CACHE-BASED SIDE-CHANNEL ATTACKS IN MULTI-TENANT PUBLIC CLOUDS AND THEIR COUNTERMEASURES

Yinqian Zhang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2014

Approved by:

Michael K. Reiter

Jan Prins

Ari Juels

Bryan Parno

Thomas Ristenpart

ABSTRACT

Yinqian Zhang: Cache-based Side-Channel Attacks in Multi-Tenant Public Clouds and Their Countermeasures
(Under the direction of Michael Reiter)

Cloud computing is gaining traction due to the business agility, resource scalability and operational efficiency that it enables. However, the murkiness of the security assurances offered by public clouds to their tenants is one of the major impediments to enterprise and government adoption of cloud computing. This dissertation explores one of the major design flaws in modern public clouds, namely insufficient isolation among cloud tenants as evidenced by the cloud's inability to prevent side-channel attacks between co-located tenants, in both Infrastructure-as-a-Service (IaaS) clouds and Platform-as-a-Service (PaaS) clouds. Specifically, we demonstrate that one virtual machine (VM) can successfully exfiltrate cryptographic private keys from another VM co-located on the same physical machine using a cache-based side-channel attack, which calls into question the established belief that the security isolation provided by modern virtualization technologies remains adequate under the new threat model in multi-tenant public IaaS clouds. We have also demonstrated in commercial PaaS clouds that cache-based side channels can penetrate container-based isolation by extracting sensitive information from the execution paths of the victim applications, thereby subverting their security. Finally, we devise two defensive techniques for the IaaS setting, which can be adopted by cloud tenants immediately on modern cloud platforms without extra help from cloud providers, to address side-channel threats: (1) for tenants requiring a high degree of security and physical isolation, a tool to facilitate cloud auditing of such isolation; and (2) for tenants who use multi-tenant cloud services, an operating-system-level defense to defend against cache-based side-channel threats on their own.

To my parents, who raised me to be a man of integrity and determination, encourage and support me unconditionally to pursue my dreams.

To my beloved wife, Ying, my biggest fan, who believes in me under any circumstances, cheers on my trivial achievements, and always stands beside me in the face of difficulties.

To my son, who is on his way to this beautiful world as of this writing. He brings joy, and makes my life more purposeful.

ACKNOWLEDGMENTS

I want to express my heartfelt gratitude to my advisor, Dr. Michael Reiter, for taking me under his wings, guiding me to mature professionally as a researcher, sharing with me his relentless optimism when facing research challenges, and offering me freedom and support in pursuing my own research interests. He has inspired me as a role model with his strong dedication and incredible enthusiasm in his work. He was the reason I chose to enter this graduate program as a graduate student; and he is the reason I choose an academic career after graduation. He is the best advisor I could ever imagine.

My special thanks go to my collaborators and mentors. I am very grateful to Dr. Ari Juels, who has collaborated with me since the start of my dissertation research, for sharing his knowledge and passion with me, inspiring me to pursue creativity, and being unconditionally supportive to my research. I also wish to thank Dr. Fabian Monrose, who offered me extensive advice and valuable feedback in my research and study. I want to thank Dr. Alina Oprea and Dr. Thomas Ristenpart, for their dedication to our collaborative research projects and the enthusiasm they have exemplified to me in pursuing outstanding academic achievements. I will be forever in debt to my mentor at Google Research, Dr. Úlfar Erlingsson, who provided me with different perspectives in research, broadened my horizon and offered me priceless career advice and support. I am also grateful to Dr. Jan Prins and Dr. Bryan Parno for their feedback and valuable suggestions that help improve my dissertation.

At last, I would like to acknowledge the financial support to my research from National Science Foundation (Award Number 0756998, 0910483, and 1330599), from the Science of Security Lablet at North Carolina State University, from RSA Laboratories and VMWare, and also from Google by awarding me the Google Ph.D. fellowship in security.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACPI	Advanced Configuration and Power Interface
APIC	Advanced Programmable Interrupt Controller
ASLR	Address Space Layout Randomization
BTB	Branch Target Buffers
CFG	Control-Flow Graph
CSR	Cross-Site Request
DAC	Discretionary Access Control
DP	Dynamic Programming
EC2	Elastic Compute Cloud
EMA	Exponential Moving Average
GnuPG	Gnu Privacy Guard
HMM	Hidden Markov Model
HPET	High Precision Event Timer
HVM	Hardware-assisted Virtual Machine
IaaS	Infrastructure-as-a-Service
IDS	Intrusion Detection System
IPI	Inter-Processor Interrupt
NFA	Nondeterministic Finite Automaton
OS	Operating System
PaaS	Platform-as-a-Service
PCIDSS	Payment Card Industry Data Security Standard
PIT	Programmable Interval Timer
PLT	Procedure Linkage Table
PMU	Performance Monitoring Unit
PRNG	Pseudorandom Number Generator
RCU	Read-Copy-Update
SaaS	Software-as-a-Service
SLA	Service-Level Agreement

SMP	Symmetric Multi-Processing
SMT	Simultaneous Multi-Threading
SVM	Support Vector Machine
TLB	Translation Lookaside Buffer
TLS	Transport Layer Security
TOFU	Trust on First Use
TSC	Time Stamp Counter
VCPU	Virtual Central Processing Unit
VDI	Virtual Desktop Infrastructure
VM	Virtual Machine

CHAPTER 1: INTRODUCTION

With its massive pooling and multiplexing of computing resources, the cloud offers enterprises the prospect of lower IT costs, lighter administrative burdens, and rapid scaling of resources. Based on the abstraction layer the cloud services export, they are commonly taxonomized as Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS). SaaS presents an application-level interface to the tenants, the customers of the cloud. PaaS offers an application-development environment but abstracts away lower software layers such as the operating systems (OS) and programming language runtime environments. In an IaaS system, computing resources are generally made available to tenants in the form of virtual machine instances. Cloud services can also be categorized as public or private. Public clouds are operated for the benefit of multiple, organizationally distinct tenants—i.e., are multi-tenant environments—and generally available as dynamically provisioned, self-service offerings. Private clouds are operated for the benefit of a single tenant, often within a facility owned and managed by the tenant itself.

Security in a public cloud is a one of the major impediments to enterprise and government adoption of multi-tenant offerings (TrendMicro, 2011; NIST, 2012; Yeboah-Boateng and Essandoh, 2014; Shriver, 2014). By relinquishing control over their IT resources, cloud tenants expose themselves to less understood security models. They have little visibility into the design and implementation of the cloud security mechanisms, leaving them no choice but to rely on the security guarantees (usually not expressed explicitly) provided by the cloud vendors. The effectiveness of traditional security tools such as firewalls and intrusion detection systems (IDSs) diminishes in the cloud due to the obscure security perimeters. Even worse, new threats emerge in cloud computing. Because of the nature of multi-tenancy in public clouds, it has been widely perceived that businesses may find themselves sharing adjacent or overlapping computing resources with partners, suppliers, competitors, or attackers. The potential security threats from a co-located tenant have been projected but largely ignored in

practice because of the blind faith in the cloud providers' ability to enforce strong isolation among different tenants.

This dissertation aims at exploring one of the major design flaws in public clouds—insufficient isolation between cloud tenants as evidenced by the inability to prevent side-channel attacks between them. Side channels exploited in our studies are primarily timing channels in the CPU caches, which allow secret information to be leaked across the boundary between cloud tenants. This insufficient isolation, we hypothesize, exists in both public IaaS and PaaS clouds.

In public IaaS clouds, e.g., Amazon's Elastic Compute Cloud (EC2) and Rackspace, tenants essentially have complete ownership of their VMs but no control over the lower layers of the infrastructure, i.e., hypervisors (a.k.a. virtual machine monitors or VMMs) and the cloud management fabric. The security isolation among VMs run by different tenants on the same physical server is provided by its virtualization technology (e.g., Xen) and its VMM. A VMM is designed to enforce strong logical isolation between the VMs so that one VM cannot access data possessed by another. However, analogous to the threat of cross-process side-channel attacks—e.g., low-privilege processes exfiltrating sensitive information from high-privilege processes via cache-based timing-channel analysis—cross-VM side-channel attacks have been a type of oft-discussed yet never demonstrated threat. The use of cache-based side channels to exfiltrate sensitive information, e.g., cryptographic keys, in virtualized environments, if successful, would significantly undermine the established belief that the security isolation provided by modern virtualization technologies remains adequate under the new threat model of multi-tenant public clouds.

Compared to IaaS clouds, PaaS provides different cloud service abstractions. With a PaaS user interface, the tenants are usually only allowed to upload the source code of their applications and launch the applications through the user control panels, without privileged accesses to the underlying, shared OS resources. As of today, many PaaS providers (e.g., Heroku, AppFog, DotCloud, Openshift) build their business on top of third-party IaaS clouds, and therefore are motivated economically to multiplex each VM to multiple (usually hundreds of) PaaS tenants. The high density of tenancy exacerbates the risks of insufficient tenant isolation in such environments. Existing side-channel attacks mountable by one process on another running in the same OS seem well suited to performing cross-tenant attacks in PaaS deployments. However, to the best of our knowledge, no such attack has been successfully demonstrated in PaaS settings, because the particular types of applications in PaaS,

which involves few cryptographic operations, are less susceptible to the existing side-channel attacks particularly designed to steal cryptographic keys. A novel approach that generalizes the cryptographic side channels to extract other sensitive information from PaaS-based applications and a successful demonstration of such attacks against realistic applications in public PaaS would provide strong evidence of the hypothesized, insufficient isolation, and call into question the dominant practices for isolating tenants in PaaS clouds.

This dissertation includes three major components. First of all, we demonstrate a cache-based cross-VM side-channel attack that extracts a cryptographic private key from another VM co-located on the same physical machine (outlined in Section 1.1 and detailed in Chapter 3). Such attacks evidence the insufficient isolation between IaaS cloud tenants in the face of hardware side channels, and call into question the use of multi-tenant clouds for security critical applications. Second, we propose two defensive techniques to address threats of such side-channel attacks in IaaS clouds. One defense, called HomeAlone, helps cloud tenants who request dedicated cloud servers to remotely verify that the tenant's VMs are physically isolated, even without help from the cloud provider (outlined in Section 1.2 and detailed in Chapter 4). Another defensive tool, named Düppel, allows tenants running on shared hardware to confound cache-based side-channel observations by modifying their VM's own OS kernels (see Section 1.3 and Chapter 5). Finally, we propose a general framework to apply cache-based side-channel attacks to PaaS-based applications, and demonstrate three practical attacks in public PaaS offerings (see Section 1.4 and Chapter 6).

Together, these results establish: *The design of, and common practices in, modern public multi-tenant IaaS and PaaS clouds are flawed in their insufficient isolation of cloud tenants from side-channel attacks by co-located tenants. However, in IaaS clouds, tenants can defend against prominent side-channel attacks by themselves with modified operating system kernels.*

1.1 Cross-VM Cryptographic Side Channels

Modern virtualization technologies such as Xen, KVM, HyperV, and VMWare are rapidly becoming the cornerstone for the security of many computing systems, including but not limited to IaaS public clouds. This reliance stems from their seemingly strong isolation guarantees, meaning their ability to prevent guest virtual machines running on the same system from interfering with each

other's execution or, worse, exfiltrating confidential data across VM boundaries. The assumption of strong isolation underlies the security of public cloud computing systems (Ristenpart et al., 2009; Armbrust et al., 2010) such as Amazon EC2, Microsoft Windows Azure, and Rackspace; military multi-level security environments (Meushaw and Simard, 2000); home user and enterprise desktop security in the face of compromise (England and Manferdelli, 2006); and software-based trusted computing (Garfinkel et al., 2003).

VMMs of modern virtualization systems attempt to realize this assumption by enforcing logical isolation between VMs using traditional access-control mechanisms. But such logical isolation may not be sufficient if attackers can circumvent them via side-channel attacks. Concern regarding the existence of such attacks in the VM setting stems from two facts. First, in non-virtualized, cross-process isolation contexts, researchers have demonstrated a wide variety of side-channel attacks that can extract sensitive data such as cryptographic keys on single-core architectures (Percival, 2005; Osvik et al., 2006; Neve and Seifert, 2007; Aciiçmez and Seifert, 2007; Aciiçmez et al., 2007c,b; Aciiçmez, 2007; Tromer et al., 2010; Gullasch et al., 2011; Aciiçmez et al., 2010; Yarom and Falkner, 2013). The most effective attacks tend to be access-driven attacks (see Section 2.1) that exploit shared micro-architectural components such as caches. Second, Ristenpart et al. (2009) exhibited coarser, cross-VM, access-driven side-channel attacks on modern symmetric multi-processing (SMP, also called multi-core) architectures. But their attack could only provide crude information (such as aggregate cache usage of a guest VM) and, in particular, is insufficient for extracting cryptographic secrets.

Despite the clear potential for attacks, no actual demonstrations of fine-grained cross-VM side-channels attacks have appeared. The oft-discussed challenges (Weiß et al., 2012; Ristenpart et al., 2009) to doing so stem primarily from the facts that VMMs place more layers of isolation between attacker and victim than in cross-process settings, and that modern SMP architectures do not appear to admit fine-grained side-channel attacks (even in non-virtualized settings) because the attacker and victim are often assigned to disparate cores. A lack of demonstrated attack is not a proof of security, and so whether fine-grained cross-VM side-channel attacks are possible has remained an important open question.

In Chapter 3, we will present the development and application of a cross-VM side-channel attack in exactly such an environment. Like many attacks before, ours is an access-driven attack in which

the attacker VM alternates execution with the victim VM and leverages processor caches to observe behavior of the victim. However, we believe many of the techniques we employ to accomplish this effectively and with high fidelity in a virtualized SMP environment are novel. In particular, we provide an account of how to overcome three classes of significant challenges in this environment: (i) inducing regular and frequent attacker-VM execution despite the coarse scheduling quanta used by VMM schedulers; (ii) overcoming sources of noise in the information available via the cache timing channel, both due to hardware features (e.g., CPU power saving) and due to software ones (e.g., VMM execution); and (iii) dealing with core migrations, which give rise to cache "readings" with no information of interest to the attacker (i.e., the victim was migrated to a core not shared by the attacker). Finally, we customize our attack to the task of extracting a private decryption key from the victim and specifically show how to "stitch together" these intermittent, partial observations of the victim VM activity to assemble an entire private key.

As we demonstrate in a lab testbed, our attack establishes a side-channel of sufficient fidelity that an attacker VM can extract a private ElGamal decryption key from a co-resident victim VM running GNU Privacy Guard (`GnuPG`, `http://www.gnupg.org`), a popular software package that implements the OpenPGP e-mail encryption standard (Callas et al., 1998). The underlying vulnerable code actually lies in the most recent version of the `libgcrypt` library, which is used by other applications and deployed widely. Specifically, we show that the attacker VM's monitoring of a victim's repeated exponentiations over the course of a few hours provides it enough information to reconstruct the victim's 457-bit private exponent accompanying a 4096-bit modulus with very high accuracy—so high that the attacker was then left to search fewer than $10,000$ possible exponents to find the right one.

We stress, moreover, that much about our attack generalizes beyond ElGamal decryption (or, more generally, discovering private exponents used in modular exponentiations) in `libgcrypt`. In particular, our techniques for preempting the victim frequently for observation and sidestepping several sources of cache noise are independent of the use to which the side-channel is put. Even those components that we necessarily tune toward ElGamal private-key extraction, and the pipeline of components overall, should provide a roadmap for constructing side-channels for other ends. We thus believe that our work serves as a cautionary note for those who rely on virtualization for guarding

highly sensitive secrets of many types, as well as motivation for the research community to endeavor to improve the isolation properties that modern VMMs provide to a range of applications.

