3 during which attacks remain feasible. While this situation is still better than the one presented in Figure 2.3 it highlights the need to pay particular attention to the implementation subtleties when implementing co-scheduling of processes in the same security domain on modern operating systems. Effective elimination of side-channels dictates that cross-core synchronization be performed before state cleansing occurs and subsequent domain scheduling takes place.

Our SCS algorithm is implemented by modifying the default CFS scheduler in Linux. The default Linux scheduler performs time accounting on a per group basis through the usage of layers of red-black trees in place of a traditional run-queue [77]. At the lowest layer, there is a red-black tree consisting of all the tasks in the system. The cgroup layer is directly above the layer of `task_structs`. A `task_group` is a Linux kernel data structure used to group "schedulable entities." These entities may be other `task_group`'s or individual `task_struct`'s representing a single thread. One node at this layer would hold all the `task_structs` for a single container for example. Above that is a layer of `parent-cgroups` - a layer that only exists if the user specifically creates a cgroup holding other cgroups. In our system, one such node is created by the Docker daemon anytime a new `parent-cgroup` option is used. A node at this layer contains per-CPU run queues that can be traversed through the child-cgroup layer to reach all the tasks belonging to a given domain.

The per-CPU data structures associated with nodes at each layer are stored in the `task_group` struct which also has a `siblings` field, a linked list pointing to other `task_group`'s at the same level. We hold a pointer to the head of this list (which is not associated with any `task_group`) and maintain a separate pointer to the `PRIVILEGED_DOMAIN`. Transferring the cache-access capability can be performed by simply setting the `PRIVILEGED_DOMAIN` pointer to the next item in the list that is not the head and pointing to a `task_group` with runnable tasks. This makes up the *nextDomain* function as outlined by Algorithm 2.1.

To ensure that no two schedulable units belonging to different security domains run on an isolated cache partition simultaneously, we implement the core synchronization protocol

shown in Figure 2.10. The protocol works by making the first core in an isolated partition a *leader* core. The leader core is responsible for revoking the cache-access capability, synchronizing cores, state cleansing via a cache flush, and finally, granting the cache-access capability to another security domain. All isolated cores only schedule tasks belonging to the `PRIVILEGED_DOMAIN`, a global variable pointing to the `task_group` for the security domain holding the cache-access capability. We work at the third level of `task_group`'s such that all containers belonging to a security domain are contained within the `task_group` pointed to by `PRIVILEGED_DOMAIN`. Note that while only 2 cores are shown in Figure 2.10, the approach works with any number of cores. In Section 3.8 we evaluate the protocol with 4 cores assigned to an isolated partition.

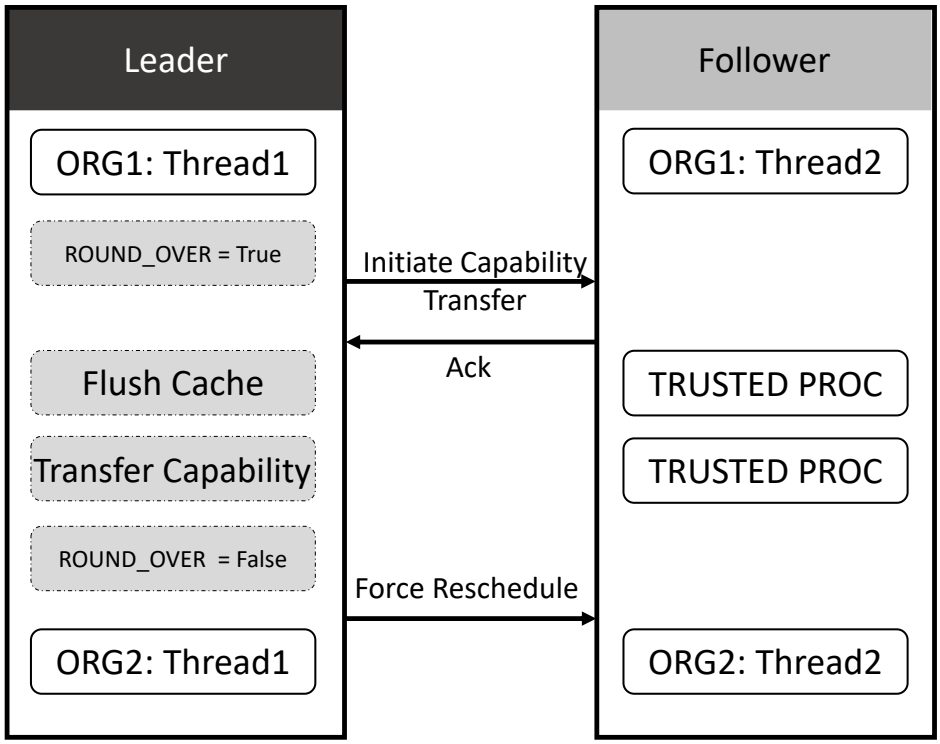


Figure 2.10: Strict Co-Scheduling Protocol

To allow multiple cores to be assigned to the same cache partition, we use a follow-the-leader approach to scheduling. A leader core is in charge of initializing capability transfers and flushing the cache between revoking the capability and granting it to another domain.

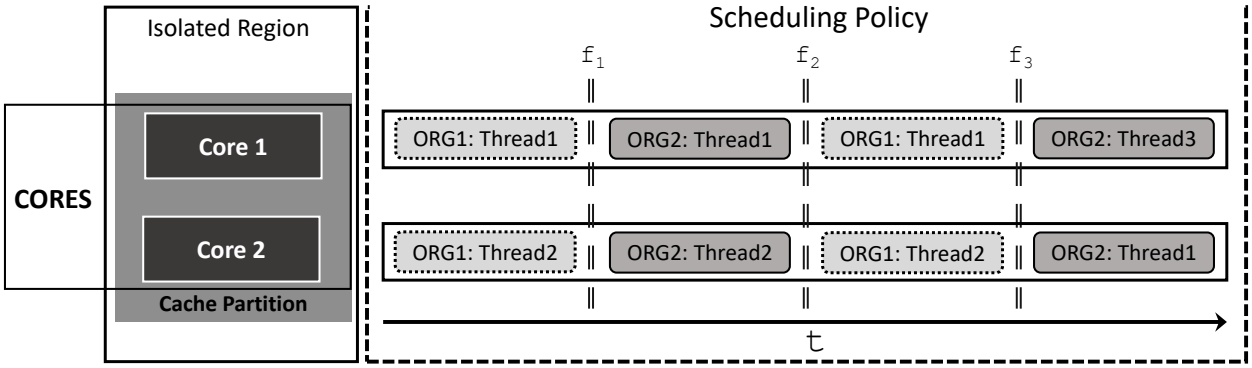


Figure 2.11: Strict Co-Scheduling Example

This algorithm is invoked under two circumstances. The first is the result of a timer that runs every $MinRuntime$ milliseconds. The timer checks to see if work is available in another domain and, if so, grants the capability to access this cache region to the next domain with available work. If no other domain can be run, this process simply reschedules the timer interrupt. On the other hand, if another domain is to run, the leader invokes a synchronization routine in which each of the follower cores schedule an idle process and send an acknowledgment to the leader core. Upon receiving acknowledgments from all follower cores, the leader will flush the partition and force the follower cores to re-run their scheduler functions in order to schedule tasks from domain now holding the cache-access capability.

Isolated cores rely on two pieces of shared state to achieve strict synchronization. The leader core is the only core that can modify state. The `ROUND_OVER` variable indicates to follower cores that a capability transfer is about to occur. A timer on the leader core initiates a domain change by modifying this variable and invoking the `__schedule` function on the leader core. The `capability transfer` event fires every `sysctl_sched_min_granularity`, a configurable parameter exposed to administrators on Linux based systems to control system responsiveness. This is the variable exposed to allow administrators to control $MinRuntime$. After setting the `ROUND_OVER` variable to `true`, the leader core issues a reschedule command via an Inter-Processor Interrupt to follower cores and waits for them to send back an acknowledgment. The acknowledgment is performed within the `__schedule` function on fol-

lower cores. When the `ROUND_OVER` variable is set, partitioned cores can only run *trusted processes*. These are only kernel tasks, including: `ksoftirq`, `watchdog`, and the idle task. Ensuring such processes can run prevents deadlocks due to `watchdog` timeouts.

After receiving an acknowledgment back from all follower cores, indicating they are no longer running tasks belonging to any security domain, the leader then flushes the cache and updates the `PRIVILEGED_DOMAIN` to point to the next security domain using Algorithm 2.1. Our system uses a separate `task_group` within the Linux kernel for each security domain. Run-queue checking is performed to ensure a domain with runnable tasks is chosen.

Having chosen the next domain, the leader core sets `ROUND_OVER` to `false` and again issues a reschedule command to follower cores. The `__schedule` function will eventually be invoked on the follower cores, but we use the reschedule command to reduce the idle time of follower cores. This protocol corrects the problem presented in Figure 2.9 resulting in "strict" co-scheduling as seen in Figure 2.11.

As previously mentioned, to ensure that less idling occurs, we will initiate a capability transfer if the `PRIVILEGED_DOMAIN` has no more runnable tasks, as shown by the first run of $ORG1$ tasks in Figure 2.8. This is accomplished by having a `cpu_mask`, the bits of which represent which cores in the partition are idling. If a core schedules an idle process, it sets the corresponding bit to 1. That core then checks to see if all the other cores in the partition are also idle. If so, it forces the timer described above to run on the leader core. This could be extended further by making the transfer occur once a certain percentage of cores in the partition are idle. For now, we focus on the capability-enforcement mechanism and leave policy optimizations up to future work.

It is clear that this approach will impose overhead due to the synchronization costs between cores assigned to a given partition. It is important to remember that these costs are only paid by containers for which increased isolation is required. Regular containers will not pay additional overheads and can be co-located on the same machines as containers utilizing the above scheduling policies without incurring any penalty as we show in our evaluation

section. Despite the overhead, we feel that the trade-off is worth it for critical applications.

2.5.3 State Cleansing

The isolated cache partition must be cleaned or flushed before switching context to a different security domain. For a processor cache, state cleansing or flushing equates to invalidating the cache lines or evicting them, but no hardware mechanism exists to flush the cache lines assigned to a single CAT partition. The *WBINVD* instruction invalidates the entire shared cache, disrupting processes in all partitions.

One way to implement state cleansing is for the user process to invoke the *CLFLUSH* instruction, which can evict cache lines corresponding to a linear virtual address and can be invoked from user space. This can be done by the application process before being switched out as in [78]. However, this requires changes to user applications which is not desirable. Another possibility is for the kernel to invoke *CLFLUSH* on the entire virtual address space. While this is guaranteed to work across processor generations, this approach is too costly, taking up to 10x `sysctl_sched_min_granularity` on applications we tested. An optimization is to do it only on valid virtual addresses for the task being switched out as was done in [79]. However, this can still be a large range compared to the size of a cache partition ($4 - 6$MB).

Another approach is to create an *eviction set* – a set of addresses which when loaded are guaranteed to evict the entire cache partition. However, the memory-address to cache-line mapping is proprietary and subject to change across processor generations. Cache-side-channel attacks also have to contend with this challenge and have addressed it by reverse-engineering the memory address to cache line mapping for a given micro-architecture [33]. Apart from the one-time cost of reverse engineering, the cost of this approach is equal to loading memory the size of the cache partition from a linear address space. To evaluate the performance of such an approach, we use the memory load method.

We perform this state cleansing anytime the security domain is changed, as shown by the

protocol in Figure 2.10 and the co-scheduling overview in Figure 2.2. To reduce performance impact in the case of other domains lacking runnable threads (due to blocking on I/O, *etc.*), flushing is only performed when `PRIVILEGED_DOMAIN` changes. If only a single security domain has runnable tasks, no flushing will occur.

2.5.4 Selective Page Sharing

Docker uses a UFS to present a unified view of the several different layers. Of the several UFS that Docker supports like btrfs [80], overlayfs [81], and AUFS [82], AUFS is mature and supports all of Docker's storage feature set. In a UFS, multiple directories on the host are unified in a single directory called a *union mount*, without replicating the actual contents of individual layers. Contents of all the layers become visible at the *union mount*. Docker keeps a single copy of each layer on the host file-system and AUFS mounts all the layers to a single *union mount* point, which becomes the container's root file system. Each layer can be a part of multiple *union mounts* and thus can be shared across different containers.

Our implementation modifies Docker[4], specifically the AUFS storage driver, to transparently allow selective sharing of file system layers. We modified the AUFS driver to have separate copies of each layer for each security domain. In this way, no two containers belonging to different security domains share any common layers.

2.6 PERFORMANCE EVALUATION

In this section, we present an evaluation of the performance impact of our core-synchronization algorithm used during capability transfer. We evaluate overheads in terms of machine-utilization loss as observed by the cloud provider. We then explore the impact as observed by cloud tenants. We also explore the memory growth impacts of our selective-sharing mechanism.

[4]v1.14.0-dev, compiled from source code available on Github

2.6.1 Impact of Scheduler Changes

Our prototype implementation is evaluated using a CPU bound workload to determine the impact on applications in a worst case scenario. Consider a batch workload such as Hadoop or a web serving workload. The case in which all threads have work and are not waiting for input is approximated.

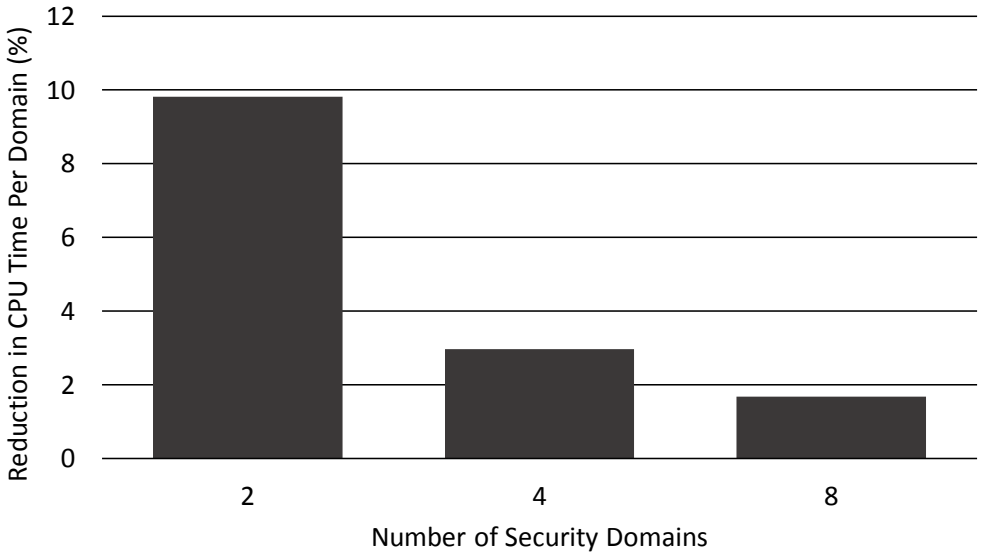


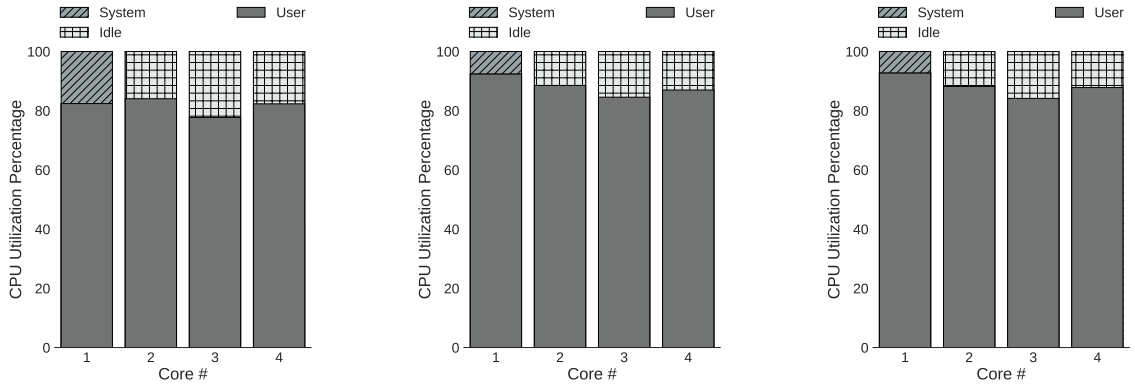
Figure 2.12: Tenant observable loss in CPU time allocated to each security domain as a function of the number of security domains. Note this performance degradation only occurs in the isolated region. The performance impact when there is only a single security domain is 0, thus is not shown.