1.2 Co-Residency Detection

Strong security isolation among tenants is a pillar of secure cloud computing. Logical isolation of computing resources can help protect against poorly or inadequately implemented or conceived access-control policies. However, virtual machines that execute on the same physical machine share a range of hardware resources—computing, memory, and so forth. Even when solid logical isolation ensures against abuse of explicit logical channels, shared hardware creates vulnerabilities to side-channel attacks. Due to the projection of such security risks, as evidenced by Ristenpart et al. (2009) and our research (described in Chapter 3), government agencies and enterprises often demand *physical* isolation for their cloud deployments. For example, NASA and Amazon negotiated a cloud service contract for seven months, due to wrangling over NASA's rights to hardware inspection (Stone and Vance, 2010). More recently, cloud providers started to provide isolated VM instances to customers with high security demands (e.g., Amazon introduced a new cloud service with physically isolated, tenant-specific hardware.)

While cloud providers may promise physical isolation, and even commit to it in service-level agreements (SLAs), enforcement by tenants and cloud auditors is a challenge. Cloud systems make heavy use of virtualization to abstract away underlying hardware for simplicity and flexibility. They are architected to be hardware opaque, not hardware transparent, and thus sit at odds with the goal of verifying physical isolation.

In Chapter 4, we will introduce *HomeAlone*, a new tool that allows a tenant or auditor to remotely verify that the tenant's VMs are physically isolated, i.e., that the tenant has exclusive use of a given physical machine. HomeAlone permits such verification with *no hypervisor modification*, and with no explicit action on the part of the cloud provider. The provider need not even be aware that HomeAlone is in operation. The key insight behind HomeAlone is that side channels aren't just vulnerabilities: They can aid *defensive* detection. HomeAlone exploits side channels (via the last-level cache) to detect undesired co-residency. The basic idea in HomeAlone is for the tenant to coordinate its VMs (called *friendly VMs*) so that they silence their activity in a selected cache

region for a period of time. The tenant then measures the cache usage during the resulting quiescent period and checks that there is no unexpected activity. Any such activity suggests the presence of a *foe VM*—our generic term for another tenant's VM—running on the same physical machine.

In practice, HomeAlone requires an approach more complicated than simple silencing of friendly VMs and listening for foe cache activity. Even without friendly VM activity, the last-level cache in a virtualized environment is never entirely quiet, and measurement of its activity (via techniques described in Chapter 2) is error-prone. The timing channel by which HomeAlone measures cache activity is subject to many forms of noise, including scheduling interruptions, coarse timer readings, and core migration in a multi-core environment. Even more challenging is the background noise created by low-level system activity (e.g., that of the hypervisor and `Dom0` in Xen), which HomeAlone needs to distinguish from foe VM activity. Consequently, a major challenge in the design of HomeAlone is the construction of an effective classifier that can distinguish normal cache activity in a friendly environment from the activity introduced by a foe. This classifier in HomeAlone is carefully designed to address complications such as core migration and the impact of friendly-VM and `Dom0` activity on the cache.

Another major technical challenge in HomeAlone is performance overhead: It is desirable in practice that silencing friendly VMs doesn't substantially degrade their performance. HomeAlone thus silences VMs in a *selective* manner. During detection periods, friendly VMs coordinate avoidance of just a small, randomly selected region of the cache, set aside for foe detection. Selective cache avoidance is challenging, and requires kernel modifications in the guest OS of the friendly VMs. By taking advantage of the double indirection layer in memory virtualization, we build an *address remapper* that remaps a set of physical memory pages (corresponding to the cache region avoided by friendly VMs) to a reserved pool of available pages. We show that the impact of selective cache avoidance on the performance of several realistic workloads is modest. For this reason, and because HomeAlone requires no hypervisor modification or cloud-provider support, tenants can use HomeAlone undisruptively and as often as desired to verify isolation policies.

We demonstrate that HomeAlone effectively detects foe VMs whose activities are significantly evidenced in the last-level cache during their execution. We believe that HomeAlone will most commonly detect policy misconfigurations or cost cutting by a service provider that produces

undesired co-residency. We further show, however, that HomeAlone can impose significant obstacles even to hostile foe VMs that attempt to use the last-level cache as an avenue for side-channel attacks.

1.3 OS-Level Side-Channel Mitigation

In side-channel attacks in multi-tenant public clouds as demonstrated to date, shared CPU caches enable virtual machines administered by competing organizations to exfiltrate sensitive information from each other, which has been shown possible despite considerable interference and background "noise" from the hypervisor and other activities on the machine (see Chapter 3). A potential but very *expensive* solution, as discussed in Section 1.2, is to physically isolate mutually untrusted cloud tenants with high security demands, which may not be a viable solution for regular public cloud users who will continue the use of multi-tenant machines in public IaaS clouds.

Most approaches to address side-channel attacks in multi-tenant clouds have focused on altering the cloud platform in some way (see Section 2.3). However, to our knowledge, these defenses have not gained traction in existing public clouds. Rather, a typical tenant of a public cloud concerned about these attacks is left with little choice but to try to *defend itself*. One approach is to construct its software to resist side channels, e.g., (Molnar et al., 2006; Könighofer, 2008; Coppens et al., 2009), but these techniques can result in significant slowdown.

In Chapter 5 we will explore another possibility, namely a method by which a tenant can construct its VMs to automatically inject additional noise into the timings that an attacker might observe from caches. Since these timings are the most common side channels by which an attacker infers sensitive information about a victim VM, injecting noise into them will generally make the attacker's job more difficult. Our implementation of this idea, called Düppel[1], modifies only the guest OS kernel and is general enough to protect arbitrary types of user-space applications. Düppel can be configured to protect the user-space application, any dynamically linked libraries it uses, or both. Düppel does not need support from hypervisors or cloud providers. To our knowledge, our scheme is the first work that provides tenant VM OS-layer mitigation of cross-VM side channels.

Unlike the noise overcome by previous attacks, the noise injected by Düppel is designed specifically to confound attacks mounted via timing the per-core L1 cache (or per-core L2 cache,

[1]Düppel takes its name from a radar countermeasure developed by the German Luftwaffe during World War II, in which aircraft disperse clouds of tiny pieces of material to interfere with radar.

if present) on the platform. Düppel does so by repeatedly cleaning the L1 cache alongside the execution of its tenant workload, at a pace that it adjusts based upon the possibility with which timings reflecting the workload execution could actually have been observed from another VM. We also discuss extensions of Düppel to defeat timing attacks via other time-shared resources such as the branch prediction cache, should attacks via this cache (Aciiçmez et al., 2007c,b; Hund et al., 2013) someday be adapted to a virtualized setting. We emphasize, however, that even with just addressing the L1 cache, Düppel already interferes with all known cryptographic side-channel attacks that have been demonstrated in a virtualized SMP environment. (See Section 5.5 for further discussion on this point.)

Overhead of Düppel is modest. In tests on Amazon EC2, we show that file download latencies and server throughput over a TLS-protected connection from an Apache web server suffer by at most 4% when Düppel is configured to protect the OpenSSL library. We also demonstrate using the PARSEC benchmarks and other programs that computational workloads suffer by up to 7% when the applications are protected by Düppel. We believe that these overheads are acceptable given the substantial challenge involved in defending against cache-based side-channels with no help from the underlying hardware or hypervisor.

1.4 Cross-Tenant Side Channels in PaaS

Public PaaS clouds are an important segment of the cloud market, being projected for compound annual growth of almost 30% through 2017 (Mahowald et al., 2013) and "on track to emerge as the key enabling technology for innovation inside and outside enterprise IT" (Natis, 2014). For our purposes here, a PaaS cloud permits tenants to deploy tasks in the form of interpreted source (e.g., PHP, Ruby, Node.js, Java) or application executables that are then executed in a provider-managed host OS shared with other customers' applications. As such, a PaaS cloud often leverages OS-based techniques such as Linux containers to isolate tenants, in contrast to hypervisor-based techniques common in IaaS clouds.

A continuing threat to cloud tenant security is failures of isolation due to side-channel information leakage. A small but growing handful of works have explored side channels in settings characteristic of IaaS clouds, to which tenants deploy tasks in the form of virtual machines. Demonstrated attacks

include side channels by which an attacker VM can extract coarse load measurements of a victim VM with which it is co-located (Ristenpart et al., 2009); identify pages it shares with a co-located victim VM, allowing it to detect victim VM applications, downloaded files (Suzaki et al., 2011) and its operating system (Owens and Wang, 2011); and even exfiltrate a victim VM's private decryption key (Chapter 3). However, only the first of these attacks was demonstrated on a public cloud, with the others being demonstrated only in lab settings. To the best of our knowledge, no side-channel attack capable of extracting granular information from a victim has been demonstrated in the wild.

In Chapter 6, we will initiate the study of cross-tenant side-channel attacks specifically in PaaS clouds and, in doing so, provide the first demonstration of granular, cross-tenant side channels in commercial clouds of any sort. Existing side-channel attacks mountable by one process on another running on the same OS, particularly those that leverage processor caches, e.g., (Percival, 2005; Neve and Seifert, 2007; Tromer et al., 2010; Gullasch et al., 2011; Yarom and Falkner, 2013), seem well suited to performing cross-instance[2] attacks in PaaS deployments. This is largely true in our experience, though directly leveraging these attacks in PaaS settings is not as straightforward as one might think. One reason is that even identifying suitable targets to attack in a PaaS deployment requires some thought. After all, cryptographic keys that commonly form their most natural targets are largely absent in typical PaaS environments where cryptographic protections (e.g., storage encryption, or application of TLS encryption to network traffic) are commonly provided as a service by the cloud operator, often on a different computer than those used to host tenant instances.

In Chapter 6, we will report on our investigation of cache-based side channels in PaaS clouds that, among other things, therefore identifies several novel targets (in the context of cross-tenant side-channel attacks) for PaaS environments: (i) We show how an attacker instance can infer aspects of a victim web application's responses to clients' service requests. In particular, we show that an attacker can reliably determine the number of distinct items in an authenticated user's shopping cart on an e-commerce site (the victim instance) running the popular Magento e-commerce application. (ii) We show how an attacker instance can hijack a user account on a web site (the victim instance) by predicting the pseudorandom number it embeds in a password reset link. We specifically demonstrate

[2]While "instance" typically refers to an instantiated VM in an IaaS setting, here we borrow the term for the PaaS setting, to refer more generically to a collection of running computations on one physical machine that are associated with the same tenant and should be isolated from other tenants.

this attack against the PHP pseudorandom number generator that the site uses. (iii) We show how an attacker instance can monitor the victim so as to obtain a padding oracle to break XML encryption schemes. In particular, we demonstrate a Bleichenbacher attack (Bleichenbacher, 1998) against SimpleSAMLphp, an open-source SAML-based authentication application that implements PKCS#1 v1.5 RSA encryption in a manner resistant to these attacks via other vectors (but not via our side-channel attacks).

We stress, moreover, that we have successfully mounted each of these attacks in commercial PaaS clouds (though obviously against victims that we deployed ourselves). Moreover, as a side effect of doing so, we have also addressed how to achieve co-location of an attacker instance with a victim instance in these PaaS clouds. To our knowledge, our attacks are thus the first granular, cross-tenant attacks demonstrated on commercial clouds, PaaS or otherwise.

A key ingredient in our attacks is a framework we develop through which the attacker instance can trace a victim's execution paths inside shared executables. Starting with the control-flow graph (CFG) of a executable shared with the victim, our framework consists of building an *attack nondeterministic finite automaton* (*attack NFA*) that prescribes the memory chunks that the attack instance should monitor over time, using a known cache-based side channel (Gullasch et al., 2011; Yarom and Falkner, 2013), in order to trace the victim's execution path in the CFG. This general framework can then be used to characterize the victim's execution for specific attacks, such as the exact number of times a certain execution path segment was traversed in a short interval (in the first attack above); the precise time at which certain path segments were traversed by the victim (as in the second attack); or the direction taken in a specific branch of interest (in the third attack). We believe the attack NFA framework that we introduce here will be similarly useful in subsequent work on both evaluating and defending against cross-instance side-channel attacks. The contributions of the work described in Chapter 6 will be threefold: (i) a general framework for expressing and guiding cross-instance side-channel attacks leveraging shared executables; (ii) identification of novel and important targets for side-channel attacks in PaaS environments; and (iii) demonstration of attacks against these targets in commercial PaaS clouds.

CHAPTER 2: BACKGROUND AND PRIOR WORK

This dissertation revolves around side channels in modern CPU caches and their applications in the context of cloud computing. In this chapter, we will first introduce the background knowledge of CPU cache architectures and cryptographic side channels in CPU caches. Then we will briefly introduce the concept of cloud computing and side-channel threats in the cloud context. We will also summarize the related work on side channels along with the background introduction.

2.1 Cryptographic Side Channels in CPU Caches

2.1.1 Caches in Modern Architectures

Modern CPU micro-architectures extensively make use of hardware caches to speed up expensive operations. The hardware caches on an x86 platform may include data caches, instruction caches, translation lookaside buffers, trace caches, branch target buffers (BTB) in the branch prediction unit, etc. The most commonly known caches are data and instruction caches sitting between the processor cores and the main memory, establishing a storage hierarchy in which each level stores the interim data for the next level storage system for quick reference. While current main memory latencies are on the order of several hundred nanoseconds, the fastest L1 cache has latency as low as several nanoseconds, resulting in a difference of two to three orders of magnitude. To reduce the cost of L1 cache misses, current processors include larger L2 and sometimes even L3 caches with slightly higher access latencies.

Cache sizes range from several KB to several MB. They are organized as a sequence of blocks called *cache lines*, with fixed size typically between 8 and 512 bytes. We define a *chunk* as a cacheline-sized, aligned region in the memory. For example, if a cache-line is of size 64B, then each address that is a multiple of 64 defines the chunk starting at that address. Typical caches are *set-associative*. Each chunk in the main memory can be placed into only one cache set to which it maps, but can be placed in any of the w lines in that set. The spectrum of set-associative caches

includes two extremes: *direct-mapped* caches in which each set contains only one cache line and thus a chunk has a unique location in the cache, and *fully associative* caches in which there is only one cache set containing all cache lines—a chunk can be mapped to any location in the cache. Increasing the degree of associativity usually decreases the cache miss rate, but it increases the cost of searching a chunk in the cache.

A w-way set-associative cache is partitioned into m sets, each with w lines of size b. So, if the size of the cache is denoted by c, we have $m = c/(w \times b)$. For example, in the L1 instruction cache of an Intel Yorkfield processor as used in our lab testbed, $c = 32\text{KB}$, $w = 8$, and $b = 64\text{B}$. Hence, $m = c/(w \times b) = 64$. Moreover, the number of chunks in a 4KB memory page is $4\text{KB}/b = 64$. In such cases, memory chunks with the same offset in a memory page will be mapped to the same cache set in such instruction caches.

Caches in SMP and SMT architectures. On multi-core (a.k.a., SMP) architectures, different cores may or may not share a cache. For example, in the four-core Intel Extreme processor, each core has its own L1 cache and each of the two L2 caches is shared by two cores. In more modern Intel CPUs, a unified L3 cache is shared by all cores, while each core has its L1 data/instruction caches and a unified L2 cache. CPUs supporting simultaneous multi-threading (SMT)[1] allow multiple threads to execute simultaneously on the same CPU core and share the same cache hierarchy. It has been shown in previous research that SMT facilitates CPU-based side channel attacks because two threads can simultaneously use CPU resources (Percival, 2005; Tromer et al., 2010; Aciiçmez, 2007; Aciiçmez et al., 2010, 2007c).We target systems with multi-core configurations without SMT due to their dominance in today's CPU market. A CPU cache in this case is either time-shared by the attacker and victim, or simultaneously shared by the two entities. An example of a time-shared cache is the L1 cache, which can only be shared by two threads running on the same core in turns.

2.1.2 Taxonomy of Cache Side Channels

Side-channel attacks on computer systems and their use to extract cryptographic keys from a victim entity have been studied in a variety of settings. According to the vantage points of the

[1]SMT is called hyperthreading in Intel's terminology.

attackers, existing cryptographic side-channel attacks are generally categorized into one of the three classes: time-driven attacks, trace-driven attacks and access-driven attacks.

2.1.2.1 Time-Driven Attacks

A time-driven cryptographic side-channel attack is possible when the *total* execution times of cryptographic operations with a fixed key are influenced by the value of the key, e.g., due to the structure of the cryptographic implementation or due to system-level effects such as cache evictions. This influence can be exploited by a *remote* attacker who can measure many such timings to statistically infer information about the key, e.g., (Kocher, 1996; Brumley and Boneh, 2003; Aciiçmez et al., 2005; Bernstein, 2005; Bonneau and Mironov, 2006; Aciiçmez et al., 2006). More recently, Weiß et al. (2012) mounted time-driven attacks against an embedded uniprocessor device virtualized by the L4 microkernel, but the differences between virtualized and non-virtualized targets in time-driven attacks are not as significant as those in other attack types. Compared to the cross-VM attacks proposed in Chapter 3, their techniques do not translate to the style of attack we pursue or the virtualized SMP environment in which we attempt it.

2.1.2.2 Trace-Driven Attacks

A second class of cryptographic side-channel attacks is called trace-driven. These attacks require the attacker to have physical access to the cryptographic system and be able to continuously monitor the execution trace throughout a cryptographic operation. Such execution traces can be generated via power consumption analysis (Kocher et al., 1999; Akkar et al., 2000; Bertoni et al., 2005; Lauradoux, 2005; Aciiçmez and Koç, 2006), fault analysis (Blömer and Seifert, 2003) or electromagnetic analysis (Gandolfi et al., 2001). The ability to continuously monitor the device makes these attacks quite powerful but typically requires physical proximity to the device. In the cloud context, physical access to the target system is unlikely in most attack scenarios.

2.1.2.3 Access-Driven Attacks

The third class of side-channel attacks is a access-driven attack, in which the attacker runs a program on the same system that is performing the cryptographic operation of interest. It assumes the attacker has logical access to the system (i.e., the ability to run his own program) and be able to

monitor usage (e.g., via timing) of a shared architectural component, e.g., the data cache (Percival, 2005; Osvik et al., 2006; Neve and Seifert, 2007; Tromer et al., 2010; Gullasch et al., 2011), instruction cache (Aciiçmez, 2007; Aciiçmez et al., 2010), floating-point multiplier (Aciiçmez and Seifert, 2007), or branch-prediction cache (Aciiçmez et al., 2007c,b). The strongest attacks in this class, first demonstrated only recently (Aciiçmez et al., 2010; Gullasch et al., 2011), are referred to as *asynchronous*, meaning that they don't require the attacker to achieve precisely timed observations of the victim by actively triggering victim operations. These attacks leverage CPUs with SMT or the ability to game operating system process schedulers. Our cross-VM side-channel attack described in Chapter 3 is an asynchronous access-driven side-channel attack through CPU caches in virtualized environments that games the hypervisor scheduler from the guest OS.

2.1.3 Prime-Probe Protocols

One particular type of access-driven cache side-channel attack is based on PRIME-PROBE protocols. According to the type of caches the attacks are applied to, techniques used to establish a PRIME-PROBE protocol may slightly differ. Two types of cache PRIME-PROBEs will be described: data cache probing and instruction cache probing. We will defer the introduction to another type of access-driven side channels—FLUSH-RELOAD protocols—to Chapter 6.

2.1.3.1 Data Cache Probing

Cache-based timing channels have been widely studied in various contexts. In spite of different methodologies employed in constructing these channels, they all exploit the timing difference in access latencies between the cache and main memory. We first consider the cache-based timing channel constructed by measuring the cache load of a monitored entity V that shares a common data cache with the monitoring entity U. A common method for conducting an access-driven cache attack is to PRIME and later PROBE the cache, a so-called PRIME-PROBE protocol, as introduced by Tromer et al. (2010).

PRIME: Entity U fills an entire cache set S by reading memory region M from its own memory space.

IDLE: Entity U waits for a prespecified PRIME-PROBE *interval* while the cache is utilized by monitored entity V.

PROBE: Entity U times the reading of the same memory region M to learn V's cache activity on cache set S.

If there is much cache activity from V during U's PRIME-PROBE interval, then U's data is likely to be evicted from the cache set and replaced with data accessed by V. This will result in a noticeably higher timing measurement in U's PROBE phase than if there had been little activity from V.

Cache-based side channels are typically dependent on the processor architecture and the cache level utilized. In a typical x86 platform, the L1 data caches and the unified L2 or L3 caches can serve as the data cache in the PRIME-PROBE protocol, as they can be loaded via memory reads.

2.1.3.2 Instruction Cache Probing

The basic technique for instruction cache probing is similar to that used in data cache probing, except that it times how long it takes to execute instructions from memory associated with individual cache sets, as described previously by Aciiçmez (2007). To do so, we first allocate sufficiently many contiguous memory pages so that their combined size is equal to the size of the I-cache. We then divide each memory page into 64 chunks. The i^{th} chunk in each page will map to the same cache set. To fill the cache set associated to offset i, then, it suffices to execute an instruction within the i^{th} chunk of each of the allocated pages. Filling a cache set is called PRIME-ing. We will also want to measure the time it takes to fill a cache set, which is called PROBE-ing. To PROBE the cache set associated with offset i, we execute the `rdtsc` instruction, then jump to the first page's i^{th} chunk, which has an instruction to jump to the i^{th} chunk of the next page, and so on. The final page jumps back to code that again executes `rdtsc` and calculates the elapsed time. This is repeated for each of the m cache sets to produce a vector of cache set timings. More specifically, a VCPU U of the attacker's VM spies on a victim's VCPU V by measuring the cache load in the L1 instruction cache in the following manner:

PRIME: U fills one or more cache sets by the method described above.

IDLE: U waits for a prespecified PRIME-PROBE *interval* while the cache is utilized by V.

PROBE: U times the duration to refill the same cache sets to learn V's cache activity on those sets.

Cache activity induced by V during U's PRIME-PROBE interval will evict U's instructions from the cache sets and replace them with V's. This will result in a noticeably higher timing measurement in U's PROBE phase than if there had been little activity from V. Of course, PROBE-ing also accomplishes PRIME-ing the cache sets (i.e., evicting all instructions other than U's), and so repeatedly PROBE-ing, with one PRIME-PROBE interval between each PROBE, eliminates the need to separately conduct a PRIME step.

2.1.4 Sources of Noise

The PRIME-PROBE protocols described in previous sections may be subject to many sources of noise, especially in the virtualized environment. We categorize them as hardware noise and software noise.

2.1.4.1 Hardware Noise

TLB misses. Most CPUs implement virtual memory as a method of providing a contiguous address space to processes. To speed up address translation, translation lookaside buffers (TLBs) cache recently used page table entries containing virtual-to-physical memory mappings. In x86 processors, hardware TLBs are usually small set-associative caches that cache the translation from virtual addresses to physical addresses. Upon a TLB miss, the CPU itself walks the page tables to look for a mapping of the virtual address not found in the TLB, which can be expensive (as high as 100 cycles). When PROBEing per-core caches that involve context switches, such as L1 data caches or L1 instruction caches, because the TLB is flushed at each context switch, the PROBE of the first cache set will always involve TLB misses and so will be abnormally high; as such, the PROBE results for the first cache set will be discarded. When PROBEing large caches, such as last-level caches, the number of memory pages used for the PRIME-PROBE protocol is usually large which may invoke TLB evictions during the PRIME-PROBE protocols, adding noise in the cache timing measurements.

Speculative execution and hardware prefetching. Modern superscalar processors usually fetch instructions in batches and execute them out-of-order. This is an issue for accurate time measurement

as well as for PROBEing the instruction caches. In order to force the in-order execution of our PROBE code for accurate measurement, the instructions need to be serialized using instructions like `cpuid` and `mfence`. Data cache misses usually involve hardware prefetching activities. CPUs will prefetch several cache lines from a memory page that incurs several cache misses. To obtain accurate timing measurements in the PROBE phase, one technique is to access the buffer in pseudo-random order in the PROBE phase (Percival, 2005).

Power saving. The speed of a PROBE may be subject to change due to CPU power saving modes. If the attacker VM is solely occupying a CPU core, when it finishes its PROBE and relinquishes CPU resources, the core may be slowed to save power. During the PRIME-PROBE protocols, it appears to take longer for the CPU to recover from the power saving mode and, in our experience, yields a much longer effective PRIME-PROBE interval. Thus, longer-then-expected PRIME-PROBE intervals may indicate there was no victim on the same core and so their results are discarded.

2.1.4.2 Software Noise

Emulation of RDTSC instructions. Not all processors maintain a constant-rate timestamp counter (TSC), which means the value of the TSC (from which `rdtsc` calls are answered) may exhibit nonlinearity under certain conditions, e.g., when enabling Intel SpeedStep technology. Newer generations of Intel processors are equipped with a constant-rate TSC, which maintains a constant rate except when the core is put to sleep. The latest generation of processors come with a so-called invariant TSC, the frequency of which never changes regardless of the state of the processor. In the absence of an invariant TSC, Xen 4.0 or later emulates the `rdtsc` call to prevent time from going backwards. In this case, the `rdtsc` call is about 15 or 20 times slower than the native call, which diminishes the attacker VM's ability to measure the duration to PROBE a cache set. In the cross-VM attack described in Section 3, the `rdtsc` instructions are not emulated, in accordance with our findings in the public clouds, therefore rendering more accurate time measurements.

Noise due to other domains. Xen offers a paravirtualized virtual machine abstraction that requires some changes to the guest operating systems running in each VM. Xen implements a thin hypervisor that controls only basic operations, and a control management virtual machine, dubbed `Dom0`. To perform privileged operations, guest VMs can issue software traps into the hypervisor, called

hypercalls. Dom0 is responsible for creating and terminating other VMs, configuring some of their parameters, and handling virtual network interfaces and block devices. Both the hypervisor and Dom0 produce cache activity that introduces noise when measuring cache load. For example, to ensure secure partitioning of VMs, Xen validates modifications to guest page tables. Updates to page tables trigger hypercalls into Xen, and thus they induce hypervisor activity that leaves a pattern in the cache. In the PRIME-PROBE protocol, noise from the hypervisor might evict cache lines primed by the monitoring VM and so increase the timings observed in the PROBE phase. Dom0 is responsible for multiplexing I/O devices across different virtual machines. Dom0 implements all device drivers and has access to the network and physical hard drives. All other VMs transfer data through Dom0 using an asynchronous buffering mechanism. Thus, an I/O intensive application triggers significant activity in Dom0, resulting noise in the cache timing measurements.

Address space layout randomization. Address space layout randomization (ASLR) changes the layout of the virtual address space of a program in each execution. Such randomization, in principle, will alter a program's cache patterns from the point of view of side-channel observations. However, it does not interfere with the attack we describe in Section 3 because the L1 cache set to which memory is retrieved is determined purely by its offset in its memory page (see Section 2.1.1), and because ASLR in a Linux implementation aligns libraries to page boundaries. It doesn't affect our side-channel measurements in Section 4 as well, as HomeAlone does not rely on per-cache-set PROBE timing values for co-residency detection.

2.2 Cloud Computing and Side-Channel Threats

2.2.1 Infrastructure-as-a-Service Clouds

In an IaaS system, computing resources are generally made available to tenants in the form of VM instances. Tenants essentially have complete control of these VMs but no visibility into the lower layers of the infrastructure, e.g., hypervisors (virtual machine monitors) and data-center management consoles. The tenant VM instances may be configured with operating systems from a catalog but are also typically custom-configurable. (Supporting network and storage are often bundled with computing instances but can also be purchased separately.) Amazon's Elastic Compute Cloud (EC2), IBM Computing on Demand, and Rackspace Cloud are well-known examples of IaaS offerings.

Side-channel attacks in the context of infrastructure clouds, or cross-VM settings in general, has been studied in prior work. Demonstrated attacks include side channels by which an attacker VM can extract coarse load measurements of a victim VM with which it is co-located (Ristenpart et al., 2009), and identify pages it shares with a co-located victim VM, allowing it to detect victim VM applications, downloaded files (Suzaki et al., 2011) and its operating system (Owens and Wang, 2011). To the best of our knowledge, the work presented in Chapter 3 is the first demonstrated cryptographic side-channel attack in the cross-VM context.

2.2.2 Platform-as-a-Service Clouds

A canonical public PaaS cloud allows customers to upload interpreted source code (e.g., PHP, Ruby, Node.js, Java) or even application executables, that are then run in a provider-managed host operating system. This OS may itself be running within a guest VM on a public IaaS platform such as Amazon EC2. The host OS facilitates data storage, monitoring and logging, and other value-adds that enable customers to quickly provision applications. A canonical use case is for dynamic web hosting, where the customer provides scripts or applications defining the webpage (i.e., PHP scripts or similar) and a MySQL schema. The convenience and flexibility that PaaS provides to customers, together with the fact that mature IaaS clouds enable quick time-to-market for a new PaaS system, has lead to an explosion in the number of offerings.

2.2.2.1 PaaS Isolation Techniques

PaaS systems are usually multi-tenant, meaning they run multiple customers' instances on the same operating system. As such, isolation between tenants is essential for the security of PaaS clouds. In Table 2.1 we summarize the isolation mechanisms used in a variety of PaaS systems, and describe these models in more detail below.

Runtime-based isolation. Some PaaS clouds host applications owned by multiple tenants in the same process and isolate them with application runtimes. Multiple tenants therefore may share, e.g., the same JVM environment and be isolated only by JVM runtime security mechanisms.

User-based isolation. A more widely used isolation technique is traditional user-based isolation within the host OS. Each hosted application runs as a non-privileged user on the OS, and the instance

PaaS cloud	URL (`http://...`)	Isolation
AppFog	`www.appfog.com`	User
Azure	`azure.microsoft.com`	VM
Baidu App Engine	`developer.baidu.com/en`	Container
Cloud Foundry	`cloudfoundry.org`	User
Elastic Beanstalk	`aws.amazon.com/elasticbeanstalk/`	VM
Engine Yard	`www.engineyard.com`	VM
Heroku	`www.heroku.com`	Container
HP Cloud Application PaaS	`www.hpcloud.com/products-services/application-paas`	Container
Joyent SmartOS	`www.joyent.com`	VM
WSO2	`wso2.com/cloud`	Runtime

Table 2.1: Example PaaS isolation techniques

is a set of processes run by that user. Basic OS-facilitated memory protection prevents illegal memory accesses across instance boundaries, and correctly configured discretionary access control (DAC) in Unix-like systems prevents cross-instance file accesses.

Container-based isolation. The main limitation of user-based isolation is the unrestricted use of computer resources by individual instances. This has been relatively recently addressed with the advent of Linux containers, as implemented by Linux-VServer (`linux-vserver.org`), OpenVZ (`openvz.org`), and LXC (`linuxcontainers.org`). The last has been merged into mainstream Linux kernels. A container is a group of processes that are isolated from other groups via distinct kernel namespaces and CPU scheduling quotas (so-called CPU groups or cgroups).

VM-based isolation. Some PaaS clouds give each customer instance a separate IaaS VM instance, thereby leveraging the isolation offered by modern virtualization.