The machine is configured as outlined in Section 2.3. We allocate 2 physical cores and 4 logical processors to an isolated cache region. The cache region is 4MB (4 cache ways out of 20 available on the system). Each security domain is assigned 4 threads, and the number of domains is varied from 2 to 8. Each domain consists of 4 CPU-bound tasks, 1 for each logical processor. Measurements are taken using `sar` and `pidstat` at an interval of once per second for 100 seconds. Figure 2.12 shows the overhead normalized to security domain for the 4 security domain case. The reduction in time spent in userspace per domain is small. Figure 2.13 shows the overhead of our system running while varying the number of security domains assigned to an isolated region. System utilization when our system is disabled is

very near 100% in each case, thus is not shown. It is clear to see that the overheads for a single logical processor are a function of the system and not of the number of security domains assigned to a partition. Follower cores can be seen idling during domain changes, but the overheads never exceed slightly above 20%, with the average case being slightly below 20% on follower processors. From our tests, we know that flushing significantly increases the performance penalties. The leader core spends the most time executing in system space due to its responsibility to manage capability transfer and synchronize cores, so this was to be expected. In the future, we will investigate mechanisms to reduce system time on the leader core and idle time on follower cores. Hardware based per-partition flushing mechanisms, such as an enhanced `WBINVD`, would significantly reduce these overheads, though our approach would still be needed to enable multiple logical processors in an isolated region. Because the isolated region suffers $\approx 20\%$ reduction in utilization as visible to the provider, the impact to tenants within the isolated region will be strictly less. Cloud computing benefits from over-subscription making these gains possible. Impacts to utilization are amortized across the security domains assigned to a given isolated region. Figure 2.12 is indicative of tenant observable application performance, while Figure 2.13 is indicative of the cost to the cloud provider in terms of lost CPU utilization.

We have not shown the performance impact on schedulable units running outside of an isolated region. The Linux scheduler runs on a per-core basis, so there is no interference across core boundaries unless processes are being re-balanced, which we disable across isolated and non-isolated regions. Thus, the only performance impact to schedulable units in the non-isolated region is due to the reduction in cache sized from 20MB to 16MB. Reduced cache sizes have little impact on cloud workloads [64].

Hardware based per-partition flushing mechanisms would significantly reduce these overheads if such instructions are added by Intel in the future. For example, an instruction like `WBINVD` could be made that took a partition id and only flushed the relevant lines in that partition. This would reduce the amount of system time our approach takes, but would not



(a) 2 Security Domains (b) 4 Security Domains (c) 8 Security Domains

Figure 2.13: Scalability of Isolation Mechanisms with respect to Number of Security Domains. 4 logical processors are in the isolated region. Per logical-core utilization is shown for cores in the isolated region with co-scheduling and state cleansing enabled.

eliminate performance impact of strict co-scheduling. Our scheduling approach would still be beneficial even in the case of hardware-accelerated partition flushing because it would allow processing being scheduled during an epoch to gain maximal sharing of cache lines as they all belong to the same security domain. Furthermore, co-scheduling is still necessary to enable multiple cores within an isolated region regardless of the overhead of partition flushing. Future research is necessary to determine an effective hardware implementation of a per-partition flushing mechanism to reduce the system time.

2.6.2 Impact of Shared Memory Reduction

By enabling selective sharing of base layers in Docker, we expect an increase in the memory footprint of containers as there are multiple copies of pages that would otherwise be shared. To understand the memory growth vs. the number of security domains, we ran 2 experiments each with a web server (Apache Tomcat) and an in-memory database (Redis). We used `smem` to measure the proportional set size (PSS) per container as it represents realistic memory usage by only measuring the fair share of the total shared memory. To better understand PSS, consider two processes that share 50MB of memory and have 10MB of memory that is unique to each process. PSS would report the memory usage to be the `(shared memory`

/ `# of processes)` + `unique memory`, thus would be 35 MB for the example given. PSS is used to highlight the benefits of sharing in the non-modified cases and to show the gains from selective-sharing within security domains. We also track Resident Set Size (RSS), which is the total amount of memory used by a process. This allows us to show the cost of a naïve solution that disables sharing entirely. To make sure that the code is resident in memory before we take measurements, we send 100 requests to the Apache Tomcat servers and added 1000 random key-value pairs to each Redis server. We designed our experiments to be representative of real world deployments of micro-services where multiple containers of a single type run in a distributed fashion.

We measure how the average memory usage of each container increases as we increase the number of security domains from 1 to 4. Each security domain has 5 containers. The result is compared against the same number of containers running without modifications on the same host (*i.e.,* all the containers share base layers). These measurements are also compared against a naïve solution in which sharing is disabled all together. We measure the average memory usage across 50 runs. Figures 2.14 and 2.15 show that memory usage for Redis increases ≈ 0.45 MB per container in the worst case (4 security domains with 5 containers each) and for Apache Tomcat only ≈ 1.71 MB per container in the worst case. When looking at the overall memory consumption of these processes (≈ 7 and ≈ 230 respectively) it is clear that the additional memory cost per container is marginal. Selective sharing that enables layer sharing within a security domain provides substantial improvements over disabling sharing entirely.

Figure 2.14: Full Sharing vs. Selective Sharing (Redis)

Figure 2.15: Full Sharing vs. Selective Sharing (Tomcat)

The results support our claim that the additional cost of selective sharing of memory is negligible. The increase that we see arises from the duplicate copies of memory pages, one per security domain, which were shared among all the containers within that domain. Sharing still occurs between containers belonging to the same security domain which is why memory growth remains manageable with our modifications.

2.7 CACHE CAPABILITY ENFORCEMENT SUMMARY

In this chapter, we have presented a hardware-software technique that can enforce a cache-access capability for schedulable units belonging to separate security domains (**C1**). Unlike many existing solutions, our solution allows SMT to remain enabled (**C2**) and does not require application level changes (**C3**). We implement our system on top of the Linux CFS scheduler and present an evaluation of the system under a CPU bound workload. In our evaluations, we observed a worst case reduction in utilization in the case of 2 security domains of 9.8% and only 2.97% and 1.68% decrease in utilization for the 4 and 8 security domain configurations respectively (with 4 logical processors assigned to an isolated partition, as shown in Figure 2.12) (**C4**). A user simply notifies the provider that a given workload should be run in isolation. Our technique can eliminate an attacker's ability to use the cache as a noisy communication channel and does not rely on probabilistic methods to decrease the granularity of information available on the channel (**C1**).

Chapter 3: Trustworthy Monitoring and Intrusion Detection

Numerous event-based probing methods exist for cloud computing environments allowing a hypervisor to gain insight into guest activities. Such event-based probing has been shown to be useful for detecting attacks, system hangs through watchdogs, and for inserting exploit detectors before a system can be patched, among others. Here, we illustrate how to use such probing for trustworthy relationship logging and highlight some of the challenges that existing event-based probing mechanisms do not address. Challenges include ensuring a probe inserted at given address is trustworthy despite the lack of attestation available for probes that have been inserted dynamically. We show how probes can be inserted to ensure proper logging of every invocation of a probed instruction. When combined with attested boot of the hypervisor and guest machines, we can ensure the output stream of monitored events is trustworthy. Leveraging these probing mechanisms we can build trustworthy relationship monitoring systems to perform capability violation detection.

3.1 SYSTEM CAPABILITIES

Relationship monitoring at the system layer can be used to define and detect violations of system-level capabilities. These capabilities capture relationships between system-level subjects such as processes and the files they are capable of accessing or executing. Table 3.1 defines the terms we use when discussing system level capabilities.

Table 3.1: System Capability Definitions

	Definition of Term
$\mathbb{C}_{\mathcal{P}}$	Set of Capabilities for micro-service $\mathcal{P}$
$\mathbb{M}$	Set of micro-services
$C^{call}_{x,y,..}$	Capability to execute system call *call* with arguments $x, y, ...$

Capabilities at the system-level revolve around system calls as the system call interface is how micro-services create relationships between system-level subjects. In this chapter,

we focus on monitoring two system calls, `open` and `execve`, but explore the overhead of more thorough monitoring in Section 3.8.1. Accordingly, we introduce the notion of two capabilities to be monitored from within the VMM. First, we denote $C_{f,p}^{open}$ to mean the capability to execute the `open` system call with filename f and permissions p. Then we denote C_{f}^{exec} to mean the capability to execute the `execve` system call with binary named f.

In this chapter, we focus on the ability to place trust in the monitoring of the set of capabilities, $\mathbb{C}$. We also build a mechanism to allow for easier building of capability lists for every micro-service a tenant may be deploying, $\mathbb{C}_{\mathcal{P}} \; \forall \; \mathcal{P} \; \in \; \mathbb{M}$.

3.2 GOALS OF A HYPERVISOR-BASED TRUSTED LOG

We set forward five requirements that must be met to guarantee the integrity of a trusted log meant to monitor guest VMs. Increasing the integrity and completeness of our trusted log provides better guarantees for higher level services built using such a log. The requirements are as follows:

R1 Information provided by the guest cannot alter the logging entity's control flow. Information is simply logged and higher level services can respond to logged data appropriately,

R2 Guests cannot modify or remove an event from the log after the fact,

R3 In-guest modifications to instrumented locations should be logged,

R4 Modifications to functions invoking the hooked instruction should similarly be logged,

R5 The event log must contain every event ϵ_T of type T if there exists any probe P_T in the set of probes which produces output corresponding to events of type T, up to and including a malicious action within the guest.

We also have three design goals that drive the engineering choices behind the architecture proposed here. These are:

D1 Minimize the performance impact on guests,

D2 Minimize additions to the trusted compute base,

D3 Require no modification of guests (*i.e.,* remain transparent).

3.3 BACKGROUND

Below we describe the technologies being used to help meet the requirements listed above. By utilizing and extending these existing techniques, we meet our design goals and increase resilience against attacks.

3.3.1 Hardware Assisted Virtualization

The x86 architecture was not originally designed with virtualization in mind, but as VMs became popular, hardware manufacturers looked at ways to improve their performance and robustness. Both AMD and Intel have released support for Hardware Assisted Virtualization (HAV) in the form of extensions to the x86 instruction set.

HAV allows a VM to execute instructions natively on the hypervisor's CPU(s). However, the hypervisor must maintain control of the VM's execution. When the CPU is executing a VM's instructions, `VMExit` events are generated for any privileged operations that the VM attempts. A `VMExit` transfers control from the VM to the hypervisor, allowing the hypervisor to perform any necessary operations before returning control back to the VM.

While allowing for robust and simplified hypervisor software, `VMExit`'s do incur performance overhead. Historically, one of the major causes of overhead in HAV was due to page faults in the VM. In early HAV implementations, every page fault would result in a `VMExit` since the guest could not control its own page tables. To alleviate this, vendors introduced

a technique called two-dimensional page tables (TDP). In this paper we utilize Intel's TDP implementation, known as Extended Page Tables (EPT). The techniques apply to AMD's equivalent Nested Page Tables.

EPT allows VMs to manage their own page tables by managing guest-physical to host-physical address translations in hardware, effectively eliminating `VMExit`'s on page faults. Similar to conventional x86 page tables, EPT also provides a set of access flags that can be set at the page level: execute enable, write enable, and read enable. A `VMExit` is triggered on accesses that violate the access flags due to an EPT violation. We later show how EPT access flags can be used to guarantee that probing systems do not miss events of interest occurring within the guest immediately after guest boot, helping to fulfill **R5**. For more information on EPT we refer the reader to Volume 3 of the Intel Software Development Manuals [83]; for AMD's equivalent NPT the reader can refer to Volume 2 of AMD's Programmer Manual [84].

3.3.2 Virtual Machine Monitor Based Probing

The research community has shown that timer based guest introspection (passive monitoring) can be circumvented by a malicious or compromised guest [85, 86]. We avoid this issue by using event-based monitoring. Here, we highlight the mechanism used to enable such logging. Event-based probing using debugging techniques has been proposed and applied in a number of different contexts [87–90]. Lengyel *et al.* use event-based probing for dynamic analysis of malware with the goal of remaining undetected during monitoring [87]. Estrada *et al.* show the effectiveness of similar techniques for reliability and security monitoring [89, 91]. XenProbes uses the technique for profiling performance inside guests [88] and Spider uses it for stealthy debugging [90].

All of these approaches utilize HAV to invoke `VMExit`s upon execution of `int3` (`0xCC`) instructions in the guest. The key feature of event-based probing is that an instruction within an untrusted environment can be replaced by an instruction (`int3` in this case) that causes a hardware enforced trap (i.e., a `VMExit`) to transfer control flow to a trusted

environment. After guest inspection is done, the original instruction is executed within the guest and the breakpoint is re-inserted before guest execution resumes. Because probes cause a `VMExit`, which is an expensive operation, one must carefully design services built on such probes to reduce the number of exit events while also ensuring enough information is available to allow meaningful services be developed utilizing the logged data. We do not consider the event-based approached used by LibVMI [92] as it invokes a `VMExit` on every single instruction in the target page for the logged event. Such an approach causes high overhead and is intractable due to our performance requirement **D1**. Our approach gives users the flexibility to determine the overhead paid based on the level of protection deemed necessary for a given application.

3.3.3 The Semantic Gap

Any Virtual Machine Introspection (VMI) application must cross the "Semantic Gap" - the gap faced by developers of code running within the VMM that must inspect guest memory with no knowledge of the kernel data structures or memory layout of the guest. Much research has been done in this area, and we point the reader to the overview done by Hebbal *et al.* for a more thorough discussion of the issue and many of the proposed solutions [93]. For this work, we assume that the address of the `sys_exec` and `sys_open` calls in Linux, along with the offset at which the Linux kernel `.text` addressing begins are provided (this memory mapping is well documented [94]). The latter is needed in order to identify the guest physical locations of the above functions, which are loaded into memory before paging is enabled in the guest. In Section 3.5 we discuss in more detail why this is necessary.

We favor the approach of querying `System.Map` for the location of relevant functions due to ease of access; this approach has shown to be successful in the literature for providing a low cost method for crossing the semantic gap [88, 95, 96]. We limit our discussion to Linux guests as the open source nature of Linux lends itself to easier distribution of VAs, the focus

of our IDS, but a similar approach of querying the debug symbols for the Windows kernel has also been met with success [87, 96].