2.2.2.2 Side Channels in PaaS

Side-channel attacks in platform clouds are similar in concept to traditional cross-process side-channel attacks. Most access-driven attacks, e.g., (Percival, 2005; Osvik et al., 2006; Neve and Seifert, 2007; Aciiçmez, 2007; Aciiçmez and Seifert, 2007; Aciiçmez et al., 2007c,b; Tromer et al., 2010; Gullasch et al., 2011; Aciiçmez et al., 2010; Yarom and Falkner, 2013), would, in principle, be applicable in such context. However, for reasons explained in Section 1.4, our proposed attacks in Chapter 6 are so far the only ones demonstrated in public PaaS clouds. Software-based side-channel information leaks through the `procfs` file system within an OS have been explored by

several researchers (Zhang and Wang, 2009; Jana and Shmatikov, 2012; Qian et al., 2012). In particular, Zhang and Wang (2009) exploited `procfs` to extract the value of `ESP` register of another process to perform the inter-keystroke timing analysis. Jana and Shmatikov (2012) used machine learning to infer the web sites being rendered from the number of memory pages owned by the browser process, which is learned by reading `/proc/pid/statm`. Similarly, Qian et al. (2012) made use of `procfs` to guess TCP sequence numbers and performed off-path hijacking attacks using them. These attacks were targeting at client-side applications, e.g., guessing user passwords or web contents rendered by a browser, and require sharing of namespaces in Linux kernels, which is prohibited in container-based isolations. Therefore we do not believe this type of software-based side channels can be applied to PaaS clouds in general.

2.3 Countermeasures to Side-Channel Attacks

Countermeasures to cache side-channel attacks can be applied to various layers in a computer system. In the context of mitigating cross-VM side channels, prior works can roughly be classified into one of the following categories, based on the layer in which the countermeasure is implemented.

2.3.1 Hardware-Layer Approaches

The first category includes proposals of new cache designs by applying the idea of resource partitioning, e.g., (Page, 2005; Wang and Lee, 2006, 2007; Domnitser et al., 2012), or access randomization, e.g., (Wang and Lee, 2007, 2008; Keramidas et al., 2008), to mitigate timing channels in CPU caches on the hardware layer. Other works have proposed approaches to eliminate fine-grained timing sources in hardware designs (Martin et al., 2012). Compared to implementations in other layers, hardware methods usually have lower performance overhead. However, adoption of new hardware techniques is a complex process, which may involve considerations of side effects (e.g., power consumption) and economic feasibility. Therefore, it might take years before these techniques are merged into production and finally used in clouds.

2.3.2 Hypervisor-Layer Approaches

Countermeasures in the second category intend to mitigate cache timing side channels by adapting the hypervisor. One direction along this line is to hide nuances in the program execution time, either by providing a fuzzy timer (Vattikonda et al., 2011) or by forcing all executions to be deterministic (Aviram et al., 2010). However these approaches will exclude many applications that rely on a fine-grained timer from running in the cloud. A conceptually similar but more comprehensive solution is provided by Li et al. (2013), in which all sources of timing channels a VM can observe are identified and categorized; they are mitigated either by aggregating timing events among multiple VM replicas, or by making them deterministic functions of other timing sources. Another direction is to partition the shared resources in the hypervisor (Raj et al., 2009; Shi et al., 2011; Kim et al., 2012). In particular, Raj et al. (2009) statically partition the last level cache (LLC) into several regions and allow VMs to make use of different regions by partitioning physical memory pages accordingly. Kim et al. (2012) improved the performance of this approach by dynamically partitioning the LLCs and extended the protection to L1 caches as well.

2.3.3 OS-Layer Approaches

Previous OS-layer approaches were mostly proposed to defend against cross-process side-channel attacks, e.g., (Percival, 2005; Tromer et al., 2010; Gullasch et al., 2011). To our knowledge, the approach we propose in Chapter 5 is the first to modify the OS layer to mitigate cross-VM side channels.

2.3.4 Application-Level Approaches

The last approach is to construct side-channel resistant software implementations. For example, the program counter security model (Molnar et al., 2006; Coppens et al., 2009) eliminates key-dependent control flows by transforming the software source code. Other efforts focus on side-channel free cryptographic implementations e.g., (Könighofer, 2008). These approaches incur significant overheads in some cases (e.g., Coppens et al. (2009) indicate up to 24×).

CHAPTER 3: CROSS-VM CRYPTOGRAPHIC SIDE CHANNELS[1]

In this chapter, we detail our study in cross-VM cryptographic side channels outlined in Section 1.1. We will first give an overview of the proposed attacks and the challenges required to overcome in Section 3.1. From there, a full attack pipeline will be explained in Sections 3.2–3.5. The proposed attacks are then evaluated in a lab environment, which is documented in Section 3.6.

3.1 Overview and Challenges

Attack setting. The setting under consideration is the use of confidential data, such as cryptographic keys, in a VM. Our investigations presume an attacker that has in some manner achieved control of a VM co-resident on the same physical computer as the victim VM, such as by compromising an existing VM that is co-resident with the victim.

We focus on the Xen virtualization platform (Barham et al., 2003) running on contemporary hardware architectures. Our attack setting is inspired not only by public clouds such as Amazon EC2 and Rackspace, but also by other Xen use cases. For example, many virtual desktop infrastructure (VDI) solutions (e.g., Citrix XenDesktop) are configured similarly, where virtual desktops and applications are hosted in centralized datacenters on top of a XenServer hypervisor and delivered remotely to end user devices via network connections. Another representative use case separates operating systems into several components with different privilege levels and that are isolated by virtualization (England and Manferdelli, 2006; Piotrowski and Joseph, 2010). An example of such systems is Qubes (Rutkowska and Wojtczuk, 2012), which is an open source operating system run as multiple virtual machines on a Xen hypervisor.

In terms of computer architecture, we target modern multi-core processors without SMT capabilities or with SMT disabled. This choice is primarily motivated by contemporary processors used in public clouds such as Amazon AWS and Microsoft Azure, whose SMT features are intentionally

[1]This chapter is excerpted from Zhang et al. (2012).

disabled, if equipped, since SMT can facilitate cache-based side channel attacks (Marshall et al., 2010).

We assume the attacker and victim are separate Xen DomU guest VMs, each assigned some number of disjoint virtual CPUs (VCPUs). A distinguished guest VM, Dom0, handles administrative tasks and some privileged device drivers and is also assigned some number of VCPUs. The Xen credit scheduler (Chisnall, 2007) assigns VCPUs to the physical cores (termed PCPUs in Xen's context), with periodic migrations of VCPUs amongst the cores. Our threat model assumes that Xen maintains logical isolation between mutually untrusting co-resident VMs, and that the attacker is unable to exploit software vulnerabilities that allow it to take control of the entire physical node. We assume the attacker knows the software running on the victim VM and has access to a copy of it. The attack we consider will therefore uses cross-VM side-channels to reveal a code path taken by the victim application. We will use as a running example—and practically relevant target—a cryptographic algorithm whose code-path is secret-key dependent (look ahead to Figure 3.2). However, most steps of our side-channel attack are agnostic to the purpose for which the side-channel will be used.

Constructing such a side channel encounters significant challenges in this cross-VM SMP setting. We here discuss three key challenge areas and overview the techniques we develop to overcome them.

Challenge 1: Observation granularity. The way Xen scheduling works in our SMP setting makes spying on a victim VM challenging, particularly when one wants to use per-core micro-architectural features as a side channel. For example, the L1 caches contain the most potential for damaging side-channels (Percival, 2005), but these are not shared across different cores. An attacker must therefore try to arrange to frequently alternate execution on the same core with the victim so that it can measure side-effects of the victim's execution. This strategy has been shown to be successful in single-core, non-virtualized settings (Tromer et al., 2010; Neve and Seifert, 2007; Gullasch et al., 2011) by attackers that game OS process scheduling. But no gaming of VMM scheduling suitable for fine-grained side-channels has been reported, and the default scheduling regime in Xen would seem to bar frequent observations: the credit scheduler normally reschedules VMs every $30\mathrm{ms}$, while even a full 4096-bit modular exponentiation completes in about $200\mathrm{ms}$ on a modern CPU core

(on our local testbed, see Section 3.6). This leaves an attacker with the possibility of less than 10 side-channel observations of it.

In Section 3.2, we overcome this challenge to use the L1 instruction cache as a vector for side-channels. We demonstrate how to use inter-processor interrupts (IPIs) to abuse the Xen credit scheduler in order to arrange for frequent interruptions of the victim's execution by a spy process running from within the attacker's VM. This takes advantage of an attacker having access to multiple VCPUs and allows the spy to time individual L1 cache sets. The scheduling nuances abused are a vulnerability in their own right, enabling degradation-of-service attacks and possibly cycle-stealing attacks (Zhou et al., 2011; Tsafrir et al., 2007).

Challenge 2: Observation noise. Even with our IPI-based spying mechanism, there exists significant noise in the measured timings of the L1 instruction cache. Beyond the noise involved in any cache-based measurements, the VMM exacerbates noise since it also uses the L1 cache when performing scheduling. Manual analysis failed to provide simple threshold-based rules to classify cache timings as being indicative of particular victim operations.

In Section 3.3, we use a support vector machine (SVM) to relate L1 cache observations to particular operations of the victim. A critical challenge here is gathering accurate training data, which we accomplish via careful hand instrumentation of the target victim executable. Still, the SVM is error-prone, in the sense that it classifies a small fraction of code paths incorrectly. Fortunately, for many types of victim operations, the fine granularity achieved by the attack VM's IPI-based spying can yield multiple observations per individual operation. We use the redundancy in these observations together with knowledge of the set of possible victim code paths to correct SVM output errors by means of a hidden Markov model (HMM). This is detailed in Section 3.4. The SVM plus HMM combination, when correctly trained, can translate a sequence of observations into a sequence of inferred operations with few errors.

Challenge 3: Core migration. Our SMP setting has attacker and victim VCPUs float amongst the various PCPUs. The administrative Dom0 VM and any other VMs may also float amongst them. This gives rise to two hurdles. First, we must determine whether an observation is associated with the victim or some other, unrelated VCPU. Second, we will only be able to spy on the victim when assigned to the same PCPU, which may coincide with only some fraction of the victim's execution.

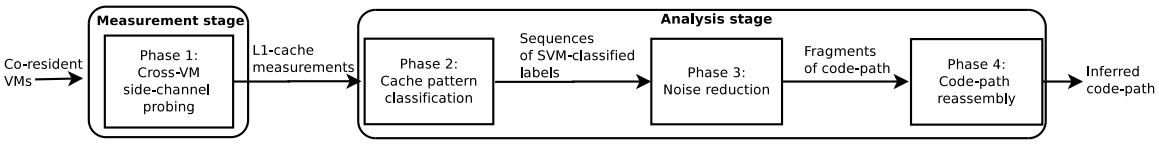


Figure 3.1: Diagram of the main steps in the cross-VM side-channel attack

In Section 3.4, we describe how the HMM mentioned above can be modified to filter out sequences corresponding to unrelated observations. In Section 3.5, we provide a dynamic programming algorithm, like those in bioinformatics, to "stitch" together multiple inferred code-path fragments output by the SVM+HMM and thereby construct fuller hypothesized code-paths. By observing multiple executions of the victim, we can gather sufficiently many candidate sequences to, through majority voting, output a full code path with negligible errors.

Putting it all together. Our full attack pipeline is depicted in Figure 3.1. The details of our measurement stage that addresses the first challenge described above are presented in Section 3.2. The analysis of these measurements to address the second and third challenges is then broken down into three phases: cache-pattern classification (Section 3.3), noise reduction (Section 3.4), and code-path reassembly (Section 3.5).

3.2 Cross-VM Side Channels

In this section, we demonstrate how an access-driven cross-VM side channel can be constructed on the L1 instruction cache in a modern x86 architecture running Xen. We have already described the basic PRIME-PROBE protocol for conducting the proposed side-channel attack (see Section 2.1). A fundamental difficulty in an I-cache attack without SMT support, however, is for the attacker VCPU to regain control of the PCPU resource sufficiently frequently (i.e., after the desired PRIME-PROBE interval has passed). To accomplish this, we leverage the tendency of the Xen credit scheduler to give the highest run priority to a VCPU that receives an interrupt. That is, upon receiving an interrupt, the attacker VCPU will preempt another guest VCPU running on the PCPU, provided that it is not also running with that highest priority (as a compute-bound victim would not be). As such, our attack strategy is to deliver an interrupt to the attacker VCPU every PRIME-PROBE interval.

We consider three types of interrupts to "wake up" the attacking VCPU: timer interrupts, network interrupts and IPIs. A fourth option is high precision event timer interrupts as used by Gullasch

et al. (2011), but these are not available to guests in Xen. Timer interrupts in a guest OS can be configured to be raised with a frequency of at most 1000Hz, but this is not sufficiently granular for our attack targets (e.g., a cryptographic key). Network interrupts, as used in OS level CPU-cycle stealing attacks by Tsafrir et al. (2007), can achieve higher resolution, but in our experiments the delivery times of network interrupts varied due to batching and network effects, rendering it hard to achieve microsecond-level granularity.

We therefore turn to IPIs. In SMP systems, an IPI allows one processor to interrupt another processor or even itself. It is usually issued through an advanced programmable interrupt controller (APIC) by one core and passed to other cores via either the system bus or the APIC bus. To leverage IPIs in our attack, another attacker VCPU, henceforth called the *IPI VCPU*, executes an endless loop that issues IPIs to the attacker VCPU which is conducting the PRIME-PROBE protocol, henceforth called the *probing VCPU*. This approach works generally well but is limited by two shortcomings.

First, due to interrupt virtualization by Xen, the PRIME-PROBE interval that can be supported through IPIs cannot be arbitrarily small. In our local testbed (see Section 3.6), we find it hard to achieve a PRIME-PROBE interval that is shorter than 50,000 PCPU cycles (roughly 16 microseconds). More frequent interrupts will be accumulated and delivered together. Second, if the IPI VCPU is descheduled then this can lead to periods during which no usable observations are made. If the Xen scheduler is non-work-conserving—meaning that a domain's execution time is limited to a budget, dictated by the *cap* and *weight* parameters assigned to it by Xen—then the IPI VCPU will be descheduled when it exceeds its budget. These periods then must be detected and any affected PRIME-PROBE instances discarded. However, if the scheduler is work-conserving (the default) and so allows a domain to exceed its budget if no other domain is occupying the PCPU, then descheduling is rare on a moderately loaded machine. It is worth noting that the probing VCPU executes so briefly that its execution appears not to be charged toward its budget by the current Xen scheduler.

Reducing noise due to irrelevant domains. As discussed in Section 2.1.4.2, the PRIME-PROBE measurements in the L1 instruction cache may subject to noise due to irrelevant domains. We discuss our solution to this hurdle in detail in Section 3.4, though even with our solution described there, it is beneficial if we can minimize the frequency with which PROBE results reflect a VM other than the victim's. In the configurations we will consider in Section 3.6, if Dom0 is idle then the Xen

scheduler will move the attacker and victim VCPU's to distinct cores most of the time. Thus, an effective strategy is to induce load on Dom0: if multiple victim VCPUs and the IPI VCPU are also busy and so together with Dom0 consume all four cores of the machine, then the probing VCPU, by relinquishing the PCPU frequently, invites another VCPU to share its PCPU with it. When the co-resident VCPU happens to be the victim's VCPU that is performing the target computation, then the PROBE results will be relevant to the attacker.

Since Dom0 is responsible for handling network packets, a general strategy to load Dom0 involves sending traffic at a reasonably high rate to an unopened port of the victim VM and/or attacker VM from a remote source. This can be especially effective since traffic filtering (e.g., via `iptables`) and shaping are commonly implemented in Dom0. In some cases (e.g., Amazon AWS), the attacker can even specify filtering rules to apply to traffic destined to his VM, and so he can utilize filtering rules and traffic that will together increase Dom0's CPU utilization.

3.3 Cache Pattern Classifier

In this section as well as the sections that follow, we introduce a set of techniques that, when combined, can enable an attacker VM to learn the code path used by a co-resident victim VM. In settings where control flow is dependent on confidential data, this enables exfiltration of secrets across VM boundaries. While the techniques are general, for concreteness we will use as a running example the context of cryptographic key extraction and, in particular, learning the code path taken when using the classic square-and-multiply algorithm. This algorithm (and generalizations thereof) have previously been exploited in access-driven attacks in non-virtualized settings, e.g., (Percival, 2005; Aciiçmez, 2007), but not in virtualized SMP systems as we explore here.