3.3.4 Virtual Appliances

VAs are a popular method for deploying micro-services. One can simply choose an appliance from a list of images made available on a cloud provider's marketplace and immediately deploy services such as databases or web servers with minimal configuration. The tuned nature of these appliances makes their behavior more predicable than a VM used for general purpose computation. In this paper we present an IDS that leverages the "appliance" nature of cloud based deployments of micro-services instantiated using VAs. The IDS is built using guest-event driven hypervisor-level probes to deliver relevant information to the policy compliance layer.

A typical deployment of a cloud based web application may have a reverse proxy routing requests to an application processing layer, each of which communicate with a database before returning a response. Each of these services would be a different VA. We envision a system for which different policies protect each kind of VA. In our example, there there would be three main policies, one for the reverse proxy VA, one for the database VA and one for the VAs which serve as the application server(s). Policies can share rules if VAs are built using the same base distribution as a single distribution will have the same cron binaries running, for example. While policies are stackable, the main advantages reside in the policies for each that differ, allowing for good coverage while limiting false positives. For example, a database server running MySQL should never execute a shell outside of configuration events; our monitoring system would detect such an operation as a violation. The event log can then also be used as compliance monitoring during configuration periods, and could serve as a method to detect insider threats attempting to re-configure applications in an attempt to cause unstable behavior.

3.4 ATTACK MODEL AGAINST THE LOGGING SYSTEM

Keeping the above technologies in mind, we motivate our design choices using the attack model and assumptions presented in this section to improve the resilience of the logging system. We assume that the hypervisor is a trusted entity and that the hypervisor side of the logging framework is secure. For the log file itself, a simple way to provide guarantees is to use remote logging, or approaches used in literature [97]. Here, we focus on the elements of logging that must be in place to facilitate proper logging of a guest that may become malicious at some point after boot.

We assume that the hypervisor is using trusted boot, thus the integrity can be attested. Additionally, we assume that guests running on the hypervisor are also using attested boot mechanisms, such as those presented in [98] and [99], and that guest kernels are known, non-malicious builds of Linux. This allows the hypervisor to guarantee the integrity of any guest kernel before the guest boots. We assume that the guest kernel is not malicious until after the first user-space program runs. This is a reasonable assumption as attempts to exploit a kernel will come from software loaded after boot (either malicious software will be loaded or vulnerable software exploited).

Attacks can include loading kernel modules, modification of the kernel in place, or attempts to circumvent logging through process tampering (more details in Attack **A2**). An attacker may try and copy the page of memory with the replaced instruction, fix said instruction, and redirect system calls to this new page. Such a redirect would either require modification of the Interrupt Descriptor Table in memory that is referenced by the general system call handler or may come as a write to a hardware register in an effort to circumvent the code block executed after an interrupt.

To protect the integrity of an event placed at a given location, we must protect the entire stack trace leading up to the execution of that event. In this chapter, we are primarily concerned with logging the specific handlers of select system calls. Our attack model against

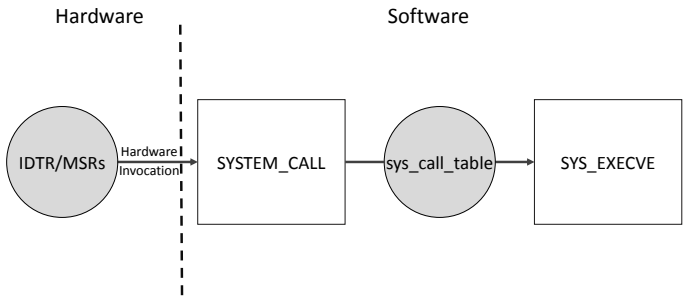


Figure 3.1: Invocation Process for a Specific System Call Handler

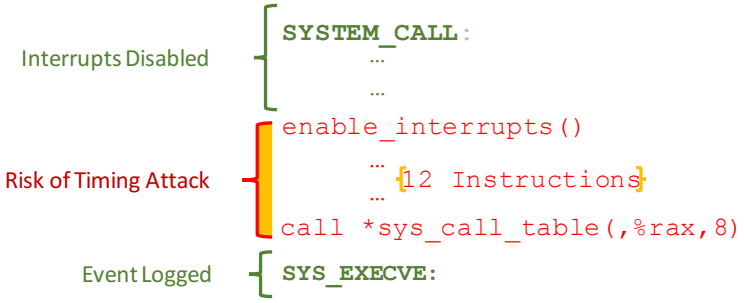


Figure 3.2: Timing Constraints for Interrupt Attack (**A2**)

the logging system concerns protecting against any modification of the steps leading up to the invocation of the monitored system call handler as highlighted by Figure 3.1. The figure shows the hardware invocation of the `system_call` code block, the location of which is designated by values stored in hardware registers. During execution of the general system call handler, interrupts are re-enabled (shown in Figure 3.2). The general system call handler transfers control to the specific handler through the `sys_call_table`. Our goal is to place probes on the first instruction of the specific handler. With that in mind, consider the following list of attacks that could circumvent logging:

A1 Write to either the `IDTR` register (for legacy `int $80` based system calls) or various MSRs for so called "fast" system calls to force the hardware to invoke a malicious code block after interrupts. (See Section 3.6 for a more detailed discussion of the specific registers).

A2 Coordinate an interrupt after a system call (that is being logged) is made and interrupts

have been re-enabled, but before the specific system call handler has been invoked. Upon interruption, modify the `thread_struct` of the system call invoking process to point to a different system call handler upon being re-scheduled. The timing constraint of this attack is highlighted by Figure 3.2.

A3 Rewrite the general system call handler to reference a new, attacker supplied, Interrupt Descriptor Table.

A4 Rewrite the entry for the specific system call being hooked in the Interrupt Descriptor Table to point to an attacker supplied handler for the system call.

A5 Rewrite replaced instruction(s) with the original instruction.

A6 Simulate a system call interface using an alternative means of communication between userspace and a root-kit

In section 3.6 we highlight how attacks can be accounted for through hardware enforced events. It is worth noting here that **A2** has tight timing constraints (an interrupt would have to occur within a 12 instruction window). We later discuss how removing the protection guarantees for **A2** greatly reduces the performance impact and we believe it has minimal effects on the overall trustworthiness of our logging architecture. While we do not currently defend against **A6** style attacks, we believe our system greatly increases the cost of attack while providing good detection coverage of many of the attacks that might try and circumvent logging.

3.5 TRUSTWORTHY LOG ACQUISITION

With our attack model in place, we now discuss the specifics of the logging mechanism. In particular, we discuss how and where probes are inserted. The algorithm for guaranteeing a probe is inserted before the execution of the probed instruction is introduced and the relationship between the implementation and design requirements is described.

3.5.1 Probing Mechanism

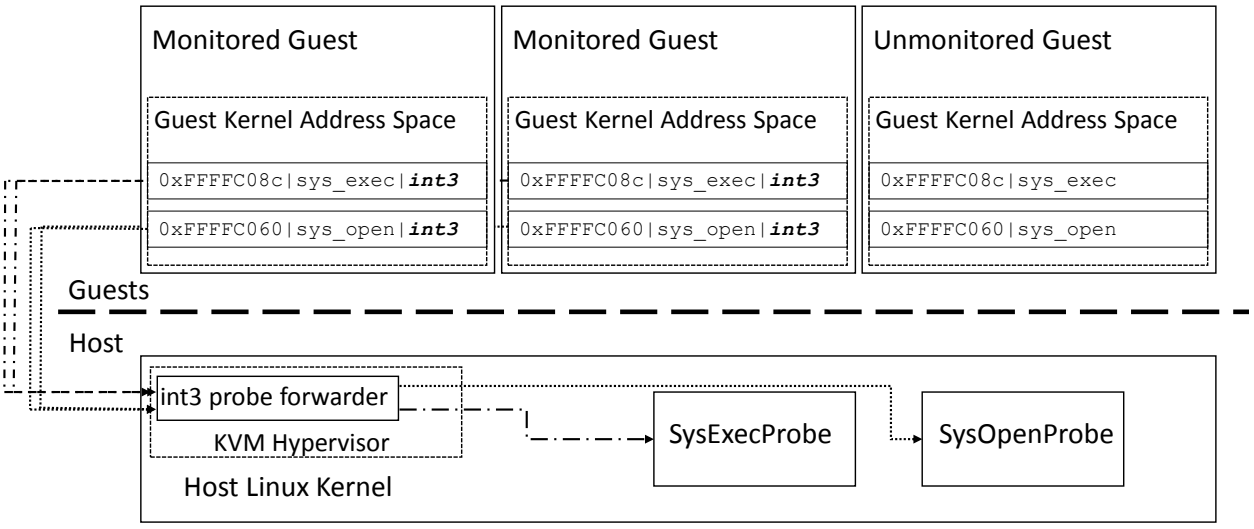


Figure 3.3: Event Driven Probe Architecture

Figure 3.3 highlights the mechanism to probe the Linux kernel system calls `sys_exec` and `sys_open`. An event-based probing mechanism is utilized to replace instructions in the guest kernel [89], ensuring information is logged anytime the affected functions are called, fulfilling **R2**.

To ensure that any attempt to modify a probe is logged (**R3**) we use EPTs to remove write permissions for the affected page, register a callback to handle these EPT violations, and within the callback handler only log attempts to modify the affected page if the violation occurs for the guest virtual address on which we inserted the probe. While performance monitoring within the guest might cause non-malicious writes to locations of logging probes, an administrator would know the event is benign. Event classification is left to higher level services; we guarantee only that modification events do appear in the log. Logging code remains small, making formal verification more feasible. There are only 72 and 41 lines of code for our `sys_exec` logger and `sys_open` logger respectively (not including the code required to insert the probes), keeping in line with **D2**.

3.5.2 Log Completeness

We have defined log completeness to mean that our logging service guarantees that every invocation of a probed event be present in the log. In order to ensure log completeness and fulfill **R5** we must place probes in their respective locations *before* the instructions at those locations are executed. While we can guarantee this, it may be possible for an attacker to perform system call like actions, all together bypassing probed instructions (**A6**). The system calls being probed will be loaded at a predictable location within the guest physical memory (as noted in Linux's memory mapping documentation [94]). The knowledge of these locations allows us to determine the page number indicating the page containing the target instruction, which we use to watch for EPT violations of any guest physical address that occurs on the same page as an instruction of interest during the guest boot sequence. We are able to watch for such violations by utilizing a callback handler that gets invoked after we have allowed KVM to perform any necessary actions to handle the violation. Upon observing the first write violation for any address within the page of interest, we remove the execute bit from that page, allowing our callback handler to be invoked if any instruction on the page is executed. Subsequently, upon observation of any instruction execution on the page of interest, we know that the remaining code for that page must be loaded and can safely insert the probe. Having inserted the probe, we restore EPT permissions to allow execution and remove our checks for EPT violations due to execution exceptions on the page in which the probe is inserted as the checks are only required as the final step before probe insertion. By inserting probes in this manner during boot of guests, we are able to ensure log completeness and log every call to these two system calls, even while the first userspace applications are being started. This process is shown in Figure 3.4.

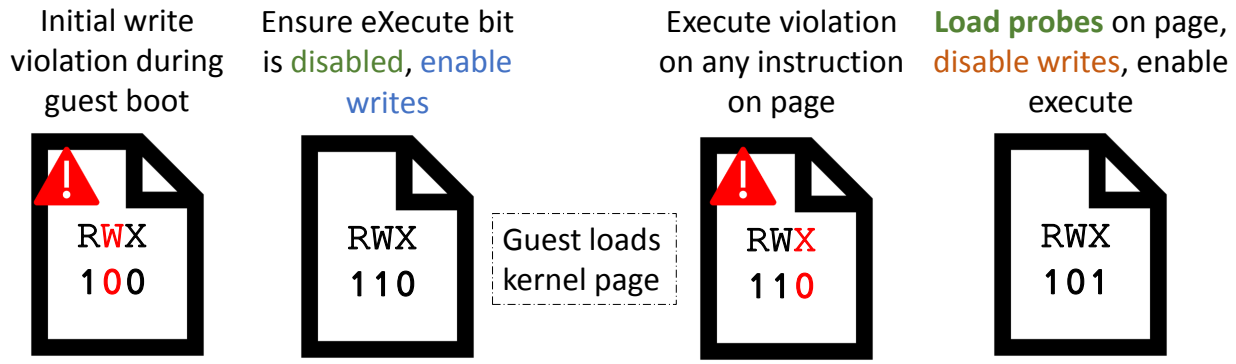


Figure 3.4: Induced EPT Signature & Probe Insertion

3.5.3 Implications of an Untrusted Guest

Finally, we must ensure that the actions taken within the probe do not place unwarranted trust in data obtained from the guest (**R1**). For example, our `sys_exec` logger logs two variable length string arrays. While these strings are typically `\0` terminated, the guest could point the probe to a location with an arbitrarily large number of bytes before a `\0` is encountered. To protect against copying strings from guest memory, we only copy 500 bytes and place a `\0` at the 500th byte. While we may log garbage data in cases of an intentionally malicious guest and may truncate binary names in the case of exceptionally long, but legitimate, calls to `sys_exec`, this is a necessary trade off to ensure the probing interface remains resilient. Potential for truncating can be seen again when iterating through variable length arrays, which should be `NULL` terminated. We only iterate over up to 50 entries and exit iteration if `NULL` is encountered (in a legitimate case) and stop at 50 in the case of a malicious guest pointing the probe to a random memory location. Again, this has the side effect of potentially truncating logged arguments. In our experiments, we never truncated any legitimate data. The length decisions did not impact the ability to log meaningful data. Regardless of whether or not arguments are truncated, we are able to protect the logging facility from the attacks listed above.

3.6 LOGGED EVENTS

Choosing which events to log is critical to increasing resilience against the attacks listed in Section 3.4 and detect capability violations. Here, we show which events should be logged and how each event type can be used to improve resilience.

In addition to the information listed for each event type as defined below, all events also include the `hostname` of the KVM hypervisor on which the event occurs, a `timestamp` for the event, and the `vmid` (the qemu-kvm process id of the VM on the host on which the event is logged).

The five event types currently in our system, and information collected unique to each type, are as follows:

- T_{mod} - Address modification events containing:
  - `gva` - A long integer indicating the guest virtual address being modified.

The next two events log activity of interest and are useful for facilitating detection of abnormal actions.

- T_{se} - `sys_exec` events containing:
  - `filename` - a `\0` delineated string.
  - `argv` - a `NULL` delineated variable length array containing string pointers.
  - `envp` - a `NULL` delineated variable length array containing string pointers.
- T_{so} - `sys_open` events containing:
  - `filename` - a `\0` delineated string.
  - `flags` - an integer flags variable indicating options for the file.
  - `mode` - an integer indicating the mode for the file being opened.

The following two events are unique to logging system calls and increase the cost of circumventing the logging mechanism. These require additional callbacks be provided by the underlying probing framework. To prove viability, we have implemented the `wrmsr` event for modern system calls. These two events are not currently provided by any event-based monitoring framework discussed in the literature [87–89] as they only become necessary when providing defenses for system call monitoring.

- T_{lidt} - `lidt` event. Triggered on execution of the `lidt` (Load interrupt descriptor table) x86 instruction.

- T_{wrmsr} - `wrmsr` event. Triggered on execution of the `wrmsr` (Write Model Specific Register) x86 instruction.

These two events are hardware enforced; once the hypervisor has configured the processor to trap these calls, their execution will always force a `VMExit`. The `lidt` trap can be configured by setting bit 2 (Descriptor Table Exiting) of the `IA32_VMX_PROCBASED_CTLS2` MSR to 1 within the hypervisor before VMs are started. Similarly, writes to MSRs within the guests can be trapped by ensuring bit 28 of the same MSR is 1 and then configuring the MSR bitmap field in the Virtual Machine Control Structure to only cause `VMExit`s on writes to the specific registers that need monitoring. This ensures that performance overhead remains low by not inducing `VMExit`s for writes to every MSR. For `int $80` based system calls, the `lidt` trap is sufficient. For `sysenter` invoked system calls, the three MSRs `IA32_SYSENTER_{ES, EIP, ESP}` must be monitored through the `wrmsr` trap. Finally, for `syscall` invoked system calls, the MSR `IA32_LSTAR` must be monitored with the `wrmsr` trap. The registers listed above are used to register Interrupt Service Routines (ISRs) with the processor. In Linux, these point to the general system call handler. The performance impact of these two events should be negligible under normal operation as these events occur only during boot of the guest kernel and during configuration of MSRs.

3.6.1 Detection of Attacks on the Logging System

Let us now consider how these event types can facilitate detection of the attacks against the logging facility listed above. We protect against attacks **A5**, **A3**, and **A4** by properly removing the write enable bits for the pages containing the instruction modified, the general system call handler, and the interrupt descriptor table and listening to events of type T_{mod}. The event T_{mod} is hardware enforced by EPT. Attempts to modify pages for which the write enable bit has been removed will trigger a `VMExit` through an EPT violation. Attacks that try to change the ISR for system calls (**A1** above) can be logged with events of type T_{lidt} and T_{wrmsr}. Finally, careful placement of probes can ensure that logging occurs before interrupts have been re-enabled by placing the probe on the general system call handler, mitigating attack **A2**. Mitigating **A2** does have high performance impact as we discuss in Section 3.7; we believe placing the probe at the specific system call handlers is a reasonable trade-off as attacks of this kind must meet a tight timing constraint. Finally, if an attacker can compromise the guest kernel it would be possible to recreate a separate system call interface (**A6**). Consider a root-kit that finds the `task_struct` of the userspace process it is hiding. It could poll the memory of the process for system call like arguments and then execute a separate code block, performing the same actions as a system call. Such an attack may be possible to detect by monitoring timing interrupts. Future work can explore ways to use event-based probing to further protect the guest kernel against attacks of this kind.

Note that many more event types are possible as event-based probing provides a trusted mechanism with which to hook any kernel function. But in keeping with **D1** and **D2**, we choose to keep this number small.

3.6.2 Event Logging Format

All probe output is placed into the VMM's `/var/log/kern.log`. Output is processed by a user-space application that builds and processes events. This design is shown in Figure

3.5. In order to allow for easier processing by higher level applications, we adhere to a JSON like format when doing logging within the host kernel.

A log sample for a `touch text.log` event is shown in Listing 3.1.

Listing 3.1: Example `sys_open` Probe Output

```
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "BEGIN"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
  "KIND": "ARG","ARG_NAME": "filename", "VALUE": "test.text"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
  "KIND": "ARG", "ARG_NAME": "flags", "VALUE": "0x941"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
  "KIND": "ARG", "ARG_NAME": "mode", "VALUE": "0x1b6"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "END"}
```

The `TIMESTAMP, HOSTNAME` and `LOG_ID` are also included and are set by the `printk` function within the hypervisor. We trust these fields to be accurate when read by higher level tools. The accuracy of these fields is important as will be discussed in the next section on the development of higher level tools.

For each event, we have a `BEGIN` statement and an `END` statement. Everything in between those statements make up the body of the event and are used to log parameters read from the guest.

3.7 INTRUSION DETECTION FOR MICRO-SERVICES

To highlight our approach to services built on top of an event based log, we have developed an IDS which triggers alerts on violations of capability lists. The checks are performed on filenames passed to guest `sys_exec` and `sys_open` calls to enforce the capabilities described in Section 3.1. To enable ease of use, we have also built a policy recorder that translates guest events to white-list capability sets during the recording or learning phase.

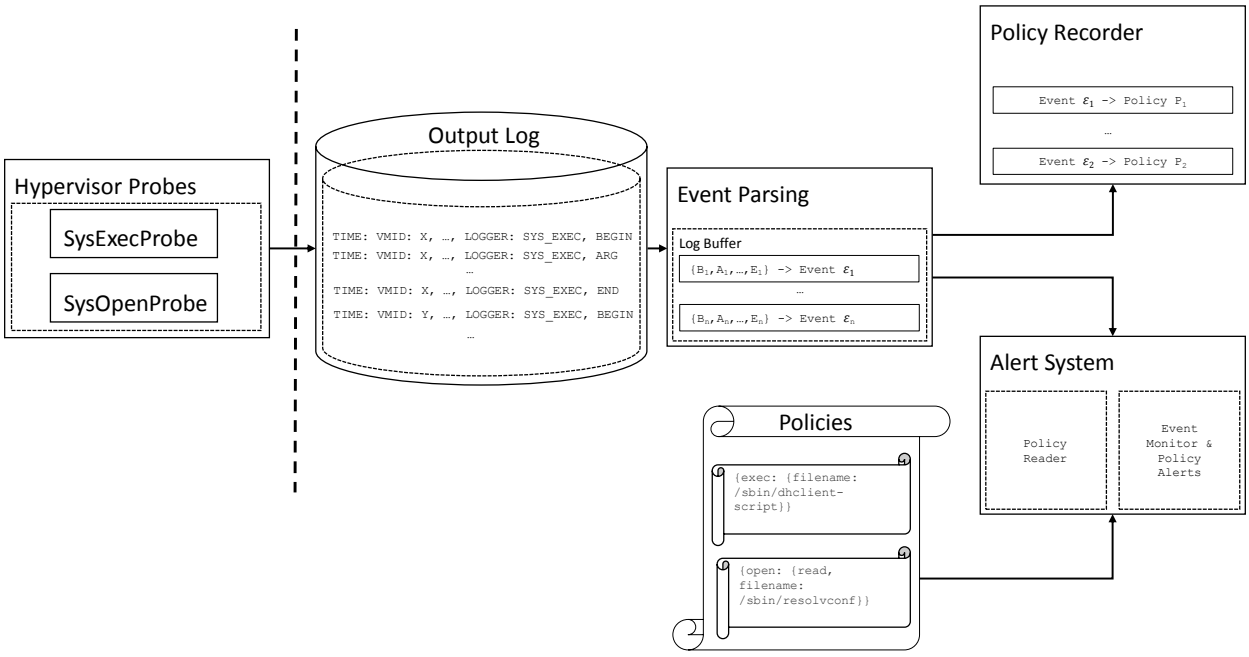


Figure 3.5: Trustworthy-Log Driven IDS Architecture

3.7.1 IDS Architecture

The architecture of our intrusion detection system is shown in Figure 3.5. Raw probe logs are transferred from kernel to user space using the `/var/log/ kern.log` interface. From there, the logs are placed in a buffer as they are read from the file. An `ioctl` interface to `/var/log/kern.log` is used to ensure updates are pushed to the user space application as soon as probes write to the file. Within the user space event parser, buffers must be used to ensure that output from a probe P^1_{so} into guest G_1 do not become integrated into an event ϵ_2 from the output of the probe P^2_{so} placed into guest G_2, as the arrival of such logs may be intermingled within `/var/log/kern.log`. This is ensured by placing all logs from a given probe into a unique buffer identified by the `LOGGER_TYPE,HOSTNAME,VMID` sequence. Since the buffer being used is determined by this sequence of values, these values must be set by the hypervisor. No value read from the guest is used to identify a probed event or which buffer in which to place a logged statement, ensuring the guest can not impact actions taken by the logging system. For now, we do not consider multiple vCPU guests, thus only need to worry about intermingling between guests. In the case of multiple vCPUs, the vCPU

id would also need to be used as a unique identifier as it would be possible that a probed location be called from multiple vCPUs simultaneously. This limitation is partly due to the chosen probing framework, other frameworks [87] would support multiple vCPU guests.

3.7.2 Policy Generation

After event parsing is complete, processed events are passed to either a policy recording layer or an alert system for our IDS. The policy recording system allows an administrator to build capability sets by recording standard behavior for a VA in terms of white-listing the actions taken during policy recording. Listing 3.2 shows an example policy built using our policy recorder while executing the `which` command on a guest under inspection. Currently, white-lists are separated from attackers executing in the guest by the VMM. Continuous integration test suites could be used to generate polices through this policy recording mechanism.

Listing 3.2: Example `which.policy` file

```
{ "policies": [
{"exec": {"type": "whitelist","filename": "/usr/bin/which"}},
{"open": {"type": "whitelist","access_type":
"read", "filename": "/etc/ld.so.cache"}},
{"open": {"type": "whitelist","access_type":
"read", "filename":
"/lib/x86_64-linux-gnu/libc.so.6"}},
{"open": {"type": "whitelist","access_type":
"read", "filename": "/usr/bin/which"}}]}
```

3.7.3 Threat Analysis

We note that there are certain limitations to our approach that would allow an attacker to commit a malicious action without being logged. Consider a vulnerable binary running on a system that is compromised through a buffer overflow attack. Assuming the attacker does not crash the binary, it would be possible to run code under the guise of an already executing process. As long as the payload never opened a file or executed another binary, it would go unlogged. While any such process hijacking will go unlogged, our approach substantially reduces the actions that can be taken by an attacker. Adding a separate event for system calls dealing with network access would further mitigate the possibility that a malicious payload is able to do any useful work without being logged. Our approach is complimentary to and can be combined with other defense-in-depth approaches such as ASLR, non-executable heaps and other defenses to increase the cost of implementing a successful attack.

3.8 EVALUATION

In this section we evaluate both the impact of the probes on the performance of the guest and on the ability of the IDS to detect attacks on applications running in VAs. The IDS is evaluated against a real world attack on a popular cloud based web application.

3.8.1 Performance

To evaluate the overhead of our probing mechanisms driven by guest events, we run three benchmarks that are representative of cloud workloads. These include:

- Apache Bench - a benchmark for the Apache web server [100],
- Redis Bench - a benchmark for the in memory data store [101],
- OpenSSL Profiling - used to understand the impact on encrypted communication within guests.

These tests were chosen because they represent a disk-read heavy workload (Apache), network heavy workload (Redis, Apache), and a CPU heavy workload (OpenSSL). Cloud applications will often call in-memory caches before sending a response using Apache configured with OpenSSL. All tests are configured using the Phoronix Test Suite and are run 90 times each. The first 30 runs are performed with our trusted probes loaded and then run 30 times without. The last 30 runs are done while having probes loaded at the general system call handler, before interrupts have been re-enabled in the guest to highlight the performance penalty paid while protecting against **A2**. Figure 3.6 shows the results for both Apache Bench and for OpenSSL. Apache bench results are in terms of requests served per second and those for OpenSSL are in terms of signatures generated per second, but here both have been normalized to highlight the percentage decrease in performance caused by probing. Looking at the first two bars in Figure 3.6, it is easy to see the performance implications of placing probes at the generic system call handler. In the case of the specific handler (the first bar), we see less than 10% overhead. But Apache has about a 55% overhead when placing the probe at the general handlers. The next two bars highlight how probing has little impact on the performance of OpenSSL, regardless of probe location. This is because OpenSSL does not have to interact with the kernel as much as Apache and Redis to complete its workload. OpenSSL works by loading a key in memory and then generates

signatures using that key. It is up to another process, Apache for example, to write out any information to the network.

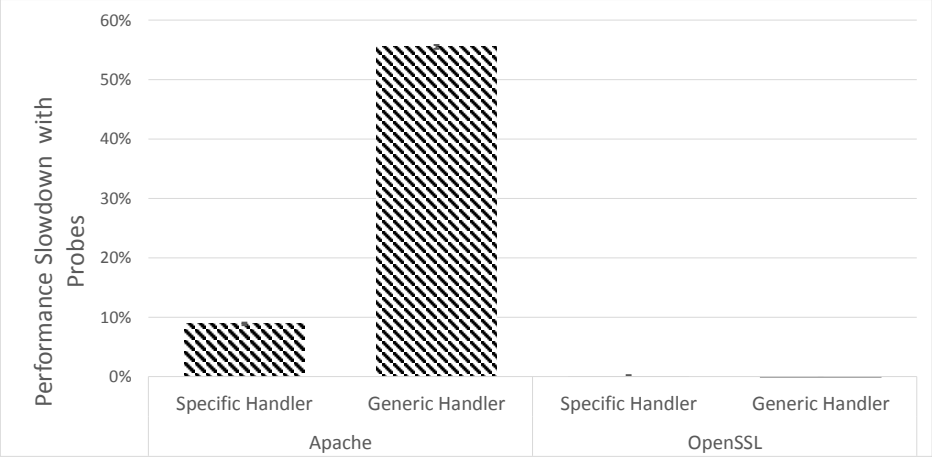


Figure 3.6: Apache Bench and OpenSSL Overhead Relative to Running with no Probing.

Figure 3.7 is for the Redis benchmark, which runs five separate request types against the in memory data store. As seen from the "B" bars in the figure, the performance overhead of probing specific handlers when compared to no probing, the "A" bars, is negligible. The "C" bars for each query type show the high performance impact of defending against **A2**, which is about 75%.

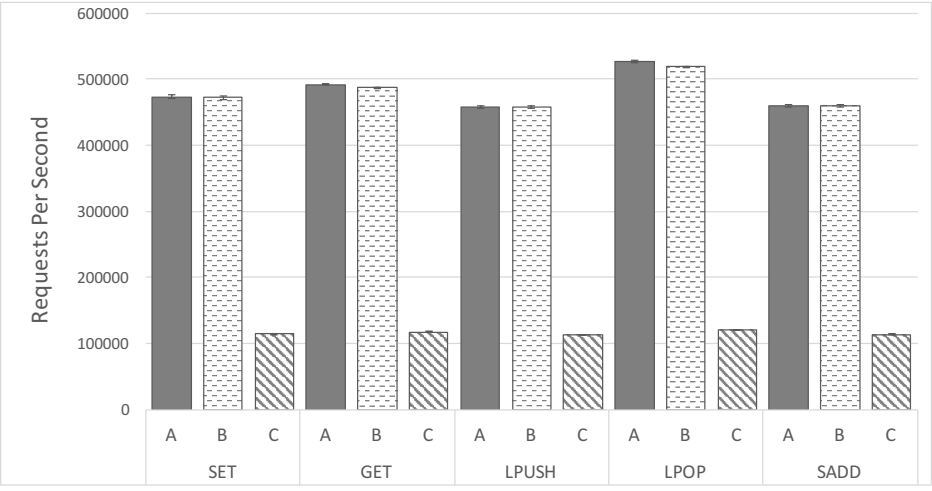


Figure 3.7: Redis Benchmark Overhead for 5 Redis Operations. (A) without probing the guest, (B) probing only the specific system call handlers, and (C) probing the general system call handler.