The square and multiply algorithm is depicted in Figure 3.2. It efficiently computes the modular exponentiation $x^s \bmod N$ using the binary representation of e, i.e., $e = 2^{n-1}e_n + \cdots + 2^0 e_1$. It is clear by observation that the sequence of function calls in a particular execution of `SquareMult` directly leaks e, which corresponds to the private key in many decryption or signing algorithms. We let M, S, and R stand for calls to `Mult`, `Square`, and `Reduce`, respectively, as labeled in Figure 3.2. Thus, the sequence $SRMRSR$ corresponds to exponentiation by $e = 2$.

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← Reduce(y, N)             (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← Reduce(y, N)           (R)
    }
  }
  return y
```

Figure 3.2: The square-and-multiply algorithm of modular exponentiation

The techniques we detail in the next several sections show how an attacker can, despite VMM isolation, learn such sequences of operations.

Recall from Section 3.2 that the output of a single PRIME-PROBE instance is a vector of timings, one timing per cache set. The first step of our algorithm is to classify each such vector as indicating a multiplication (M), modular reduction (R) or squaring (S) operation. To do so in our experiments, we employ a multiclass support vector machine, specifically that implemented in `libsvm` (Chang and Lin, 2011). An SVM is a supervised machine learning tool that, once trained, labels new instances as belonging to one of the classes on which it was trained. It also produces a *probability estimate* in $(0, 1]$ associated with its classification, with a number closer to 1 indicating a more confident classification.

To use an SVM to classify new instances, it is necessary to first train the SVM with a set of instance-label pairs. To do so, we use a machine with the same architecture as the machine on which the attack will be performed and configure it with the same hardware settings. We then install a similar software stack for which we have total control of the hypervisor. To collect our training data, we create a victim VM and attacker VM like those one would use during an attack. We use the `xm` command-line tools in Dom0 to pin the VCPUs of the victim VM and attacker's probing VCPU to the same PCPU. We then set the victim VM to repeatedly performing modular exponentiations with the same arguments and, in particular, with an exponent of all 1's, and the probing VCPU to repeatedly performing PRIME-PROBE instances.

This allows for the collection of vectors, one per PRIME-PROBE instance, but there remains the challenge of accurately labeling them as M, S or R. To do so, we need to establish communication from the victim VM to the attacker VM to inform the latter of the operation being performed (multiplication, squaring, or modular reduction) at any point in time. However, this communication should take as little time as possible, since if the communication takes too long, measurements collected from the PRIME-PROBE trials during training would differ from those in testing.

We employ cross-VM shared memory for this communication. Briefly, Xen permits different domains to establish memory pages shared between them (see (Chisnall, 2007, Ch. 4)). We utilize this shared memory by modifying the victim VM to write to the shared memory the type of operation (M, S, or R) immediately before performing it, and the attacker VM reads this value immediately after completing each PROBE of the entire cache.

Another challenge arises, however, which is how to modify the victim VM's library that performs the exponentiation while keeping other parts of the library unchanged. Adding the shared-memory writes prior to compilation would change the layout of the binary and so would ruin our PROBE results for the purpose of training. Instead, we prepare the instructions for shared-memory writing in a dynamic shared library called `libsync`. Then we instrument the binary of the exponentiation library to hook the `Square`, `Mult`, and `Reduce` functions and redirect each to `libsync`, which simply updates the shared memory and jumps back to the `Square`, `Mult` or `Reduce` function, as appropriate. Because the `libsync` and victim's library are compiled independently, the address space layout of the latter remains untouched. Even so, the memory-writing instructions slightly pollute the instruction cache, and so we exclude the cache sets used by these instructions (three sets out of 64).

3.4 Noise Reduction

There are two key sources of noise that we need to address at this phase of the key extraction process. The first is the classification errors of the SVM. Noise arising from events such as random fluctuations in probe timings causes the majority of SVM classification errors. Incomplete PRIME-PROBE overlap with exponentiation operations occasionally creates ambiguous cache observations, for which no classification is strictly correct. The second is the presence of PRIME-PROBE results and,

in turn, SVM outputs that simply encode no information of interest to the attacker for other reasons, e.g., because the victim VCPU had migrated to a different PCPU from the attacker or because it was performing an operation not of interest to the attacker. Consequently, we develop a sequence of mechanisms to refine SVM outputs to reduce these sources of noise.

3.4.1 Hidden Markov Model

We start by employing a hidden Markov model to eliminate many of the errors in SVM output sequences. (We assume reader familiarity with HMM basics. For an overview, see, e.g., (Bishop, 2007).) Our HMM models the victim's exponentiation as a Markov process involving transitions among hidden "square," "multiply," and "reduce" states, respectively representing `Square`, `Mult`, and `Reduce` operations. As exponentiation is executed by the victim, labels output by the attacker's SVM give a probabilistic indication of the victim's hidden state.

As the SVM outputs multiple labels per `Square`, `Mult`, or `Reduce` operation, we represent an operation as a *chain* of hidden states in the HMM. As is the case with any error-correcting code, the redundancy of SVM output labels helps rectify errors. Intuitively, given multiple labels per operation (e.g., induction of correct sequence $SSSSS$ by a `Square` operation), the HMM can correct occasional mislabelings (e.g., the outlying M in $SSSMS$). The HMM also corrects errors based on structural information, e.g., the fact that a square or multiply is always followed by a modular reduction.

While the HMM takes as input a sequence of SVM labels, each such label, S, M, or R, is first mapped into an expanded label set that reflects its corresponding level of SVM confidence. Given a "high" SVM confidence, in the range $[0.8, 1.0]$, a label remains unchanged. For "medium" confidence, lying within $[0.6, 0.8)$, a label is transformed into a different label indicating this medium confidence; i.e., S, M, and R are mapped respectively to new labels s, m, and r. Finally, given a "low" confidence, in $[0, 0.6)$, any of S, M, or R is mapped to a generic, "low confidence" label L.

In brief, then, the SVM output label set $\{S, M, R\}$ is expanded, through coarse integration of SVM confidence measures, into a set of seven labels $\{S, M, R, s, m, r, L\}$. This expanded set constitutes the emission labels of the HMM. We found that hand-tuning the transition and emission probabilities in the HMM resulted in better performance, i.e., fewer errors in HMM decoding, than

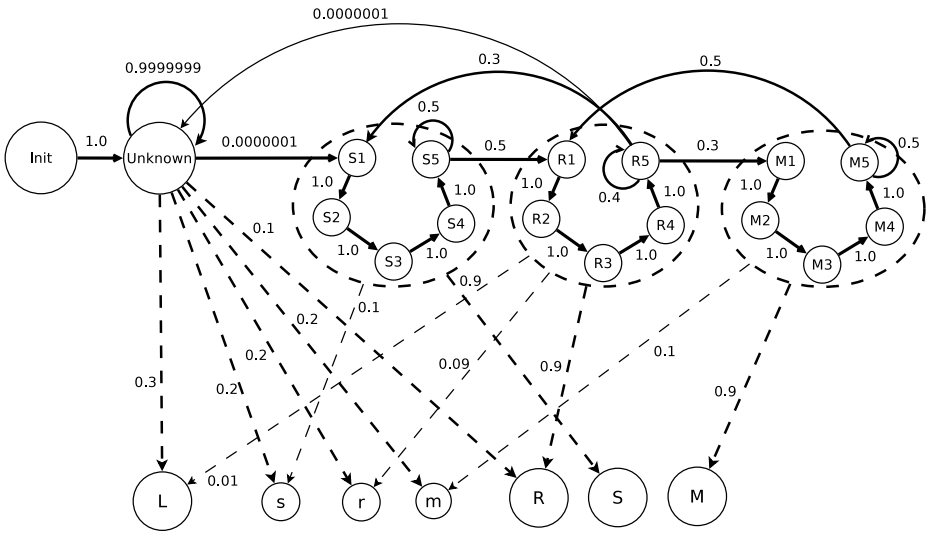


Figure 3.3: Diagram of the HMM used in our experiments with 4096-bit base x and modulus N. Emission labels are depicted in the lower half, hidden states in the upper half. Solid arrows indicate transitions, dotted arrows denote emissions. Emission probabilities below 0.01 are omitted.

training via the Baum-Welch algorithm (Baum et al., 1970). The full HMM, with hidden states, emission labels, and transition and emission probabilities, is depicted in Figure 3.3.

Given this HMM and a sequence of expanded-label SVM emissions, we run a standard Viterbi algorithm (Viterbi, 1967) to determine the maximum likelihood sequence of corresponding hidden states. The result is a sequence of labels from the hidden-state set $\{$*S1*, . . ., *S5*, *M1*, . . . , *M5*, *R1*, . . . , *R5*, *Unknown*$\}$. We refer to this as the HMM output sequence.

3.4.2 Post-Processing HMM Outputs

We post-process HMM output sequences to remove state labels that are redundant or agnostic to key-bit values. The states in an HMM operator chain (e.g., *S1* . . . *S4* *S5*$^+$, where "*S5*$^+$" denotes one or more occurrences of *S5*) collectively indicate only a single operation (e.g., one instance of `Square`). Thus, our post-processing step replaces every chain of the form *S1* . . . *S4* *S5*$^+$ with a single *S* and every chain of the form *M1* . . . *M4* *M5*$^+$ with a single *M*. *Unknown* states carry no information about key bit values and so are discarded.

We found it necessary to post-process *R1* . . . *R4* *R5*$^+$ chains in a somewhat more refined manner. Recall that `Reduce` operations are key-agnostic, and so we discarded such chains, provided that they were short. The HMM output sequence, however, would sometimes include long chains of the form

R1 ... *R4* $R5^+$, which would typically signal a cryptographic observation that passed unobserved. Any such chain of sufficient length was thus replaced in post-processing with a $*$, indicating a hypothesized omitted `Mult` or `Square`. In our experiments described in Section 3.6, we did so for chains of the form *R1* ... *R4* $R5^+$ of length 24 or more. Chains of length less than this were discarded.

Consequently, post-processing yields refined HMM outputs over the label set $\{S, M, *\}$.

3.4.3 Filtering out Non-Cryptographic HMM Outputs

Successful key reconstruction requires that we reliably identify and retain observed sequences of cryptographic operations, i.e., those involving private key material, while discarding non-cryptographic sequences. Extraneous code paths arise when the victim migrates to a core away from the attacker or executes software independent of the cryptographic key.

The SVM in our attack does not include a label for extraneous code paths. (Non-cryptographic code constitutes too broad a class for effective SVM training.) Long non-cryptographic code paths, however, are readily distinguishable from cryptographic code paths based on the following observation. A random private key—or key subsequence—includes an equal number of 0 and 1 bits in expectation. As a 1 induces a square-and-multiply, while a 0 induces a squaring only, the expected ratio of corresponding *S* to *M* labels output by the SVM, and thus the HMM, is 2:1. Thus, a reasonably long key subsequence will evidence approximately this 2:1 ratio with high probability. In contrast, a non-cryptographic code path tends to yield many *Unknown* states and therefore outputs sequences that are generally discarded, or in rare cases yields short sequences with highly skewed *S*-to-*M* ratio.

The following elementary threshold classifier for cryptographic versus non-cryptographic post-processed HMM output sequences proves highly accurate. Within a given HMM output sequence, we identify all subsequences of *S* and *M* labels of length at least α, for parameter α. (In other words, we disregard short subsequences, which tend to be spurious and erroneously skew *S*-to-*M* ratios.) We count the total number a of *S* labels and b of *M* labels across all of these subsequences, and let $a/(b+1)$ represent the total *S*-to-*M* ratio. (Here "$+1$" ensures a finite ratio.) If this *S*-to-*M* ratio falls within a predefined range $[\rho_1, \rho_2]$, for parameters ρ_1 and ρ_2, with $0 < \rho_1 < 2 < \rho_2$, the output sequence is classified as a cryptographic observation. Otherwise, it is classified as non-cryptographic.

We found that we could improve our detection and filtering of inaccurate cryptographic sequences by additionally applying a second, simple classifier. This classifier counts the number of MM label pairs in an HMM output sequence. As square-and-multiply exponentiation never involves two sequential multiply operations—there is always an interleaved squaring—such MM pairs indicate a probable erroneous sequence. Thus, if the number of MM pairs exceeds a parameter β, we classify the output sequence as inaccurate and discard it.

We applied these two classifiers (S-to-M ratio and MM-pair) to all HMM output sequences, and discarded those classified as non-cryptographic. The result is a set of post-processed, filtered HMM outputs, of which an overwhelming majority represented observed cryptographic operations, and whose constituent labels were largely correct.

3.5 Code-path reassembly

Recall that a major technical challenge in our setting is the fact that the victim VM's VCPUs float across physical cores. This movement frequently interrupts attacker VM PRIME-PROBE attempts, and truncates corresponding HMM output sequences. It is thus helpful to refer to the post-processed, filtered HMM outputs as *fragments*.

Fragments are short, more-or-less randomly positioned subsequences of hypothesized labels for the target key operations. Despite the error-correcting steps detailed above, fragments also still contain a small number of erroneous S and M labels as well as $*$ labels. The error-correcting steps detailed in Section 3.4 operate *within* fragments. In the final, sequence-reconstruction process described here, we correct errors by comparing labels *across* fragments, and also "stitch" fragments together to achieve almost complete code-path recovery. This will reveal most of the key sequence; simple brute forcing of the remaining bits reveals the rest.

Accurate sequence alignment and assembly of fragments into a full key-spanning label sequence is similar to the well-known sequence-reconstruction problem in bioinformatics. There are many existing tools for DNA sequencing and similar tasks, such as Celera Assembler and ClustalW2. However, various differences between that setting and ours, in error rates, fragment lengths, etc., have rendered these tools less helpful than we initially hoped, at least so far. We therefore developed our own techniques, and leave improving them to future work.

In this final, sequence-reconstruction step of key recovery, we partition fragments into batches. The number of batches ζ and number of fragments θ per batch, and thus the total number $\zeta\theta$ of fragments that must be harvested by the attacker VM, are parameters adjusted according to the key-recovery environment. It is convenient, for the final stage of processing ("Combining spanning sequences," see below) to choose ζ to be a power of three.

Our final processing step here involves three stages: inter-fragment error correction, fragment stitching to generate sequences that span most of the code-path, and then a method for combining spanning sequences to provide an inferred code-path. The first two stages operate on individual batches of fragments, as follows:

3.5.1 Cross-Fragment Error-Correction

In this stage, we correct errors by comparing labels across triples of fragments.

First, each distinct pair of fragments is aligned using a variant of the well-known dynamic programming (DP) algorithm for sequence alignment (Needleman and Wunsch, 1970). We customize the algorithm for our setting in the following ways. First, we permit a $*$ label to match either a *S* or an *M*. Second, because two fragments may reflect different, potentially non-intersecting portions of the key, terminal gaps (i.e., inserted before or after a fragment) are not penalized. Third, a contiguous sequence of nonterminal gaps is penalized quadratically as a function of its length, and a contiguous sequence of matches is rewarded quadratically as a function of its length.

We then construct a graph $G = (V, E)$ in which each fragment is represented by a node in V. An edge is included between two fragments if, after alignment, the number of label matches exceeds an empirically chosen threshold γ. Of interest in this graph are *triangles*, i.e., cliques of size three. A triangle (v_1, v_2, v_3) corresponds to three mutually overlapping fragments / nodes, v_1, v_2, and v_3, and is useful for two purposes.

First, a triangle permits cross-validation of pairwise alignments, many of which are spurious. Specifically, let k_{12} be the first position of v_1 to which a (non-gap) label of v_2 aligned; note that k_{12} could be negative if the first label of v_2 aligned with an initial terminal gap of v_1, and similarly for k_{13} and k_{23}. If $|(k_{13} - k_{12}) - k_{23}| \leq \tau$, where τ is an algorithmic parameter (5 in our experiments), then the alignments are considered mutually consistent. (Intuitively, $k_{13} - k_{12}$ is a measure of alignment between v_2 and v_3 with respect to v_1, while k_{23} measures direct alignment between v_2 and v_3. Given