In the case of hooking specific system call handlers, it is clear to see that overheads remain tolerable (less than 10%), because we are only probing two guest kernel functions. The overheads are large when protecting against **A2** though, around 55% for Apache and 75% for Redis. Apache and Redis are both opening sockets and sending data over the network, which is why we see a much higher penalty being paid when hooking the generic system call handler. We feel that the protections against **A5**, **A3**, **A4**, **A1** (requirements **D3** and **R4**) go a long way in protecting the specific system call handler, substantially reducing the unloggable attack space when hooking only the specific system call handlers.

3.8.2 IDS Evaluation

We evaluate the efficacy of the IDS built on top of our trusted logging platform by looking at real world exploits for motivation. In a recent attack on the website for the Linux distribution Linux Mint [102], attackers were able to gain shell access as the `www-data` user, the user typically reserved for only running the `httpd` process [103]. The attack exploited a vulnerability in the popular blogging framework, Wordpress. Wordpress is representative of a typical cloud application as it can be deployed on many VAs to enable horizontal scalability. To see how our system would have handled such an attack, we installed a copy of Wordpress with a typical plugin and attacked the setup using Wordpress Vulnerability Database ID #8209 [104].

We first setup a Wordpress application server and separate database server to act as our VAs. Since our IDS supports policy stacking, we are able to record a separate policy for Wordpress and use the `dhcp.policy` file common to all VAs built using the same base Ubuntu 14.04 LTS distribution. The `dhcp.policy` file was auto-generated by running our policy generation tool against the log output of a default Ubuntu install. Including that policy is necessary as it removes the chance of false positives every time a dhcp lease renewal is performed. It would not be necessary for VAs using static IP's. An abridged version of the Wordpress policy file is shown in Listing 3.3. Our policy recording utility auto-generated a

policy that served as a starting point and then we used knowledge about proper Wordpress installs to fine tune the policy. For example, the policy recording utility produced many single `filename: /var/www/html/*.php` entries. We removed these and converted it into a single `directory: /var/www/html` entry as shown on the first line of the policy in the listing.

Listing 3.3: Abridged `wordpress.policy` file

```
{"open": {"type": "whitelist","access_type":  "read",
"directory": "/var/www/html"}},

{"open": {"type": "whitelist","access_type": "create",
"directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type": "modification",
"directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type": "read",
"directory": "/var/www/html/wp-content/uploads"}},

{"open": {"type": "whitelist","access_type": "create",
"directory": "/var/www/html/wp-content/plugins"}},
```

We exploit the vulnerability using Metasploit [105] to determine if our alerting system is able to capture capability lists violations. Because the exploit works by injecting arbitrary PHP code, we can only detect attacks that use PHP to access other files on the system (outside of the /var/www/html directory) or execute system binaries. We detect the exploit after the attacker performs an action anomalous to the hijacked process. In this case, the attack is detected upon attacker execution of a shell, as `/bin/sh` should never execute on the system. We could detect the exploit sooner by adding an extra probe to `sys_socket`.

While we have demonstrated the IDS on one example, it has shown the viability of our technique. Our approach relies on the fact that many exploits require a binary to load and execute on a system. If the exploit does not run in a separate process, as is the case in the example given above, the attacker will likely either execute a system binary or open a file, revealing malicious activity. For instance, the loading of kernel modules could be audited by looking at events of type T_{se} with `filename` equal to `insmod`. This would potentially reveal the loading of a root-kit by enforcing kernel module loading capabilities. Payloads executed through process hijacking can explore the full system call interface and potentially exploit the running kernel. Such an event would not be logged, though any attempt to remove our probe using such an exploit would be noted in the log. This increases the burden of carrying

out successful attacks as malicious payloads will have to be carried out within a vulnerable binary or the kernel to remain undetected. Future work could explore creating capabilities and probes for the most vulnerable locations within the Linux kernel by evaluating past exploits.

3.9 RELATED WORK

Huh *et al.* discuss a trusted logging architecture for grid computing using Xen [97]. Their approach relies on logging events as they are intercepted by Xen device drivers. Our trusted logging is more flexible as any action within the guest can be logged on instruction execution. Additionally, the authors propose an extensive architecture for guaranteeing the log is not fabricated by the provider. We view this work as complementary. Thus far, we have focused on trust related issues related to log generation and can utilize similar techniques for improving trustworthiness.

Crawford *et al.* discuss a methodology for detecting insider threats that relies on scanning the memory of running virtual machines every 30 minutes [106]. As we discussed earlier, polling techniques such as this are limited in that they are easily circumvented, giving attackers a 30 minute window in which to perform malicious activities. Kienzle *et al.* explore using VMI techniques for endpoint configuration compliance, but require the compliance audit package run in a separate VM, increasing the resources of the monitor [107]. Their approach to compliance also relies on polling, thus can be circumvented. Our approach provides a trusted log which is guaranteed to capture every event probed. Our work can be extended to perform compliance checks of Mandatory Access Control systems running within guests. Win *et al.* propose using VMI to provide additional layers of security for a similar system, but rely on information from a trusted in-guest monitoring agent to report relevant accesses to a trusted compliance layer VM [108]. Our approach places no trust in the guest after the initial kernel is loaded using an attestation technique provided by a TPM.

KvmSec is a security extension for KVM, but relies on probes running in untrusted guests [109]. Our approach places no trust in the guest. In "Space Traveling across VM" [110], the authors cross the semantic gap by relying on an additional virtual machine from which to run probes. This approach has a large overhead, thus would violate **R6**. Techniques like "Virtuoso" are complimentary to our trusted log and could be used to inform future probes of relevant locations within the guest for probing [111]. With regards to work related to IDS, Kosoresow and Hofmeyr show the effectiveness of system call traces by using temporal patterns of system calls to detect intrusions [112]. While the IDS presented here relies upon white-listing, their technique could also be applied.

HIMA [113] provides run time integrity checking of userspace programs. The authors monitor system calls to enable these integrity checks. Their approach used a much older VMM that generated `VMExit`s for every interrupt, thus the authors paid minimal overheads during monitoring. Modern hardware does not exit on every interrupt, thus we utilize event-based probing to monitor specific system calls.

Chapter 4: Intra-Application Capabilities for Micro-Services

As discussed in our chapter on IDS tuned for micro-services, detecting intra-service attacks are difficult because abnormal behavior may never access data outside of the service. In this chapter we address this limitation by exploring runtime integrity checking for applications running in the cloud. Specifically, we explore runtime integrity violations of high value assets such as databases and reverse proxies as these are cornerstones to scale out workloads common to micro-service deployments.

While our IDS is effective at detecting common attack payloads, it is unable to detect attacks that reside entirely within the vulnerable binary. The focus of the IDS is to detect abnormal behavior of binaries, say reading `/etc/shadow`, at a system level. Attacks on specific systems, such as a web server, may be able to access sensitive information without ever leaving the binary. Consider username and passwords being sent to a web server. If an attacker can divert control flow within the web server, these could be captured. These two approaches complement each other. Any attempt at Control Flow Integrity (CFI) will not detect system level changes which are still useful to check for improper configurations and insider threats.

In this chapter, we build a provenance engine for the C abstract machine that can generate memory access capabilities for instruction flows. These instruction to memory access capability lists can be used to augment CFI methods.

4.1 FINE GRAINED CAPABILITIES

A micro-service can modify its subject domain through the use of relationships as discussed in Chapter 1. These relationships may be functions of internal state as shown by $\mathcal{R}3$ in Figure 1.1. The goal of this chapter is to produce instruction to memory region modification capability lists for any memory region that can modify the micro-service's subject domain. Table 4.1 defines the terms we use when discussing fine-grained capabilities within a micro-

service.

Table 4.1: Fine-Grained Capabilities Definitions

	Definition of Term
$\mathbb{I}$	Set of memory related instructions (allocation, modification)
$\mathbb{C}_{\mathcal{I}}$	Set of capabilities indicating the memory regions a given instruction $\mathcal{I}$ is allowed to modify
$\mathbb{U}$	Set of sensitive usages
$U_{\mathcal{M}}$	Sensitive use of memory buffer $\mathcal{M}$
$\mathcal{I}$	Instruction $\mathcal{I}$ that allocates, writes to, or copies to an address
$C_{\mathcal{M}}$	Capability to influence memory region $\mathcal{M}$

Our goal is to identify "sensitive" memory regions that impact the relationships, discussed in detail in Section 4.4.1, and then protect those regions by identifying which instructions hold the capability to influence those select regions. We use the word "influence" because an instruction may modify a given memory buffer that is then used as the source of a `copy` function that writes into a sensitive buffer. We aim to capture these instructions and produce capabilities for them as well as instructions that write directly to sensitive areas.

4.2 BACKGROUND

Memory protection in type unsafe languages is needed because application vulnerabilities, such as buffer overflows, allow an attacker to craft an exploit that can overwrite function pointers or return addresses, diverting control flow from the application writer's original intent. In recent years, this has become much more difficult with the rise of Data Execution Prevention (DEP) [114] now accelerated by most hardware platforms by bit-permissions on individual pages. DEP makes it impossible to use a buffer overflow to write shell-code into memory and then direct control flow to the data region. In response, attackers began using "code-reuse" attacks such as ROP [115] to redirect control flow after overwriting a return address on the stack and Jump Oriented Programming (JOP) [116] by overwriting a function

pointer. Often, an attacker's ROP payload simply disables DEP before directing control flow to a larger payload. The payloads are built using what are known as "gadgets" that perform some operation and end in either a return or jump to an attacker controlled pointer. DEP has since been augmented by Address Space Layout Randomization (ASLR) [117] which attempts to make it harder to divert control flow to valid functions by randomizing the memory locations of functions and gadgets. This has numerous problems; a memory read vulnerability, such as use after free, can be used to leak an address, the offset of which can be used to calculate the addresses of gadgets important to attackers. Attacks on ASLR use memory leaks and side-channels to enable ROP attacks [118–121]. Attacks have now become probabilistic and depend on how well addresses can be predicted and attacks tuned to the new information at runtime.

CFI attempts to mitigate the inherit security risks associated with type-unsafe languages stemming from the lack of memory safety [122]. Transparent recompilation enabling full memory safety remains impractical due to high overheads [123, 124]. The concept of CFI was originally introduced by Abadi *et al.* [122] and focuses on enforcing a valid Control-Flow-Graph (CFG) instead of enforcing full memory safety. Current approaches to CFI attempt to apply a less-strict approximation of CFI by only ensuring that control flow can be transferred to a list of approved locations [125], but these approaches continue to be vulnerable [126]. Existing approaches rely on static analysis and instrumentation in an effort to drive adoption at low performance impact. It has been suggested in literature numerous times that some form of runtime checking be performed to increase the efficacy of CFI approximations, and it is generally accepted that full CFG enforcement comes at too high performance penalty [122, 126].

CFI works by enforcing a given CFG, usually derived using a static points-to analysis such as DSA [127], Andersen's [128], or Steensgaards [129]. The flexibility of C/C++, such as that provided by void pointer casting, causes inaccuracies in the CFG's produced by points-to analysis algorithms. These inaccuracies are due to the same language features that

cause the security challenges that CFI tries to protect against. Control Jujutsu shows that a CFG produced by DSA, the most accurate point-to analysis algorithm, remains vulnerable to attacks [26]. The authors of Control Flow Bending go a step further and evaluate attacks under a hypothetical ideal CFG, one that contains only edges intended by the original programmer and show that attacks are still possible [25].

The root cause of the attacks presented in Control Flow Bending and Control Jujutsu is carefully crafted memory corruption such that an attacker can achieve control of the system while remaining within the CFG. For example, if the buffer being passed into `exeve`'s `filename` argument is allocated on the heap, then the attacker simply has to find a heap overflow to setup an attack. The attacker can execute arbitrary programs on the host if `execve` is being executed within a function that is called via a function pointer. The authors call these types of functions "Argument Corruptible Indirect Call Sites". The indirect call is necessary for the attack to work - the sensitive function must be invoked through a function pointer. In this way, the attacker can control both the function being executed, `execve` and the data being passed into it, `filename`.

These two attacks highlight how memory corruption of select function arguments can be used to achieve malicious behavior. It is clear that CFI techniques significantly raise the bar for attackers, but remain insufficient. We explore a system in which *select memory safety* can be combined with CFI for a more comprehensive defense through instruction level capabilities. Our approach to CFI is to combine the literature on binary level provenance tracking [130] to augment checks being performed by existing CFI solutions on calls, jumps, and returns.

4.3 THREAT MODEL

We use the same threat model as works on CFI based defenses. That is, we assume the attacker can read and write from arbitrary memory addresses. This can be achieved via

common exploits. We assume that a CFI mechanism based on DSA points-to analysis is being used on the binary and that an efficient shadow-stack is being used. The enforcement mechanism presented in CCFI [131] can be used to achieve efficient enforcement of a fine-grained CFI policy and shadow-stack.

Our work focuses on generating capabilities for *select memory buffers*, not primitive types. Specifically, attacks like those presented in Control Flow Bending [25] and Control Jujutsu [26] can be eliminated using the capabilities produced by our system. We are not considering DOP attacks like those presented in [132].

4.4 SYSTEM OVERVIEW

Our goal is to augment CFI with instruction level memory access capabilities to mitigate Control Flow Bending and Control Jujutsu attacks. In this chapter, we explore creating these capability lists for sensitive memory regions. We leverage a data provenance based approach to produce instruction to memory access capability lists. Data provenance requires tagging all memory writes, tracking memory access throughout program execution, and finally, halting execution when an access deviates from a trusted data provenance graph for a given buffer.

Direct application of data provenance for security poses a number of challenges. First, tagging all memory writes imposes a large runtime overhead both on the CPU and in terms of memory bandwidth. Second, on large code bases, the number of unique tags required to build real world Data Flow Graphs (DFGs) explodes, either decreasing the accuracy of run time enforcement through approximations or further increasing runtime overheads. Third, in order to halt the program on a violation, the enforcement mechanism needs to be placed at every memory write. Provenance enforcement requires full DFG traversal for every write, further decreasing performance. Finally, a trusted source of provenance data is needed.

Our design addresses these challenges by making careful choices about which specific data

buffers are tagged and when checks are performed. Specifically, we classify certain buffers as *sensitive memory* and suggest performing checks at a *sensitive use*.

4.4.1 Identifying Sensitive Memory

Identifying which memory buffers are left vulnerable when combined with CFI is necessary to limit the overhead of a combined memory safety + CFI approach. We define sensitive functions as those that can expand the micro-service's subject domain. From Chapter 3 we know that system calls are used to create relationships with other system subjects. This implies that memory regions that can be used to influence the behavior of system calls are high-value targets. In fact, these are the exact memory regions that are the target of Control Flow Bending or Control Jujutsu attacks. "Sensitive usages" as defined in Table 4.1 are functions whose memory arguments, when corrupted, lead to policy violating subject-domains. Based on the discussion on relationships in Chapter 1, these are `libc` functions and "sensitive memory" consists of the arguments to these functions.

4.4.2 Building Provenance Graphs

Having identified sensitive memory regions, a trusted provenance graph is needed so that we can build capability lists. We have a number of options we can use to generate a provenance graph for a given buffer. We could use alias analysis to determine which data structures can be accessed by which functions. In large code bases such as Nginx, even the best alias analysis (DSA) leads to an explosion in the DFG, making this approach untenable. Alternatively, we could use symbolic execution and taint tracking to determine a DFG for a given memory region. Existing symbolic execution and taint tracking methods focus on tracking user input and the flow of that input under *normal* application execution. We want to track internal data structures that are often *never* modified by an end-user during a normal application flow. We need a reliable method to determine a DFG for arguments to internal program functions.