perfect alignment, the two are equal.) Then, the triangle is *tagged* with the "length" of the region of intersection among v_1, v_2 and v_3

The second function of triangles is error-correction. Each triangle (v_1, v_2, v_3) of G is processed in the following way, in descending order. Each position in the region of intersection of v_1, v_2 and v_3 has three corresponding labels (or gaps), one for each fragment. If two are the same non-gap label, then that label is mapped onto all three fragments in that position. In other words, the three fragments are corrected over their region of intersection according to majority decoding over labels.

Cross-fragment error-correction changes neither the length nor number of fragments in the batch. It merely reduces the global error rate of fragment labels. We observe that if the mean error rate of fragments, in the sense of edit distance from ground truth, is in the vicinity of 2% at this stage, then the remaining processing results in successful key recovery. We aim at this mean error rate in parameterizing batch sizes (θ) for a given attack environment.

3.5.2 Fragment Stitching

In this next processing stage, a batch of fragments is assembled into what we call a *spanning sequence*, a long sequence of hypothesized cryptographic operations. In most cases, the maximum-length spanning sequence for a batch covers the full target key.

The DP algorithm is again applied in this stage to every pair of fragments in a batch, but now customized differently. First, terminal gaps are still not penalized, though a contiguous sequence of matches or (nonterminal) gaps accumulates rewards or penalties, respectively, only linearly as a function of its length. This is done since the fragments are presumably far more correct now, and so rewarding sequences of matches superlinearly might overwhelm any gap penalties. Second, the penalty for each nonterminal gap is set to be very high relative to the reward for a match, so as to prevent gaps unless absolutely necessary.

Following these alignments, a directed graph $G' = (V', E')$ is constructed in which each node in V' (as in V above) represents a fragment. An edge (v_1, v_2) is inserted into E' for every pair of fragments v_1 and v_2 with an alignment in which the first label in v_2 is aligned with some label in v_1 after the first. (Intuitively, v_2 overlaps with and sits to the "right" of v_1.) Assuming, as observed

consistently in our experiments, that there are no alignment errors in this process, the resulting graph G' will be a directed *acyclic* graph (DAG).[2]

A path of fragments / nodes $v_1, v_2, \ldots, v_m \in V'$ in this graph is stitched together as follows. We start with a source node v_1 and append to it the non-overlapping sequence of labels in v_2, i.e., all of the labels of v_2 aligned with the ending terminal gaps of v_1, if any. (Intuitively, any labels in v_2 positioned to the "right" of v_1 are appended to v_1.) We build up a label sequence in this way across the entire path. The resulting sequence of labels constitutes a spanning sequence. We employ a basic greedy algorithm to identify the path in G' that induces the maximum-length spanning sequence.

3.5.3 Combining Spanning Sequences

The previous stages, applied per batch, produce ζ spanning sequences. The ζ spanning sequences emerging from the fragment stitching stage are of nearly, but not exactly, equal length, and contain some errors. For the final key-recovery stage, we implement an alignment and error-correction algorithm that proceeds in rounds. In each round, the sequences produced from the previous round are arbitrarily divided into triples, and each triple is reduced to a single spanning sequence that is carried forward to the next round (and the three used to create it are not). For this reason, we choose ζ, the number of batches, to be a power of three, and so we iterate the triple-merging algorithm $\log_3 \zeta$ times. The result is a sequence of hypothesized cryptographic operations covering enough of the target key to enable exhaustive search over all possibilities for any remaining $*$ values.

Each round proceeds as follows. Each triple (s_1, s_2, s_3) is first aligned using a basic three-way generalization of DP (e.g., see (Gautham, 2006, Section 4.1.4)). This may insert gaps (rarely consecutively) into the sequences, yielding new sequences (s'_1, s'_2, s'_3). Below we denote a gap so inserted by the "label" $\sqcup$. The algorithm is parameterized to prevent alignment of ***S*** and ***M*** labels in the same position within different spanning sequences.

To the resulting aligned sequence triple (s'_1, s'_2, s'_3), the length of which is denoted as ℓ, is applied a modified majority-decoding algorithm that condenses the triple into a single, merged output sequence. We say that s'_1, s'_2, and s'_3 *strongly agree* at position j if all three sequences have identical labels at position j or, for $1 < j < \ell$, if any two of the three have identical labels at each of positions

[2] A cycle in this graph indicates the need to adjust parameters in previous stages and retry.

$j-1$, j, and $j+1$. In this step, the output sequence adopts a label at position j if the three strongly agree on that label at position j and the label is S, M, or, in the last round, $\sqcup$. Otherwise, the output sequence adopts $*$ at position j. At the end of the last round, any residual $\sqcup$ labels are removed.

3.6 Evaluation

We performed a case study using the `libgcrypt` v.1.5.0 cryptographic library. This was the most recent version of `libgcrypt` as of May 2012; our results extend to cover earlier versions as well. To be concrete, we also fixed an application that uses the library: Gnu Privacy Guard (`GnuPG`) v.2.0.19 (`http://www.gnupg.org/`). `GnuPG` is used widely for encrypting and signing email, but we note that `libgcrypt` use goes beyond just `GnuPG`. The attack should extend to any application using the vulnerable routines from `libgcrypt`.

ElGamal encryption. Manual code review revealed that `libgcrypt` employs a more-or-less textbook variant of the square-and-multiply modular exponentiation algorithm for use with cryptosystems such as RSA (Rivest et al., 1978) and ElGamal (ElGamal, 1985). Our case study focuses on the latter.

ElGamal encryption in `libgcrypt` uses a cyclic group $\mathbb{Z}_p^*$ for prime p and generator g. The bit length size of p is dictated by a user-specified security parameter κ. Given g and p, a secret key is chosen uniformly at random to be a non-negative integer x whose bit length is, for example, 337, 403, or 457 when κ is 2048, 3072, or 4096, respectively. We note that this deviates from standard ElGamal, in which one would have $|x| \approx |p|$. The smaller exponent makes decryption faster. The public key is set to be $X = g^x \bmod p$.

To encrypt a message $M \in \mathbb{Z}_p^*$, a new value $r \in \mathbb{Z}_m$, for $m = 2^{|x|}$, is chosen at random; the resulting ciphertext is $(g^r, X^r \cdot M)$. (Typically M is a key for a separate symmetric encryption mechanism.) Decryption of a ciphertext (R, Y) is performed by computing $R^x \bmod p$, inverting it modulo p, and then multiplying Y by the result modulo p.

Our attack abuses the fact that computation of $R^x \bmod p$ during decryption is performed using the square-and-multiply modular exponentiation algorithm. The pseudocode of Figure 3.2 is a close proxy of the code used by `libgcrypt`.

Parameter	Section 3.6.1	Section 3.6.2
$[\rho_1, \rho_2]$	$[1, 4]$	$[1, 4]$
α	200	100
β	5	5
ζ	9	9
θ	30	35
τ	5	5
γ	100	50

Table 3.1: Parameter settings in cross-VM attack evaluation

3.6.1 With a Work-Conserving Scheduler

Experiment settings. We evaluated our attack in a setting in which two paravirtualized guest VMs, each of which possesses two VCPUs, co-reside on a single-socket quad-core processor, specifically an Intel Core 2 Q9650 with an operating frequency of 3.0GHz. The two guest VMs and Dom0 were each given *weight* 256 and *cap* 0; in particular, this configuration is work-conserving, i.e., it allows any of them to continue utilizing a PCPU provided that no other domain needs it. Dom0 was given a single VCPU. One guest VM acted as the victim and the other as the attacker. We ran Xen 4.0 as the virtualization substrate, with `rdtsc` emulation disabled. Both VMs ran an Ubuntu 10.04 server with a Linux kernel 2.6.32.16. The size of the memory in the guest VMs was large enough to avoid frequent page swapping and so was irrelevant to the experiments. The victim VM ran `GnuPG` v.2.0.19 with `libgcrypt` version v.1.5.0, the latest versions as of this writing. The victim's ElGamal private key was generated with security parameter $\kappa = 4096$. Other parameters for our attack are shown in Table 3.1.

In general, the attacker can either passively wait for periods where it shares a PCPU with the victim, or can actively "create" more frequent and longer such periods on purpose. To abbreviate our experiment, we assumed a situation that is to the attacker's advantage (but is nevertheless realistic), in which both Dom0 and one victim VCPU are CPU-bound, running non-cryptographic computational tasks. These conditions maximized the frequency with which the attacker and the *other* victim VCPU share a PCPU. As discussed in Section 3.2, Dom0 can be loaded by, for example, forcing it to analyze a high rate of traffic with expensive filtering rules. We experimented with several such scenarios (varying in the number of rules and packet rates), as well as other situations that would encourage the attacker and victim to share a PCPU (e.g., dedicating one core to Dom0, many of which gave results similar to those reported here.

	S	*M*	*R*
S	.91	.00	.09
M	.01	.92	.07
R	.02	.01	.97

Table 3.2: Confusion matrix of SVM in cross-VM attack evaluation

Another way in which we were generous to the attacker in this demonstration was that we assumed the victim VM would often perform ElGamal decryption with the target key—e.g., because the attacker had the ability to remotely trigger decryption, as might be realistic for a network service that the attacker could invoke—and that this decryption executed on the victim VCPU that was not already compute-bound. As such, in our demonstration, the attacker did not have to wait indefinitely for a private-key decryption, but rather the victim performed decryptions over and over. Given our ability to filter non-cryptographic observations, less frequent exponentiations would slow down, but not prevent, the attack. In fact, it is worth noting that exponentiation with the private exponent under attack constitutes only roughly 2% of the execution time of a private-key decryption, and so even in this demonstration, 98% of victim execution was irrelevant to our attack and filtered out by our techniques.

Experiment results. Our SVM was trained as per the procedure discussed in Section 3.3 with PRIME-PROBE results from $30,000$ `Square` operations, $30,000$ `Mult` operations, and $80,000$ `Reduce` operations. We skewed the training data toward `Reduce` operations to minimize `Reduce` operations being misclassified as `Square` or `Mult` operations. A three-fold cross validation resulted in the confusion matrix shown in Table 3.2.

In the attack, we performed $300,000,000$ PRIME-PROBE trials in chunks of $100,000$. The data collection lasted about six hours, during which roughly 1000 key-related fragments were recovered from our HMM (Section 3.4). Of these, 330 key-related fragments had length at least α. The lengths of these fragments are shown in Figure 3.4(a). Let the *accuracy* of a fragment be defined as 1 minus the normalized edit distance of the fragment from ground truth, i.e., the edit distance divided by the length of the fragment. On average, these fragments had 0.958 accuracy with a standard deviation of 0.0164.

We then combined fragments (Section 3.5) to produce spanning sequences. The accuracy of these spanning sequences was a function of the number of fragments in each batch, as shown in

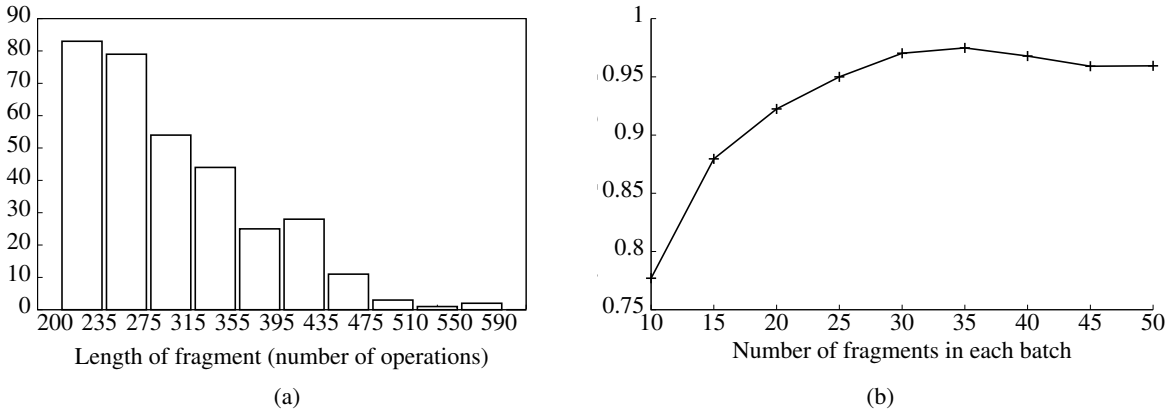


Figure 3.4: Experiment results with work-conserving scheduler (Section 3.6.1). (a) Frequency of fragment lengths extracted, each bar represents the number of fragments whose length falls between the x-axis labels. (b) Accuracy of spanning sequences as a function of number of fragments in a batch.

Figure 3.4(b). We chose to use batches of 30 fragments each, yielding an average spanning sequence accuracy of 0.981. The final step was to combine the spanning sequences (end of Section 3.5). The resulting key had no erasures, insertions or replacements. The only uncertainties arose from $*$ labels at the two ends or occasionally in the middle, each representing "no-op" (a spurious operation) or a single `Square` or `Mult` operation. This left us needing to perform a brute-force search for the uncertain bits, but the search space was only $9,862$ keys.[3]

3.6.2 With a Non-Work-Conserving Scheduler

We also evaluated the attack for a non-work-conserving setting of the Xen scheduler which is configured as *weight* = 256 and *cap* = 80 (and other parameters as shown in Figure 3.1). The induced workload in Dom0 and the victim remains the same as in the previous section. Recall from Section 3.2 that this is a more difficult case for our attack, since it causes the IPI VCPU to be descheduled more aggressively, which in turn interferes with initiating an IPI to the probing VCPU. When this occurs, the corresponding PRIME-PROBE result must be discarded. Therefore, if the scheduler is non-work-conserving, data collection takes longer and the fragments resulting from our HMM tend to be shorter.

[3]Rather than searching over all three possible operation assignments to each $*$ symbol, we prune the search space by grouping assignments into functional equivalence classes; e.g., (`Square`, no-op) is equivalent to (no-op, `Square`).

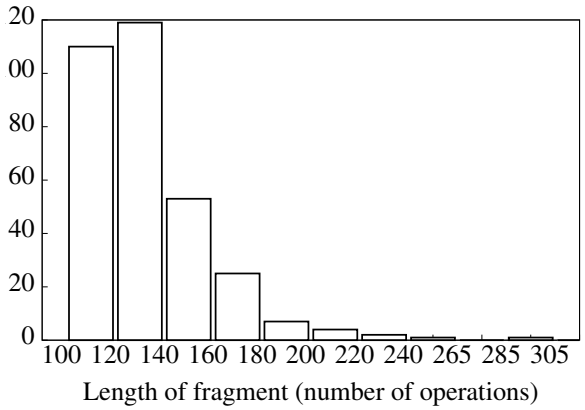


Figure 3.5: Fragment lengths, non-work-conserving scheduler (Section 3.6.2)

These effects are demonstrated in Figure 3.5, which shows the fragment lengths at the same stage of processing as is reflected in Figure 3.4(a) for the work-conserving case. Despite the fact that these fragments are based on $1,900,000,000$ PRIME-PROBE trials (collected during about 45 hours), over six times the number we collected in Section 3.6.1, only 322 fragments of length at least α resulted—an order of magnitude less than the work-conserving case. And this occurred despite the fact that we set α to only 100 in the non-work-conserving case, i.e., half of the value in Section 3.6.1. These 322 fragments yielded 9 spanning sequences with average accuracy 0.98, which were "stitched" together into a single key with only a few missing bits, yielding a search space of only 6615 keys.

3.7 Discussion

3.7.1 Applicability to Other Targets

In this chapter, we've demonstrated the use of a cache-based side channel to extract private ElGamal keys from the `libgcrypt` library across VM boundaries. Conceptually, similar approaches can be applied to other target algorithms (such as RSA, AES) in other cryptographic libraries (e.g., OpenSSL). However, we note here that some gaps exist in extending the attacks directly to other targets. In order to break RSA keys, further steps need to be taken in the "code-path reassembly" stage, as RSA algorithms adopt the Chinese Remainder Theorem, which splits one modular exponentiation into two. Consequently, additional algorithms are required to effectively distinguish the code-path fragments of the two exponents collected from the side channel. Symmetric encryption schemes

such as AES run much faster than asymmetric encryptions, and therefore they demand more frequent context switchs to perform PRIME-PROBE attacks. The granularity of the PRIME-PROBE achieved in our work is not sufficient for symmetric key extraction.

The PRIME-PROBE attacks adopted in our attack is a known threat to cryptographic implementations in traditional non-virtualized, single-core systems. Some cryptographic implementations have therefore patched the vulnerability which may render the side-channel analysis harder. For instance, the implementation of modular exponentiations in OpenSSL utilizes a sliding window protocol, which bundles multiple multiply operations and therefore fewer exponent bits can be extracted from the PRIME-PROBE side channel. However, we believe this is an orthogonal issue to our research. Should new approaches to breaking OpenSSL implementations be available in non-virtualized, single-core systems, the attack pipeline proposed in this chapter, or similar approaches, should in theory be able to extend them to cross-VM settings in multi-core platforms.