Instead, we rely on test suites and dynamic instrumentation to collect an accurate and minimal DFG from which instruction level capabilities can be built. We instrument the compiler to transparently inject a provenance engine to record this data during an offline phase. This data collection phase only has to be run once and can leverage high coverage test suites that are common in application development today.

4.5 IMPLEMENTATION

Our implementation leverages the LLVM compiler infrastructure [133] which provides a rich Intermediate Representation (IR) that can be modified with custom passes. A pass operates on a unit of compilation (module, function, or code block) taking IR as input and producing modified IR as output. Our provenance pass inserts a provenance engine by instrumenting all memory writes (`store` instructions in LLVM-IR) and memory allocation and free events. At a sensitive use, the provenance graphs for the sensitive memory being used are logged. This pass produces a binary to be used in an offline data collection phase. To ensure completeness of the provenance graphs, we also instrument "store-like" instructions and functions. These include functions such as `memcpy` and `memset` that write to regions of memory. In LLVM, setting a `struct` variable to the value of another `struct` is transformed into a call to an LLVM implementation of `memcpy`, not a `load + store` combination. Extending tracking to include "store-like" operations captures this behavior. We also extend the notion of store-like instructions to include system calls that write to userspace memory, the implementation of which is described in Section 4.5.4. We must also track `free` operations to ensure that a sensitive use is not vulnerable to a Use-after-Free vulnerability. Tracking `free`'s also allows us to prune provenance graphs to not include provenance from other allocations at the same address, a problem that is particularly relevant when tracking provenance of stack memory addresses as they have a high rate of reuse.

4.5.1 Trusted Provenance Graph Generation

We assume that the developer has access to a high coverage test suite for the application being instrumented. For Nginx, a Selenium [134] test suite could be used for a functional test of a Wordpress deployment, allowing an instrumented binary to collect trusted provenance graphs for sensitive memory.

We target infrastructure software; these types of packages are often written in low level languages and are vulnerable to issues stemming from the lack of memory safety. Applications like web servers such as Nginx and Apache and Key/Value Stores like Redis and Memchached are the cornerstone of many public facing application deployments and highlight the micro-service approach to deployment. Functional test suites of the applications built on top of these services exercise the functionality required to run on top of these platforms.

We define sensitive memory regions as the memory containing the arguments being passed to sensitive functions. As discussed in Section 4.4.1, we choose these to be `libc` functions for now as these functions are the target of both attacks able to circumvent state of the art CFI techniques and impact the service's relationship with out system-level subjects. We could easily extend this definition to include arguments to more functions or variables being modified within a loop.

To track memory operations for provenance collection, we leverage LLVM's rich intermediate representation, allowing us to distill distinct memory operations and instrument them directly. In this way, we can leverage provenance as a method for trusted capability generation while exploring efficient enforcement mechanisms of the capability sets.

4.5.2 Provenance Event Types

We define four types of provenance events that are collected for a given memory address *addr*.

Events types are as follows:

- *Use* (U) - A use of an address $addr$. For our purposes, a use is defined as an invocation of a sensitive function for which $addr$ is an argument. *Attributes:* ID, $addr$, PID.
- *Allocation* (A) - An allocation returning an address $addr$. The program being instrumented may allocate memory on the heap or stack. The C runtime may also allocate memory for program arguments and environment variables. Memory is also allocated for literals in the `.data` region by the C runtime. *Attributes:* ID, $addr$, $size$, $location$ (`HEAP`, `STACK` or `DATA`), PID.
- *Free* (F) - A free event. Free events are explicit for Heap memory and implicit on function returns for Stack memory. *Attributes:* ID, $addr$, PID.
- *Store* (W) - A store event during which there is a write to an address $addr$. *Attributes:* ID, $addr$, PID, ($source_addr$ - optional).

Note that all events have an ID attribute. This ID must be unique for all event instances logged to ensure the provenance building algorithm is complete as we discuss below.

4.5.3 Instruction & Function Tracking

Allocation and free events are tracked in a variety of ways. Stack allocations are tracked by instrumenting all `alloca` instructions. When a function returns, we insert `free` events for every address `alloca`'d during function execution into the provenance database. For heap memory, we instrument the `malloc` and `memalign` instructions. We have not encountered any sensitive memory regions lacking allocation events, but we can easily extend our method to include other memory allocations such as anonymous mappings returned from the `mmap` system call. Additionally, we have to consider allocations before a program's `main` function is invoked. Namely, the memory for the `environ`, `argc`, and `argv` symbols. The location and initialization of the memory pointed to by these symbols is implementation dependent [135].

To ease adoption, we avoid tracking memory operations in the target system's `libc`. Instead, we create allocation and store events for these symbols, if defined, upon invocation of `main`. Internal memory allocation functions being used for memory pools require additional effort on the developer's part. Using pool allocators is common in the infrastructure software we aim to harden. To work around this, we allow the developer to provide signatures of internal memory allocation and free functions; this is standard practice for instrumenting applications that use more than system calls for memory management [136]. We do not currently support hardening of JIT compilers or dynamically managed environments.

Memory writes can be tracked by instrumenting LLVM's `store` instruction. The instruction has three arguments, a *destination* pointer, a *source* variable, and the *size* of the memory being moved. The *destination* is always a pointer; the *source* can be a basic type or a pointer. We log all pointer arguments and the size so that we can track the entire provenance graph of a given buffer. When *size* is greater than the size of basic types, LLVM lowers the `store` operation into an LLVM specific `memcpy` implementation. We extend our definition of store events to include store-like function calls to track these store operations. We also extend our definition to include `libc` function calls that match the *destination*, *source*, *size* signature, such as `strcpy`, and treat these function invocations as store events as well. Just as we did with memory allocation and free events, we allow the developer to provide signatures of functions that re-implement `memcpy`/`store`-like operations.

Every function or instruction type must be assigned an ID that is unique and consistent across builds. For example, all `alloca` instructions are assigned unique ID's. These ID's can be used during an enforcement pass to limit the amount of instrumentation required to enforce a given memory modification capability set. This ID should not be confused with the runtime event-ID assigned to each event. The event-ID is unique to all events in the system and is used to provide *ordering*, but instruction-ID's are used to identify the exact instruction, $\mathcal{I}$ used to modify a sensitive memory region, $\mathcal{M}$. This information allows

us to build the fine-grained capability lists discussed in Section 4.1. We build a pruned list of capabilities that can be used for an *Enforceable Provenance Graph* as described in Section 4.5.5.

4.5.4 Store-Like System Calls

We have defined store-like events to mean any instruction or function that can store data in memory. This means that we must create store events for every system-call that writes to userspace memory. Consider the system-call `getsockopt` in which the kernel writes socket-options to a `struct` that lives in userspace. The `read` and `recv` system calls have similar behavior. In every case, the kernel writes data to a buffer that has been allocated and is manged by the userspace application. We want our provenance engine to be as widely-applicable as possible. For this reason, we cannot require custom versions of `libc` or leverage kernel modifications to track these store events that occur in code outside of the application. To overcome this, we have integrated function signatures for store-like operations that occur in the kernel or `libc` functions and they are logged immediately after the invocation of the store-like function. In doing so, we had to handle additional layers of complexity stemming from macro expansion that mangles `libc` function names. We must also gracefully handle errors that the application developer may or may not have handled internally. For example, a `recv` call may return -1 if the `socket` is interrupted during reading. In this case, we do not create an entry in the provenance database.

System calls have a variety of ways of indicating the length of the userspace buffer being passed into the kernel. Some calls leverage `struct`'s for which the size is known at compile time. We have created an abstraction that allows us to easily handle system call cases as we encounter them. Error handling using an abstraction to allow for easy addition of store-like system calls as they are encountered has to contend with the fact that different system calls use different basic types for error reporting. Developers must also specify a list of internal functions that might match a given system call that should not be treated as such. For

example, the `ngx_enable_accept_events` should not be treated as an `accept` system call. On the other hand the function `accept4` should be treated like an `accept` system call just as `llvm.memset.p0i8.i64` should be treated as a `memset`. We require the developer to list functions not to be treated as system calls so that we can easily support the variety of naming schemes used when creating system call wrappers in `libc` and LLVM. We have eased this process by creating a tool showing the developer a list of functions that match system calls.

4.5.5 Enforceable Provenance Graphs

The provenance graph for a given *addr* is represented as a tuple $Prov(addr) = \langle A_{addr}, \mathcal{S}_{addr}, F_{addr} \rangle$ where:

- A_{addr} is the allocation event for the address.
- $\mathcal{S}_{addr}$ is an ordered list of tuples numbered from 0 to n of the form: $[\langle W^0_{addr}, Prov(source_addr^0) \rangle ... \langle W^n_{addr}, Prov(source_addr^n) \rangle]$. For a given tuple x, *addr* must be in the range $[destination^x,\ destination^x + size^x)$ for every store operation W^x_{addr} where $source_addr^x$, $destination^x$, and $size^x$ are elements in the W^x_{addr} event tuple as defined in Section 4.5.2. This tuple list contains every store event that wrote to *addr* and the provenance of every *source* address in copy events. This recursive definition provides the entire history of a single write event. The base case is a write event that did not have a source address. More on this below.
- F_{addr} is the free event for the address.

An *enforceable provenance graph* must not include the final element in the tuple, namely the free event F_{addr}. Verifying this constraint at runtime, during provenance logging, grantees no Use-After-Free bugs in addresses pointing to sensitive memory.

The provenance engine is robust, even when instrumenting complex production ready software packages. This requires overcoming challenges with multi-threaded applications, shared-memory, and numerous types of memory allocators.

Existing literature on memory protection assumes singled threaded applications and a simpler model of C than that used in practice [123, 124]. In particular, these works assume that `malloc` is the only way in which memory can be allocated. In the programs we evaluated, numerous system calls are being used including `mmap`, `malloc`, and `memalign`. Each of these calls must be instrumented in order to ensure that an allocation event, A_{addr}, is in the database at the time of use, U_{addr}. For now, we are tracking `malloc` and `memalign` as they are the only allocators being used for sensitive memory, easily verified via the lack of missing allocation events for sensitive addresses. For each new allocation event type, we must assign a unique identifier to the instruction type so that the specific allocation invocation can be instrumented during an enforcement phase. We discuss the challenges with multi-threaded applications and our robust memory layout in Sections 4.5.6 and 4.5.7 respectively.

The algorithm for parsing provenance engine output and translating the raw event stream into individual provenance graphs for each usage must ensure the invariants outlined above hold. These invariants drive both the parsing algorithm and the decision about which instructions or function invocations need logging and where to place compile time instrumentation during our provenance collection pass as defined in Section 4.5.3. The full algorithm for querying the provenance database for *enforceable provenance graphs* while handling the full complexity of possibilities allowed by C is show in the Breadth-First-Search Algorithm 4.1, described using the terms in Table 4.2.

Table 4.2: Terms used in Provenance Algorithms

	Definition of Term
$forkEvents$	Fork Event Table, indexed by Event ID
$freeEvents$	Free Event Table, indexed by Event ID
$allocEvents$	Allocation Event Table, indexed by Event ID
$storeEvents$	Store and Copy Event Table, indexed by Event ID
$storeEventsRev$	Store and Copy Event Table, indexed by Starting Address

The algorithm has five main parts: *(i)* Locating the allocation for the sensitive memory region (this step additionally checks to ensure the address is not vulnerable to a Use-After-Free vulnerability), *(ii)* searching for all possible store events for the given address that occur after the allocation event, *(iii)* filtering out events that did not occur within the relevant processes (a full description of why this is necessary is in Section 4.5.6), *(iv)* masking out store(-like) events, and finally, recursively following source addresses in `copy` events. In step *(iv)* we mention event masking. Consider an an event with ID y that writes 1 byte to an address x. If another event occurs in the future with ID $y + t$ writing 1 or more bytes to address x, the second event "masks" the first because any change from the first event will not be visible to the application. We only perform single-event masking, meaning only events that write out the same number of bytes to the same addresses will mask out other events. In our tests, this is sufficient to mask out events and reduce the overhead of recursing. Future work can explore combining events to produce masks. Consider a `memset` event that stores to an entire buffer. The event could be masked by two subsequent events each of which writes to half of the buffer. Note that `getForkEventIDs` referenced in Algorithm 4.1 is described in full detail in Section 4.5.6.

Algorithm 4.1 getProvFor

```
1: function GETPROVFOR(address, event_id, pid, size)
2:     allocation ← getAllocationFor(address, event_id, pid)
3:     forkEvents ← getForkEvents(pid, event_id)
4:     possibleStores ← storeEventsRev[address]
5:     unfilteredStores ← {}
6:     for event in possibleStores do
7:         startingAddr ← event.startingAddr
8:         endAddr ← event.startingAddr + event.size
9:         if startingAddr ≤ address then
10:            if endAddr >= address then
11:                unfilteredStores[event.id] ← event
12:            end if
13:        end if
14:        if startingAddr > address then
15:            if startingAddr < endAddr then
16:                unfilteredStores[event.id] ← event
17:            end if
18:        end if
19:    end for
20:    filtered ← filterPreAllocation(unfilteredStores, allocation)
21:    forkAware ← filterPids(filtered, allocation, forkEvents)
22:    events ← maskEvents(forkAware)
23:    for event in events do
24:        if event.type ≡ COPY then
25:            event.prov ← getProvFor(event.startingAddr, event_id, pid,
                                       size)
26:        end if
27:    end for
28: end function
```

4.5.6 Cross-Process Memory Events

We also have to consider challenges to provenance tracking across multiple processes. To enable cross-PID provenance we must add an additional fork event to the list of events being tracked as defined in Section 4.5.2:

- *Fork* (K) - A fork system call. A tuple containing an event ID, PID (the *parent_pid*), and *child_pid*.

Additionally, the events defined in Section 4.5.2 must be augmented with the PID in which the event was generated. Consider a simple multi-threaded program in which a parent process allocates memory, writes to it, forks, and the child subsequently uses the memory. Tracking fork events allows us to easily identify cross-PID allocation and store events for a given address. This is common in production software. Consider the `fork() + execve()` pattern in which the parent forks and the child process executes a new binary. The arguments to `execve` are allocated and setup by the parent process.