3.7.2 Countermeasures

As discussed in Section 2.3, there are multiple avenues for possible defenses against cross-VM side-channels. Here we detail a few potential defense approaches and discuss their benefits and downsides.

Avoiding co-residency. In high-security environments, a longstanding practice is to simply not use the same computer to execute tasks that must be isolated from each other, i.e., to maintain an "air gap" between the tasks. This remains the most high-assurance defense against side-channel (and many other) attacks. But this would obviate many of the current and future uses of VMs, including public clouds that multiplex physical servers such as Amazon EC2, Windows Azure, and Rackspace, and the other VM-powered applications discussed in the introduction. In the cloud setting, for example, Amazon EC2 offers *dedicated instances* on which customers can execute their VMs with a promise that no other customers' VMs will execute on them simultaneously. This approach comes at significant cost, however, since it requires isolated hardware resources. (Consequently, for example, EC2's dedicated instances cost an additional $10 per hour.) However, the compliance of such SLA by the cloud provider need to be verified by the cloud tenants due to the high security requirements and

expenses. We propose HomeAlone to solve this cloud verification problem, which will be discussed in Chapter 4.

Side-channel resistant algorithms. There exists a long line of work on cryptographic algorithms designed to be side-channel resistant (e.g., (Page, 2005; Bernstein, 2005; Percival, 2005; Tromer et al., 2010)). Recent versions of some cryptographic libraries attempt to prevent the most egregious side-channels; e.g., one can use the Montgomery ladder algorithm (Montgomery, 1987) for exponentiation or even a branchless algorithm. But these algorithms are slower than leakier ones, legacy code is still in wide use (as exhibited by the case of `libgcrypt`), and proving that implementations are side-channel free remains beyond the scope of modern techniques. Moreover, our techniques are applicable to non-cryptographic settings where there are few existing mechanisms for preventing side-channels.

Core scheduling. Another defense might seek to modify scheduling to at least limit the granularity of interrupt-based side-channels. The current Xen credit scheduler optimizes low latency at the cost of allowing frequent interrupts, even by non-malicious programs. Newer Xen release ($>$ version 4.2) already have plans to modify the way interrupts are handled, allowing a VCPU to preempt another VCPU only when the latter has been running for a certain amount of time (default being $1\mathrm{ms}$). This will reduce our side-channel's measurement granularity, but not eliminate the side-channel. Coarser side channels may already prove damaging (Ristenpart et al., 2009). A fundamental question for future work, therefore, is what interruption granularity best balances performance and security.

3.8 Summary

The use of virtualization to isolate a computation from malicious ones that co-reside with it is growing increasingly pervasive. This trend has been facilitated by the failure of today's operating systems to provide adequate isolation, the emergence of commodity VMMs offering good performance (e.g., VMWare, Xen, HyperV), and the growth of cloud facilities (e.g., EC2, Rackspace) that leverage virtualization to enable customers to provision computations and services flexibly. Given the widespread adoption of virtualization, it is thus critical that its isolation properties be explored and understood.

In this chapter, we have shed light on the isolation properties (or lack thereof) of a leading VMM (Xen) in SMP environments, by demonstrating that side-channel attacks with fidelity sufficient to exfiltrate a cryptographic key from a victim VM can be mounted. Ours is the first demonstration of such a side-channel in a virtualized SMP environment. Challenges that our attack overcomes include: preempting the victim VM with sufficient frequency to enable fine-grained monitoring of its I-cache activity; filtering out numerous sources of noise in the I-cache arising from both hardware and software effects; and core migration that renders many attacker observations irrelevant to the task of extracting the victim's key. Through a novel combination of low-level systems implementation and sophisticated tools such as classifiers (e.g., SVMs and HMMs) and sequence alignment algorithms, we assembled an attack that was sufficiently powerful to extract ElGamal decryption keys from a victim VM in our lab tests.

CHAPTER 4: CO-RESIDENCY DETECTION[1]

In the previous chapter, we presented an attack pipeline that exploits L1 caches as side channels to exfiltrate cryptographic keys across the VM boundaries. In light of such threat, an intuitive defense in multi-tenant public clouds is to provide physical isolation among cloud tenants with high security requirements. However, as explained previously in Section 1.2, verification of such promised physical isolation is difficult. In this chapter, we will describe a system called HomeAlone, which allows a cloud tenant to detect co-resident third-party VMs with high confidence and thereby verify its physical isolation without trusting the cloud provider. The chapter is laid out as follows. In Section 4.1, we describe the cloud scenarios envisaged for use of HomeAlone and the accompanying threat model. We detail the design of HomeAlone in Section 4.2 and its implementation in Section 4.3. In Section 4.4, we evaluate the detection accuracy of HomeAlone on demonstration workloads and the performance impact of HomeAlone. We discuss in Section 4.5 a number of issues that bear on the use of HomeAlone in practice. We summarize the chapter in Section 4.6.

4.1 Motivation and Threat Model

The security concerns surrounding cloud computing arise primarily in public clouds, although they carry over to private clouds that support disparate organizational functions. Multi-tenancy in public clouds creates sharing of resources by organizations that have potentially competing or conflicting interests and thus motivation to exfiltrate data from one another and/or disrupt one another's operations. While public clouds enforce logical isolation among tenants, they often multiplex tenants across hardware. This common practice presents a realistic threat of data theft or covert intelligence gathering in public clouds (see (Ristenpart et al., 2009) and Chapter 3).

Such concerns—and interest by organizations in extending their private clouds into public clouds (creating so-called *hybrid clouds*)—have prompted some tenants, e.g., U.S. federal agencies, to

[1]This chapter is excerpted from Zhang et al. (2011).

demand physical isolation as part of their SLAs (Carlson, 2010). Others use only resource instances that are meant to provide such isolation, such as full-physical-machine instances with Amazon Web Services (AmazonAWS, 2010).

Even for tenants whose cloud providers offer assurances of physical isolation, however, a problem remains. How can these tenants verify that their computing resources (and VMs, in particular) are *actually* physically isolated?

Given this challenge, HomeAlone is designed to provide two benefits in public clouds. First, the system allows tenants (or auditors acting on their behalf) to detect hardware co-residency with foe VMs. Thus HomeAlone enables tenants to detect and mediate the presence of potentially dangerous side channels in cloud computing environments. Second—and of perhaps equal importance—by merit of its detection of unexpected co-residency, HomeAlone can give insight into possible policy violations or system misconfigurations by cloud administrators. In other words, by way of detecting physical-isolation breaches, HomeAlone can serve as a sentinel for potentially broader and more serious systemic security lapses.

4.1.1 Threat Model

We consider an IaaS tenant that operates a collection of one or more (friendly) VMs co-resident on a given physical server. (Confirmation of friendly co-residency is obtainable via techniques outlined in, e.g., (Ristenpart et al., 2009).) The tenant presumes—on the basis of a service agreement with the cloud provider, for instance—that its VMs have exclusive use of the physical server. The tenant's goal is to disprove or confirm its hypothesis via the detection or non-detection of foe VMs.

The tenant has no control over or visibility into the functioning of the hypervisor. That is, its only view into platform resource allocation is the one presented by its VMs.

We model the cloud provider as neutral. The provider does not facilitate foe-VM detection by the tenant by, e.g., giving hypervisor access to the tenant. At the same time, the provider does not modify software or hardware specifically to disable the tenant's detection tools.[2] We consider two scenarios: (1) The "foe" VM is benign, i.e., oblivious to its co-residency, or at least not attempting to

[2]A cloud provider has little incentive to actively enable foe VMs to exfiltrate data via side channels: Its control of the infrastructure means that it can simply exfiltrate data via the hypervisor if it so chooses.

exploit it to attack friendly VMs; and (2) the foe VM is an active adversary seeking to exfiltrate data from friendly VMs.

Benign "foe VMs". Co-residency with a benign foe VM may arise due to an unintentional policy violation or a configuration error by the cloud provider (or perhaps an intentional, cost-cutting violation, but not one that the cloud provider compounds via active cover-up). Indeed, we anticipate that such errors will be more common in the cloud than targeted exfiltration attacks via co-residency.

The ability to detect policy violations that lead to non-adversarial co-residency is important for two reasons. The first is regulatory compliance. Server isolation is an established best practice, for instance, for PCI (Payment Card Industry) DSS (Data Security Standards) compliance. The second is the vulnerability that co-residency evidences. Even if foe VMs are not actively targeting co-resident friendly VMs, their existence highlights an isolation breach that can ultimately lead to a true compromise.

As we demonstrate, HomeAlone effectively detects the presence of a benign foe VM whose activities are significantly evidenced in the L2 cache during its execution. HomeAlone can thus serve as an early warning of accidental co-residency and potentially even as an index into more systemic security vulnerabilities.

Adversarial foe VMs. An adversarial foe VM is one that attempts to exploit its co-residency to exfiltrate sensitive data from friendly VMs. The benefit of HomeAlone in detecting such foe VMs is clear.

As a countermeasure to detection by HomeAlone, a foe VM could attempt to minimize its L2 cache footprint. Wholesale avoidance of the L2 cache for an actively executing foe VM would be challenging, as it would severely curtail use of memory (and necessitate avoidance of services, e.g., network transmission, that induce an L2 footprint). Specific avoidance of the region monitored by HomeAlone would also be challenging. As we shall see, this region is composed of a random selection of cache sets, and a foe VM attempting to map this region would ostensibly generate L2 activity that would itself facilitate detection by HomeAlone.

Moreover, the L2 cache is a side-channel attack vector of choice in server environments. Thus, a foe VM of particular concern is one that tries to exfiltrate data by actively probing this cache. As we demonstrate in our experiments, the L2-cache footprint produced by such a foe VM renders it

more easily detectable by HomeAlone. Conversely, cornering the attacker into avoiding the L2 cache in whole or in part would be a success: It would strip a foe VM of a major adversarial benefit of co-residency. Alternatively, to evade detection, a foe VM might attempt to limit its operation to short bursts or low-level activity over a prolonged period. This approach, however, would constrain exfiltration opportunities for critical, transient-use data such as cryptographic keys.

4.1.2 Alternative Approaches

Of course, with the cooperation of the cloud provider, it is possible for a tenant to detect foe VMs more directly (and reliably) than HomeAlone permits. For example, given control of the hypervisor, the tenant could list or enumerate the set of currently executing VMs on a physical machine. Such functionality, however, would require modification of a service provider's hypervisor software or management plane to permit queries from tenants remotely or from tenant VMs locally. Extensions of this type, while technically possible, introduce their own access-control challenges and would require adoption by cloud providers, which there is no reason at present to anticipate in public clouds. As such, we focus on solutions that do not require cloud provider support.

4.2 Designing a Co-Residency Detector

The PRIME-PROBE timing channel described in Section 2.1.3.1 potentially provides a method for a monitoring VM to discover a foe VM on its machine by analyzing PROBE results for evidence of the foe. In this section, we develop a classifier for PRIME-PROBE readings that yields a classification of "foe present" or of "foe absent". In Section 4.2.1, we consider the effectiveness of a simple classifier for a single PRIME-PROBE reading. Based on that experience, we design a multi-probe classifier in Section 4.2.2 and discuss training this multi-probe classifier in Section 4.2.3. We perform a cursory evaluation of the classifier's detection capability in Section 4.2.4; this evaluation will be augmented with additional evaluations in Section 4.4.

Much of our discussion in this section is informed by experiences with the platform on which we performed the experiments reported in this chapter. This platform is a 3GHz Intel Core 2 Quad computer (without SMT) with 8GB of physical memory and two L2 caches, each serving two cores. Each L2 cache is 24-way set-associative ($w = 24$) with $m = 4096$ cache sets and a line size of

$b = 64\text{B}$, yielding a cache size of $c = 6\text{MB}$. The virtualization technology is Xen. Unless otherwise specified, VMs use Ubuntu 10.04 as their guest OS. We will often motivate our design decisions based on our experiences on this platform, but we see no reason that our framework should not extend to several other platforms, as well.

4.2.1 A Single-Probe Classifier

In this section we consider a simple classifier for a single PRIME-PROBE trial. This classifier works by averaging the PROBE timings observed in the trial (i.e., averaging over the cache sets utilized) and comparing this average to a threshold. If the average PROBE timing is less than the threshold, then this implies low baseline activity in the system, and thus results in a foe-absent classification. Otherwise, the PROBE timing implies high activity, and the classification returned is foe-present.

A factor that impacts the detection accuracy is the fraction of the cache examined in a PRIME-PROBE trial. It should be more accurate to PROBE the entire cache to detect a foe VM, but it is more desirable to utilize only a portion of the cache, so friendly VMs can utilize the remainder of the cache and, in particular, can continue execution during PRIME-PROBE trials. (An implementation for achieving this property is described in Section 4.3.) Thus, in this section we evaluate our classifier when using PROBE results from only a portion (specifically, $1/16^{th}$) of the cache that friendly VMs avoid.

The successful detection probability is also a function of the foe VM activity. To allow us to examine our classifier under a range of foe VM cache activity levels, we developed a toy application inducing a random memory access pattern with a frequency that we can tune. This toy application allocates a buffer of size much larger than the cache size and then periodically selects a random location in the buffer to read. The frequency of reads can be tuned to adjust its memory access frequency. We utilize this toy application to simulate a range of foe activities.

On multi-core cloud computing platforms, VMs are usually allowed to run simultaneously and their virtual cores to migrate among physical cores. Moreover, these physical cores may or may not share a cache, depending on the hardware architecture of the host. As a first step toward evaluating our classifier, though, we consider a simplified situation to test the potential of foe detection using a single PRIME-PROBE trial. In this simplified setting, the foe VM was pinned on one of the cores

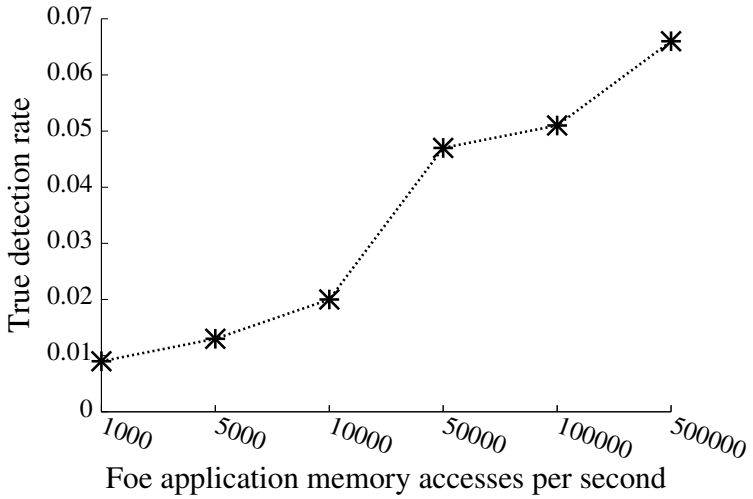