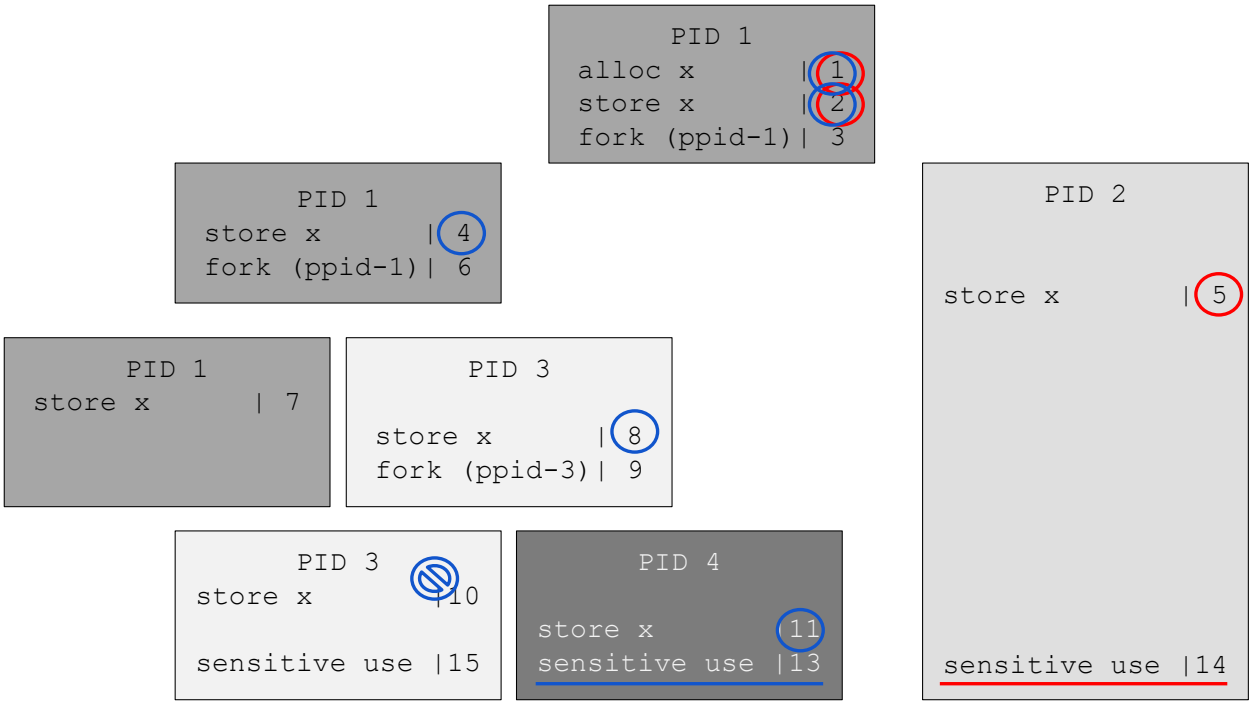


Figure 4.1: Provenance Across Fork Events

Figure 4.1 shows the full complexity of cross-PID memory events. Each box in the figure represents an instruction stream in a given process; on the left is the instruction type coupled with an ever growing event-id on the right. Consider a single process that allocates an address x, followed by various store(-like) events across processes after the original process has forked. Sensitive usages of the address x at events 13 and 14 highlight challenges in querying for cross-fork events. The *enforceable provenance graph* for sensitive use 14 is shown in red. It is fairly easy to reason about the events making up the graph. The situation is more

complex when parsing the provenance data to produce the graph for address x and event 13, shown in blue. Events span multiple processes and must filter out irrelevant event-ID's. For example, PID-3 has a store event with ID 10 and which is less than the ID of the sensitive use, 13, and in a parent process. Despite being in a parent, event 10 should not be included because it occurs after the fork event 9. Algorithm 4.2 is the algorithm for identifying the list of fork event-ID's that should be used as bounds when querying for store-events in each parent-PID.

Algorithm 4.2 getForkEventIDs

```
1: function GETFORKEVENTIDS(pid, event_id)
2:     forkIds = {}
3:     childPid ← pid
4:     childId ← event_id
5:     for i = forkEvents.size() − 1, i >= 0, i − − do
6:         event ← forkEvents[i]
7:         childPid_i ← event.childPid
8:         currId ← event.eventId
9:         if childPid_i ≡ childPid && currId < childId then
10:            childPid ← childPid_i
11:            childId ← currId
12:            forkIds[childId] ← childPid
13:        end if
14:    end for
15:    return forkIds
16: end function
```

4.5.7 Runtime Provenance Engine Memory Layout

The runtime memory layout of our provenance engine is dictated by the need to handle cross-PID tracking while allowing unique ID's be generated for events as they occur. To do this, we produce a transaction system based on shared-memory for writing into the provenance database. The engine for writing into the database lives in shared memory, along with a locking mechanism and enough memory to store event transaction data before it is written to the provenance database. This design is shown in Figure 4.2. The instrumented

application interacts with the provenance engine through an Inter-Process-Communication (IPC) mechanism leveraging a semaphore in shared-memory. The LLVM-pass inserts a function invocation to allocate and initialize the shared-memory region before any other application code runs. This ensures the shared-memory is allocated and the address is accessible to any threads that the application may create throughout execution.

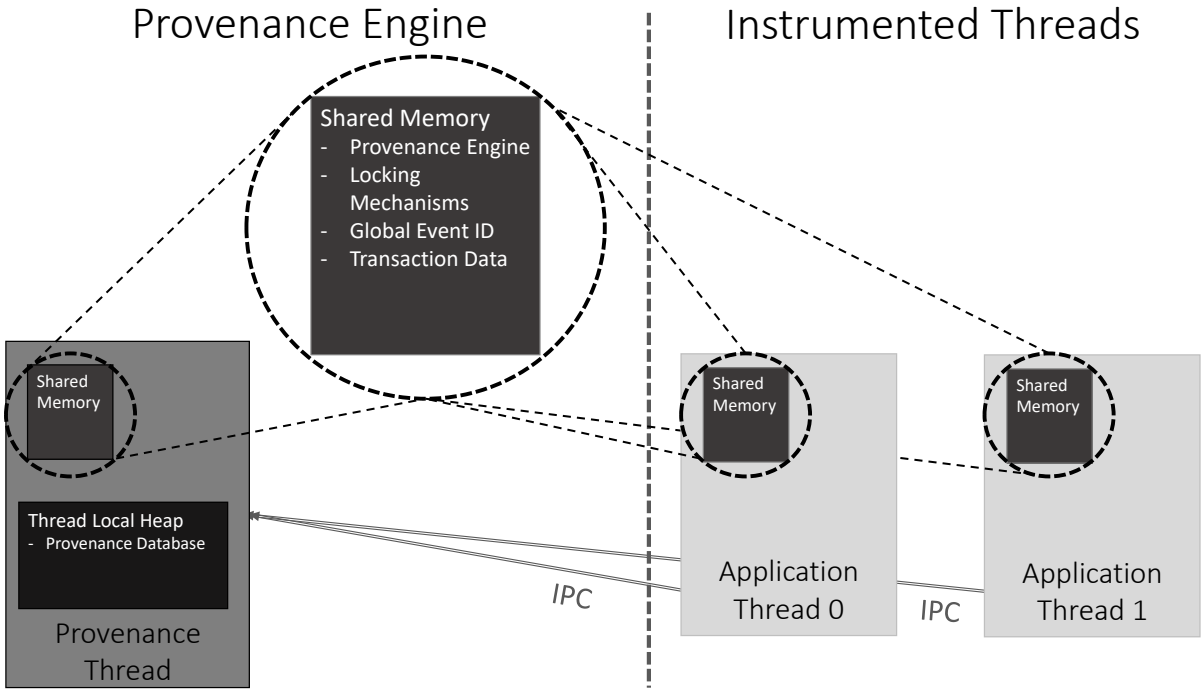


Figure 4.2: Provenance Engine Memory Layout

Event insertion into the provenance database at runtime occurs through a series of function calls, one for every event type described in Section 4.5.2. Each function's behavior is very similar: *(i)* Get a handle to the shared-memory region, *(ii)* grab the global lock, *(iii)* write data to transaction `structs`, *(iv)* signal the provenance thread via a semaphore. At this point, the lock is held and no other event can be written to the provenance database until the transaction is completed by the provenance thread.

The provenance thread waits on a semaphore in shared memory. Upon grabbing the semaphore, the provenance thread is responsible for taking one of a few actions relating to the provenance database. It can either write transaction data for an event or run Algorithm 4.1

to produce the enforceable provenance graph for a given sensitive use. The provenance database itself lives on the heap in the provenance thread; all access to this database must use IPC between application threads and the provenance thread. Once the action completes, the provenance thread unlocks the lock that was grabbed at the beginning of the transaction, allowing other events or actions to be taken.

4.6 EXAMPLE PROVENANCE GRAPHS

We have evaluated our provenance engine on the Nginx webserver. Nginx uses a master-worker process model for handling requests. All requests are processed within a worker thread while the master thread is responsible for reading the configuration file and spinning up workers. This is only one threading model an application might use; others could use threads that split work evenly across threads. In any case, the forking methods described in Section 4.5.6 and the memory layout discussed in Section 4.5.7 provide an abstraction that allows us to easily reason about provenance regardless of the design of the instrumented application.

We have also provided signatures for the store-like copy operation in Nginx that re-implements `libc` `strncpy` functionality, namely the `ngx_strncpy` function. Doing this allows the provenance graphs to track the relationship between source and destination addresses passed to the function, information that would otherwise be lost. Additionally, in future work we will leverage these provenance graphs to enforce instruction level capabilities. Treating custom copy functions as store-like operations means that enforcement can be placed after the function invocation instead of after the single `store` operation that occurs in a tight-loop within the `ngx_strncpy` function itself which can reduce overhead.

Nginx was started with the command `nginx -c /etc/nginx/nginx.conf -g "daemon off;"`. The provenance graph for sensitive use 96 with event-ID 11508 is show in Listing 4.1. For each sensitive use, we log the provenance of any memory buffer arguments. In this case,

there is only a single memory buffer holding a \0 terminated C-string which is 22 bytes. Each argument is given a number by the provenance engine, so this first and only argument is numbered 0. A name may appear after the number identifier if a variable name is still present within LLVM at compile time. We can see that this argument was allocated in the middle (732 bytes in) of a large allocation from a `memalign` call. The `memalign` has been assigned a unique ID, `512:1:1` indicating the function, block, and individual instruction at which the `memalign` occurs in the LLVM IR. Future work can leverage this ID to enforce the capability lists at runtime. The memory region was written to by a single copy instruction of `CUSTOM` type, indicating that the store-like operation stems from a developer provided function signature. Finally, the source of the copy event was allocated by the C-Runtime before the invocation of `main`. Specifically, the source was the 3rd argument to the micro-service, `argv[2]`, which was the name of the configuration file used when starting Nginx. It is clear to see that this is the open call running in the master thread to open the configuration file as Nginx starts. This defines the proper data flow for sensitive use 96.

Listing 4.1: Example `open` Provenance Graph

```
Event id: 11508 <open64>: 96 PID: 506

Event id: 11509  0/: <0x10c995c> / Size: 22 PID: 506

Allocation: Event ID => 9787 / Starting Addr => 10c9680 / Size => 16384 /
Pid => 506 / Offset for Arg Addr => 732 / KIND => MEMALIGN / MEMALIGN ID =>
512:1:1

COPY: Event ID => 10070 / DST Address: 10c995c / Size: 22 / SRC Addr:
7ffeea9b7e81 / Pid: 506 / Copy ID: 292:11:1/ Location: HEAP / Kind: CUSTOM

        Allocation: Event ID => 3 / Starting Addr => 7ffeea9b7e81 /
        Size => 21 / Pid => 506 / Offset for Arg Addr => 0 / KIND => CRT /
        CRT ID => argv[2]
```

We have also listed another provenance graph from the `sigsuspend` sensitive use in Listing 4.2. This listing highlights the need to treat certain system calls that write to userspace

memory as store-like events. The final event in the provenance graph is a call to `sigprocmask` that modifies the `struct` buffer that is then passed into `sigsuspend`.

Listing 4.2: Example `sigsuspend` Provenance Graph

```
Event id: 312400 <sigsuspend>: 206 PID: 477

Event id: 312401  0/set: <0x7fff074c9800> / Size: 128 PID: 477

Allocation: Event ID => 311417 / Starting Addr => 7fff074c9800 / Size => 128 /
Pid => 477 / Offset for Arg Addr => 0 / KIND => Alloca / Alloca ID => 609:1:8

STORE: Event ID => 311427 / DST Address: 7fff074c9800 / Size: 8 / SRC Addr: 0 /
Pid: 477 / Store ID: 609:1:1 / sigprocmask / Location: STACK / Kind: SYS_CALL
```

The examples listed in this section demonstrate our robust provenance engine. We are able to track the source of memory modifications throughout program execution and log them so they can be used as capability lists.

4.7 RELATED WORK

In this section we discuss the limitations of CFI methods which highlight the need for instruction level capability lists. We also discuss work on runtime provenance engines that have traditionally been used in audit systems.

4.7.1 Control Flow Integrity

CFI [122] attempts to mitigate ROP [115] and JOP [116] attacks by enforcing a CFG. These approaches have two steps: 2) CFG generation and 2) CFG enforcement. Early CFI approaches generated a coarse-grained CFG that categorized code pointers into only two distinct categories: one for functions whose address is taken and another for return-address locations (code addresses immediately following a call-site) [122]. Approaches to CFI have been steadily adding more ways to classify code addresses. CCFIR [125] has up to 3 different

classifications. More recent approaches can support up to 2^{80} unique classes [137], but the authors do not discuss how an effective CFG could be built that supports such a high degree of unique classification. The coarse grained approach to generating CFG's is ineffective as an attack deterrent as highlighted by numerous attacks [126, 138, 139]. Type based CFG generation can be more effective at reducing the number of gadgets available to attackers. Type based solutions require application level changes, thus fall back on function arity [140], limiting the number of classifications available to call-sites and indirect branches.

The attacks on coarse-grained CFI [126, 138, 139] necessitated the development of more fine-grained approaches. Fine-grained approaches must address two concerns: they should be able to support a large number of classifications and should be based on more precise static analysis; preferably, they should also incorporate dynamic run-time information to augment enforcement decisions. Practical Context-Sensitive CFI [141] augments custom static analysis performed on a binary with run-time information to reduce valid code-address targets. The authors further evaluate a hypothetical deployment of ideal static analysis based on DSA [127] for a further reduction in gadgets. Cryptographically Enforced Control Flow Integrity [137] introduces a hardware accelerated enforcement mechanisms that supports up to 2^{80} different classifications/labels and works in run-time information such as the location of the pointer holding the code-address, but only suggest leveraging existing CFG generation methods.

4.7.2 Provenance

Traditionally, provenance has been used as a runtime audit system [130, 142, 143]. Bates *et al.* focus on instrumenting the Linux kernel to produce W3C compliant provenance graphs, capturing the relationships between system level subjects. Our provenance engine works at a more fine grained level by determining provenance at the instruction level. CamFlow is another provenance engine that works as a modification to the Linux kernel [143]. We see these works as complimentary. Our system could be run along side these to augment their

logs with fine-grained provenance data. Spade is a generic system for storing provenance from a variety of sources [130]. Again, our provenance engine for C applications could compliment their work by providing an additional source of provenance data for their distributed provenance engine.

4.8 CONCLUSION

In this chapter we have presented a provenance engine to be inserted into micro-services via a modified compiler. This allows us to instrument and track provenance of memory that will impact the relationships micro-services create with other subjects in cloud computing systems as shown in Figure 1.1 in Chapter 1. Using our model of security, we are able to limit the amount of memory that gets classified as being "security sensitive" to only the memory that has the ability to influence the service's subject-domain, namely, memory used to modify the behavior of system calls.

To produce these fine-grained capabilities, our provenance engine must be able to track the source of memory allocation, modification, and copy events. To support the complexities of real world software, we must allow developers to extend the notion of individual event types to include developer defined functions such as internal pool allocators or custom versions of `memcpy`-like functions.

Furthermore, we must track memory modification events that occur outside of the application itself. Specifically, memory can be modified by the kernel by passing userspace buffers to system-calls. To support treating these calls as store events, we instrument the system calls that write to userspace memory so that their writes appear in the provenance database. Finally, we have to handle the complexity steaming from cross-fork allocation events.

Having handled these event types, we are able to produce the fine-grained capability lists linking individual instructions to the relationships they are capable of influencing.

Chapter 5: Summary

In this dissertation, we have provided a novel approach to micro-service security by considering subjects and relationships from every layer of the stack. We apply Take-Grant across every layer simultaneously, allowing us to consider passive information transfer stemming from stateful hardware, trustworthy relationship monitoring at the system-call boundary, and fine-grained relationship modification at the instruction level.

5.1 CROSS-LAYER TAKE-GRANT

We leverage a Take-Grant model of security that allows for straightforward reasoning of capabilities. We apply this model to micro-services in a cloud environment and leverage it in a variety of ways outlined below.

5.1.1 Passive Attacks

We start in Chapter 2 by extending the notion of a "subject" to include stateful hardware resources that until recently have not been considered a point of information flow. In this dissertation, we focus on information flow stemming from shared caches, but the technique can be applied to other hardware resources. Having added the cache to our model of security, we can begin to look for ways to apply capabilities to limit access to the shared resources.

We introduce the notion of security domains in which processes belonging to a single organization or security-level within an organization can be grouped. We then introduce a cache-access capability that can only be held by a single security domain at a time. To reduce the impacts of of enforcing such a capability on a large shared cache, we leverage Intel's CAT to partition the LLC and then apply the capability within a single isolated region consisting of an isolated cache region and any number of logical processors.

We enforce the cache-access capability through the use of strict-co-scheduling of processes

belonging to the security domain holding the capability in the isolated region. Transfer of the capability to another domain requires revoking the cache-access capability, pausing execution in the isolated region, then performing state cleansing to remove residual information before granting the cache-access capability to another security domain.

Our approach to adding capabilities to access shared-caches highlights how capabilities can be leveraged to eliminate information flow due to stateful hardware. Passive attacks will continue to remain a threat to cloud computing until capabilities can be effectively applied to all shared hardware. We encourage hardware vendors to expose other MSRs allowing explicit operating system control over these stateful resources.

5.1.2 Active Attacks at the System Level

In Chapter 3 we extend active monitoring mechanisms to be more resilient when generating logs of relationship events between micro-services and other system level subjects. Subjects at this level include files being accessed and binaries being executed. Leveraging a novel probe insertion mechanism, we are able to produce a trustworthy log.

Leveraging the trustworthy log of system level relationships, we are able to create both a policy recording mechanism and IDS that determines when system-level capabilities have been violated.

Monitoring relationships at the systems-level exposes a number of attacks, such as those trying to read system files that could expose confidential user information. Even so, relationship monitoring at this level dictates that attacks must first go outside of the micro-service before it can be detected. For earlier detection of active attacks, we look toward fine grained capability mechanism for cloud application.

5.1.3 Fine Grained Active Attacks

To address the limitations of coarse grained detection of active attacks, we look at how relationships are influenced within an application in Chapter 4. Relationships are built using

system-calls; instructions that can modify the behavior or the specific subject the system-call acts on become part of the security model.

We leverage a custom provenance engine to determine instruction level provenance of memory that is being passed into system calls. These memory regions define the files that are opened or executed along with the sockets that are opened. These are only examples of the kind of behaviors that the arguments to these functions can control. It is clear that system-level relationships are governed by these memory regions, thus the instructions that modify these regions are a logical point at which to apply capabilities.

By tracking the provenance for these memory regions, we can produce instruction level capabilities to limit an attackers ability to hijack control flow and create untrusted relationships between the micro-service and other subjects on the system. Chapter 4 highlights the challenges with producing capability lists at such a fine granularity. It is clear that for robust security, instruction level capabilities must be taken into account.

5.2 CONCLUSION

In order to have the best possible security in cloud environments, a model of security must include subjects from every layer of the application stack. These include resources as opaque as hardware caches, as coarse grained as system-level function calls, and as fine grained as instruction level memory access capabilities. Having a robust model for security allows researchers and practitioners to more easily reason about the relationships that are created between subjects in the system. Having identified the source of these relationships, capabilities can be defined to govern them. We have highlighted this method throughout this dissertation. Moving forward, security must be holistic and consider a wide-range of resources and the complex relationships between them.

References

[1] [Online]. Available: https://www.cvedetails.com/vulnerability-list.php?vendor_id=11727&product_id=&version_id=&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=7&cvssscoremax=0&year=0&month=0&cweid=0&order=1&trc=1&sha=00b38130ead426f27e5c1e857a4d1327e3313481

[2] Y. Sun, G. Petracca, X. Ge, and T. Jaeger, "Pileus: Protecting user resources from vulnerable cloud services," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 52–64.

[3] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[4] P. Mell, T. Grance et al., "The nist definition of cloud computing," 2011.

[5] A. K. Talukder, L. Zimmerman et al., "Cloud economics: Principles, costs, and benefits," in *Cloud computing*. Springer, 2010, pp. 343–360.

[6] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)," *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[7] [Online]. Available: https://aws.amazon.com/ec2/

[8] [Online]. Available: https://cloud.google.com/compute/

[9] [Online]. Available: https://azure.microsoft.com/en-us/services/virtual-machines/

[10] [Online]. Available: https://www.digitalocean.com/

[11] [Online]. Available: https://www.rackspace.com/openstack/public

[12] [Online]. Available: https://www.openstack.org/

[13] [Online]. Available: https://www.pcisecuritystandards.org/document_library?category=pcidss&document=pci_dss

[14] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014.

[15] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2015.

[16] [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[17] [Online]. Available: https://12factor.net/

[18] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[19] A. K. Jones, "Protection in programmed systems." CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1973.

[20] A. K. Jones, R. J. Lipton, and L. Snyder, "A linear time algorithm for deciding security," in *Foundations of Computer Science, 1976., 17th Annual Symposium on*. IEEE, 1976, pp. 33–41.

[21] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *Journal of the ACM (JACM)*, vol. 24, no. 3, pp. 455–464, 1977.

[22] M. Bishop and L. Snyder, "The transfer of information and authority in a protection system," in *Proceedings of the seventh ACM symposium on Operating systems principles*. ACM, 1979, pp. 45–54.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish et al., "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[24] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix." in *USENIX Security Symposium*, vol. 46, 2010, p. 2.

[25] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity." in *USENIX Security*, vol. 14, 2015, pp. 28–38.

[26] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[27] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[28] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on aes," in *International Workshop on Selected Areas in Cryptography*. Springer, 2006, pp. 147–162.

[29] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 490–505.

[30] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[31] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[32] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[33] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[34] Docker. [Online]. Available: https://www.docker.com

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 990–1003.

[36] V. Varadarajan, T. Ristenpart, and M. M. Swift, "Scheduler-based defenses against cross-vm side-channels." in *Usenix Security*, 2014, pp. 687–702.

[37] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd acm sigsac conference on computer and communications security*. ACM, 2015, pp. 1595–1606.

[38] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 871–882.

[39] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proc. of the 25th Int. conf. on Compiler Construction*. ACM, 2016, pp. 110–120.

[40] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud." in *USENIX Security symposium*, 2012, pp. 189–204.

[41] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[42] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 494–505.

[43] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on.* IEEE, 2008, pp. 83–93.

[44] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 118–129, 2012.

[45] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop.* ACM, 2011, pp. 41–46.

[46] P. Li, D. Gao, and M. K. Reiter, "Mitigating access-driven timing channels in clouds using stopwatch," in *Dependable systems and networks (DSN), 2013 43rd Annual IEEE/IFIP international conference on.* IEEE, 2013, pp. 1–12.

[47] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security.* ACM, 2009, pp. 77–84.

[48] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on.* IEEE, 2011, pp. 194–199.

[49] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd international conference on Parallel architectures and compilation.* ACM, 2014, pp. 381–392.

[50] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Security and Privacy, 2009 30th IEEE Symposium on.* IEEE, 2009, pp. 45–60.

[51] C. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, 1948.

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

[53] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie et al., "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE, 2015, pp. 20–37.

[54] J. V. Cleemput, B. D. Sutter, and K. D. Bosschere, "Adaptive compiler strategies for mitigating timing side channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[55] Intel, "Cache monitoring technology and cache allocation technology," https://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html, 2017, (Accessed on 06/08/2017).

[56] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on.* IEEE, 2015, pp. 171–172.

[57] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on.* IEEE, 2013, pp. 233–240.

[58] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches." in *USENIX Security Symposium*, 2015, pp. 897–912.

[59] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, vol. 2014, 2014, pp. 719–732.

[60] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack." *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

[61] J. R. Bulpin and I. Pratt, "Hyper-threading aware process scheduling heuristics." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 399–402.

[62] W. M. Hu, "Lattice scheduling and covert channels," in *Proc. 1992 IEEE Computer Society Symp. on Research in Security and Privacy*, May 1992, pp. 52–61.

[63] M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE Transactions on Cloud Computing*, vol. 2, no. 4, pp. 395–408, Oct 2014.

[64] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 37–48.

[65] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[66] C. Percival, "Cache missing for fun and profit." BSDCan, 2005.

[67] J. C. Bennett and H. Zhang, "Wf/sup 2/q: worst-case fair weighted fair queueing," in *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 1. IEEE, 1996, pp. 120–128.

[68] R. Love, *Linux kernel development*. Pearson Education, 2010.

[69] Intel, "Cache allocation technology," https://01.org/cache-monitoring-technology?page=1, 2017, (Accessed on 06/08/2017).

[70] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "Cain: Silently breaking aslr in the cloud," in *Proc. of the 9th USENIX conf. on Offensive Technologies*, ser. WOOT'15. Berkeley, CA, USA: USENIX Association, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831211.2831224 pp. 13–13.

[71] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," *2016 IEEE Symp. on Security and Privacy (SP)*, vol. 00, pp. 987–1004, 2016.

[72] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *Proc. of the 25th USENIX Security Symp.*, 2016.

[73] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proc. of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1972551.1972552 pp. 1:1–1:6.

[74] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proc. of the 41st Annual Int. Symp. on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665726 pp. 361–372.

[75] VMWare, "Security considerations and disallowing inter-virtual machine transparent page sharing," https://kb.vmware.com/s/article/2080735, May 2015, Accessed on 01/15/2018.

[76] N. A. Dadhania. [Online]. Available: https://lwn.net/Articles/472797/

[77] [Online]. Available: https://lwn.net/Articles/240474/

[78] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 827–838.

[79] M. Xu, L. T. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," 2017.

[80] D. Documentation, "Btrfs storage driver — docker documentation," https://docs.docker.com/engine/userguide/storagedriver/btrfs-driver/, 2017, (Accessed on 06/08/2017).

[81] D. Documentation, "Overlayfs storage driver — docker documentation," https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/, 2017, (Accessed on 06/08/2017).

[82] D. Documentation, "Aufs storage driver," https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/#how-the-aufs-storage-driver-works, 2017, (Accessed on 06/08/2017).

[83] Intel, "Intel 64 and ia-32 architectures software developers manual volume 2 (2a, 2b, 2c & 2d): Instruction set reference, a-z," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2017, (Accessed on 06/08/2017).

[84] A. Technology, "Amd64 architecture programmers's manual," http://support.amd.com/TechDocs/24593.pdf, 2017, (Accessed on 06/08/2017).

[85] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on.* IEEE, 2010, pp. 82–91.

[86] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring," in *9th USENIX Workshop on Offensive Technologies (WOOT 15).* Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

[87] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *30th Annual Computer Security Applications Conference on (ACSAC)*, 2014.

[88] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *USENIX Annual Technical Conference (ATC '07)*, May 2007, pp. 1–14.

[89] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic VM Dependability Monitoring Using Hypervisor Probes," in *Dependable Computing Conference (EDCC), 2015 Eleventh European.* IEEE, 2015, pp. 61–72.

[90] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference.* ACM, 2013, pp. 289–298.

[91] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 13–24.

[92] B. D. Payne, M. D. P. de Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Annual Computer Security Applications Conf. (ACSAC '07)*.

[93] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual Machine Introspection: Techniques and Applications," *2015 10th International Conference on Availability, Reliability and Security (ARES)*, pp. 676–685, 2015.

[94] L. K. Documentation, https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2017, (Accessed on 06/08/2017).

[95] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-based Out-of-the-Box Semantic View Reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[96] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "CloudSec: A security monitoring appliance for Virtual Machines in the IaaS cloud model," in *Network and System Security (NSS), 5th International Conference on*, 2011, pp. 113–120.

[97] J. H. Huh and A. Martin, "Trusted logging for grid computing," in *Third Asia-Pacific Trusted Infrastructure Technologies Conference (APTC '08).*, 2008.

[98] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 193–206.

[99] R. Perez, R. Sailer, L. van Doorn et al., "vTPM: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[100] "ab - apache http server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2017, (Accessed on 06/08/2017).

[101] Redis, "Redis benchmark utitlity," https://redis.io/topics/benchmarks, 2017, (Accessed on 06/08/2017). [Online]. Available: http://redis.io/topics/benchmarks

[102] L. Mint, "Main page - linux mint," https://linuxmint.com/, 2017, (Accessed on 06/08/2017).

[103] T. H. New, http://thehackernews.com/2016/02/linux-mint-hack.html, 2017, (Accessed on 06/08/2017).

[104] WPScan, https://wpvulndb.com/vulnerabilities/8209, 2017, (Accessed on 06/08/2017).

[105] Rapid7, "Wordpress ajax load more php upload vulnerability," https://www.rapid7.com/db/modules/exploit/unix/webapp/wp_ajax_load_more_file_upload, 2017, (Accessed on 06/08/2017).

[106] M. Crawford and G. Peterson, "Insider Threat Detection using Virtual Machine Introspection," in *Hawaii International Conference on System Sciences (HICSS '13)*.

[107] D. Kienzle, R. Persaud, and M. Elder, "Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection," *System Sciences (HICSS)*, 2010.

[108] T. Y. Win, H. Tianfield, and Q. Mair, "Virtualization security combining mandatory access control and virtual machine introspection," *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*.

[109] F. Lombardi and R. Di Pietro, "KvmSec: a security extension for Linux kernel virtual machines," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 2029–2034.

[110] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Symposium on Security and Privacy*. IEEE, 2012.

[111] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 297–312.

[112] A. P. Kosoresow and S. A. Hofmeyer, "Intrusion detection via system call traces," *IEEE software*, vol. 14, no. 5, pp. 35–42, Jan. 1997.

[113] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 461–470.

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

[115] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[116] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[117] P. Team, "Pax address space layout randomization (aslr)," 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[118] T. Durden, "Bypassing pax aslr protection," *Phrack magazine*, vol. 59, no. 9, p. 9, 2002. [Online]. Available: http://phrack.org/issues/59/9.html

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

[120] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 380–392.

[121] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 368–379.

[122] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[123] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[124] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[125] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

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

[127] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, 2007.

[128] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[129] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996, pp. 32–41.

[130] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 101–120.

[131] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[132] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 969–986.

[133] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[134] [Online]. Available: http://www.seleniumhq.org/

[135] ISO/IEC, "Iso international standard iso/iec 9899:2011 - programming language c. [working draft]," *Geneva, Switzerland: International Organization for Standardization (ISO)*, 2011. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[136] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813691 pp. 268–279.

[137] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - (CCS) '15*. ACM Press, 2015.

[138] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

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

[140] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm." in *USENIX Security Symposium*, 2014, pp. 941–955.

[141] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 927–940.

[142] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel." in *USENIX Security Symposium*, 2015, pp. 319–334.

[143] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 405–418.