Figure 4.1: True detection rate of the single-probe classifier when false detection rate is configured to $\alpha = 1\%$. Four friendly VMs and one foe VM. All VMs unpinned. (Section 4.2.1)

that shares a common cache with another core where the monitoring VM was pinned. `Dom0` and other VMs were pinned away from the shared cache so that the cache activity of the foe VM could be sensed by the monitoring VM without interference.

In this simplified scenario, we measured the true detection rate of our classifier as a function of the memory access rate of the foe. In these tests, the detection threshold was set to allow a false detection rate of 1%, i.e., we set this threshold to be the 99th percentile of results from 1000 PRIME-PROBE trials without foes present. The PRIME-PROBE interval was 30ms, and each true detection rate was computed using 1000 PRIME-PROBE trials (with a foe present). A high true detection rate (100%) was achieved even when the memory access rate of the foe was as low as 1000 per second.

While the results of these experiments are encouraging, they unfortunately did not persist when the VMs were unpinned and allowed to move from one physical core to another, which is typical in modern cloud computing environments. Figure 4.1 shows that when all VMs were unpinned, this classifier was not nearly as effective in detecting foe VMs. In this experiment, four friendly VMs were run on a shared platform, one of which was an `apache2` web server and three of which were unloaded; each was given 1GB of memory. The web server was driven by traffic generated by `httperf` from clients external to our "cloud." The `apache2` server was subjected to a workload consisting of requests for a 1MB file at a rate sampled uniformly at random between 1 and 64

requests per second, and re-sampled every 5 seconds. The monitoring VM (one of these four VMs) attempted to detect the foe using the PRIME-PROBE protocol, using $1/16^{th}$ of the cache. Again, the toy application ran as the foe VM. As shown in Figure 4.1, the maximum true detection rate achieved was roughly only 6.5%.

The reasons behind this low true detection rate are twofold. First, the Xen scheduler balances the workload via core migration and, in doing so, varies the view of the monitoring VM. Second, because there was significant I/O activity in these tests, when the monitoring VM and `Dom0` shared a cache, there was significant cache noise induced by `Dom0` due to this friendly I/O. That is, the friendly I/O activity increased `Dom0` activity, making it appear similar to that of a foe when it shared a cache with the monitoring VM. The amount of noise in cache timings introduced by `Dom0` dynamically changes according to the I/O workload to/from VMs. While we are able to modify friendly guest operating systems at will (see Section 4.3), it appears to be impossible to control the cache activity of `Dom0` from within a VM.

4.2.2 A Multi-Probe Classifier

In light of the difficulties in interpreting the results of a single PRIME-PROBE trial discussed in Section 4.2.1, in this section we design a classifier that works using n trials for $n > 1$. For simplicity, we first describe our classifier assuming that friendly-VM activities (mainly I/O activity), as well as the number of friendly VMs, are constant and known *a priori*, and then we relax these assumptions to present a general solution.

Constant friendly-VM activity. Assuming a constant and known number of friendly VMs and level of friendly-VM activity, a PRIME-PROBE trial yields a result—namely, the PROBE time, averaged over all cache sets probed. Recall that the result of the timing measurement should be largely (though, as we will discuss in Section 4.3, not completely) independent of friendly VM memory activity, since friendly VMs will have been instructed to avoid the parts of their caches on which the monitoring VM conducts its PRIME-PROBE trials.

As a first step towards our goal of detecting a foe VM based on the results of PRIME-PROBE trials, we plot in Figure 4.2 the distribution of timing results for 2000 PRIME-PROBE trials when no foe VM is present. The trial results exhibit a bimodal distribution, which is roughly a mixture of two

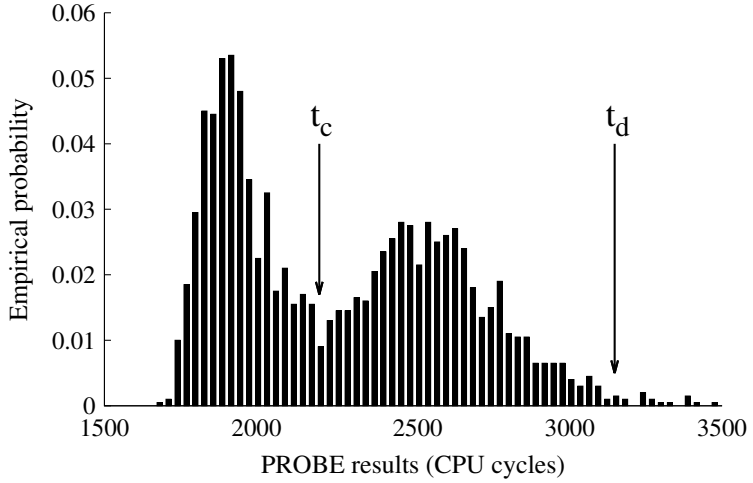


Figure 4.2: Distribution of timing results of PRIME-PROBE trials with no foe present

normal distributions. The first normal distribution characterizes the (low) level of activity that the monitoring VM registers when running on a different cache than `Dom0` during the PRIME-PROBE trial. The second normal distribution characterizes the (moderate) level of activity observed when the monitoring VM shares a cache with `Dom0`. Based on these findings, we define two overlapping classes for trial results. The first includes the majority of readings from the first normal distribution, and the second consists of the large majority of readings from both normal distributions. This design is motivated by the observation that the presence of a foe VM will tend to decrease the number of readings that fall into either class, as will be described below.

To determine if a given PRIME-PROBE trial result r belongs to one of these two classes, we empirically determine two thresholds t_c and t_d, where $t_c < t_d$, such that the cache timing measurements from the first class fall into $[0, t_c]$ and those from the second class are in the range $[0, t_d]$ (Figure 4.2). We next experimentally measure the empirical probability with which r actually falls into each of the two classes over many PRIME-PROBE trials for the friendly workload (assumed constant for the present discussion). Assuming independent trials—we revisit this assumption below—we let π_c denote the empirical probability with which r falls into the first class and π_d be the empirical probability with which it falls into the second class. Given n independent trials, we expect r to land in the first class $\overline{c} = \pi_c n$ times, and the second class $\overline{d} = \pi_d n$ times.

Let us then consider an actual execution of our detection algorithm: A sequence of n monitoring trials that aim to determine whether a foe is present. Let c denote the number of times that r actually

belongs to the first class, and d denote the number of times it belongs to the second class. In any of several ways, the presence of a foe VM might cause c to deviate from its expected value $\overline{c}$ (and d from $\overline{d}$). The foe VM could induce what would be a rare event under a friendly workload, namely a measurement r that lands above t_d and so outside of the second class. Alternatively, even if the foe VM induces only moderate cache activity (i.e., similar to that of `Dom0`), the presence of the foe could perturb the scheduling of VMs so as to decrease the odds that the monitoring VM observes a quiet cache, either by pushing the monitoring VM onto the same cache as `Dom0` or by registering cache activity itself, causing lower c than expected.

Our detection strategy, then, is to presume a foe's presence if either c or d is substantially lower than expected. More precisely, we treat a sequence of n PRIME-PROBE trials as independent events. We choose α as a desired upper bound on the rate of false detection of a foe VM. In each trial, we can view the hit/miss of the first class (i.e., $r \in [0, t_c]$ or $r \notin [0, t_c]$) as a Bernoulli trial with $\Pr[r \in [0, t_c]] = \pi_c$. It is then straightforward to compute the maximum threshold $T_c < \overline{c}$ such that $\Pr[c < T_c] \leq \alpha/2$ when no foe is present. We can similarly compute $T_d < \overline{d}$ such that $\Pr[d < T_d] \leq \alpha/2$. Summarizing, then, our basic strategy is to suspect a foe's presence if in n PRIME-PROBE trials, either $c < T_c$ or $d < T_d$. The probability of false detection of a foe over this combined test is at most $\sim \alpha$.

Arbitrary friendly-VM activity. The preceding description assumed that during the n PRIME-PROBE trials, the number of friendly VMs and the I/O activity levels of those friendly VMs were constant. In practice, this will generally not be the case, since for realistic values of n (e.g., $n \approx 25$) and for realistic times to conduct n trials (in particular, with delays between them, as will be discussed below), the total time that will elapse during the n trials would be more than long enough to witness potentially large swings in load due to fluctuations in inbound requests, for example.

As such, in practice it is necessary to compute the thresholds t_c and t_d *per trial* as a function of the set F of activity profiles of the friendly VMs during that trial. That is, F includes a profile for each of the friendly VMs' activities during the PRIME-PROBE trial. Each VM's entry could include its level of I/O and amount of computation, for example. We give details of what we included in F at the end of Section 4.2.3.

The monitoring VM collects this information F after each PRIME-PROBE trial. (We will discuss how in Section 4.3.) It uses this information to select t_c^F and t_d^F, and then evaluates whether the trial result r satisfies $r \in [0, t_c^F]$ (in which case it increments c) and whether it satisfies $r \in [0, t_d^F]$ (in which case it increments d).

Besides adjusting t_c^F and t_d^F as a function of F, we have found it helpful to adjust π_c and π_d as a function of F, as well. So, henceforth we denote them π_c^F and π_d^F. Specifically, we take π_c^F to be the fraction of training trials (see Section 4.2.3) with friendly-VM activity as described by F in which $r \in [0, t_c^F]$, and π_d^F to be the fraction of training trials with friendly-VM activity as described by F in which $r \in [0, t_d^F]$.

For n detection trials, we denote the profile characterizing the activity of the i^{th} trial by F_i. Define binary indicator random variables

$$\gamma_i = \begin{cases} 1 & \text{if } r_i \in [0, t_c^{F_i}] \\ 0 & \text{otherwise} \end{cases} \quad \text{and } \delta_i = \begin{cases} 1 & \text{if } r_i \in [0, t_d^{F_i}] \\ 0 & \text{otherwise} \end{cases}$$

where r_i denotes the result of a testing trial with friendly-VM activity characterized by F_i. We treat the observations $\gamma_1 \ldots \gamma_n$ and $\delta_1 \ldots \delta_n$ as Poisson trials. Training data suggest that under the foe-absent hypothesis, $\Pr[\gamma_i = 1] = \pi_c^{F_i}$ and $\Pr[\delta_i = 1] = \pi_d^{F_i}$. Under this hypothesis, we can then calculate probability distributions for $c = \sum_{i=1}^n \gamma_i$ and $d = \sum_{i=1}^n \delta_i$, e.g., (Chen and Liu, 1997), and maximum thresholds T_c and T_d such that $\Pr[c < T_c] \leq \alpha/2$ and $\Pr[d < T_d] \leq \alpha/2$ for a chosen false detection rate of α. As such, by detecting a foe if $c < T_c$ or $d < T_d$ we should achieve a false detection rate of at most roughly α.

We reiterate that the thresholds T_c and T_d are computed during testing as a function of $F_1, \ldots, F_n$, using values $\pi_c^{F_i}$ and $\pi_d^{F_i}$ obtained from training (see Section 4.2.3). The thresholds t_c^F and t_d^F are similarly determined using training, but which ones are used during testing is determined by the profile sets $F_1, \ldots, F_n$ actually observed.

On independence. The test outlined above requires trials that are independent, in the sense that the probability of the trial result r satisfying $r \in [0, t_c^F]$ or $r \in [0, t_d^F]$ is a function only of F and foe activities (if any), and is otherwise independent of the results of preceding trials. Achieving this independence is not straightforward, however. In practice, an effective scheduler does not migrate

virtual cores across physical cores randomly. In fact, in our experience, if the number of virtual cores is fewer than the number of physical cores, then Xen will not migrate virtual cores at all. This behavior clearly can impact achieving independent trials—in this example, if the monitoring VM is the same VM each time and if `Dom0` does not share a cache with this monitoring VM, then it never will.

For this reason, in our detector we take steps to make trials as independent as possible. Most importantly, we assign the monitoring VM randomly for each PRIME-PROBE trial from among the available friendly VMs. In addition, we employ random delays between trials to increase the likelihood that two trials encounter different virtual-to-physical core mappings (provided that friendly VMs include enough virtual cores to induce changes to these mappings). As we will show below, we believe that these steps increase the independence between trials sufficiently to construct an effective foe detector.

4.2.3 Training the Multi-Probe Classifier

The need to determine t_c^F and t_d^F as a function of F introduces a training requirement for our classifier. In this chapter we presume it is possible to train on a hardware platform that is similar, in terms of numbers of cores and caches, the arrangement of caches to cores, cache sizes, etc., to that on which the friendly VMs will eventually be run, and that this hardware platform can be equipped with the same virtualization substrate (i.e., Xen for the purposes of our discussion here) for training purposes. Of course, one way to accomplish this is to train on the cloud machines themselves, trusting that the interval in which training occurs is absent of any foes—a well-known "trust on first use" (TOFU) approach that is (unfortunately) common today in intrusion detection, key exchange, and many other contexts. A safer approach would be to replicate a machine from the cloud and use it for training, though this may require cooperation from the cloud provider.

While precisely determining t_c^F requires ground truth as to the cores (and thus caches) that `Dom0` utilized during a PRIME-PROBE trial, such ground truth would typically not be available if training were done using the first (TOFU) approach described above. To leave room for both possibilities, we employ a training regimen that does not rely on such knowledge. Specifically, we collect PRIME-PROBE trial results for fixed F in the (assumed or enforced) absence of a foe and then model these results using a mixture of two normal distributions. Intuitively, one normal distribution

should capture readings when Dom0 is absent from the cache observed by the monitoring VM, and one normal distribution should represent readings when Dom0 is present. As such, we compute a best fit of the training trial results to a Gaussian mixture model of two normal distributions, and call one normal distribution (with the smaller mean) the *quiet distribution* and the other the *like-Dom0 distribution* for F. We then use these two distributions to generate values for t_c^F and t_d^F. Specifically, we choose t_c^F to be the mean plus the standard deviation of the quiet distribution, and we choose t_d^F to be the the mean plus the standard deviation of the like-Dom0 distribution.

As described previously, each element of F describes the relevant activities of a distinct friendly VM during the PRIME-PROBE trial from which the result will be tested using t_c^F and t_d^F. Moreover, F includes a distinct such descriptor for each friendly VM. To train our classifier, it is necessary to incorporate training executions that match the profiles F likely to be seen in practice. The extensiveness of this data collection depends in large part on the features that are incorporated into the VM activity descriptors in F and on the granularity at which these features need to be captured. The training executions should also range over the possible number of VMs on the same computer, which we assume the party deploying the VMs can determine (c.f., (Ristenpart et al., 2009)).

While our framework permits building a detector based on a variety of features included in the friendly VM profiles, we found in our experiments that the most relevant feature to capture is the level of I/O activity in each friendly VM. As already discussed, the I/O activity of friendly VMs is highly correlated with Dom0's activity evidenced in the cache. Fortunately, capturing this information at only a coarse granularity is sufficient to build an effective detector. Specifically, in the experiments we report in this chapter, we compute the *aggregate* number of bytes of I/O activity involving friendly VMs during the PRIME-PROBE trial (as measured in sys_read and sys_write calls). We bin the total friendly-VM I/O activity during the PRIME-PROBE trials into one of 20 bins. Any two profiles F and F' falling into the same bin are treated as equivalent for our purposes.

4.2.4 Multi-Probe Detection Capability

In this section we provide a cursory evaluation to confirm that our multi-probe detector overcomes the limitations of the single-probe detector of Section 4.2.1. Our results here are not intended to be exhaustive; we will consider detection in the context of additional workloads in Section 4.4.

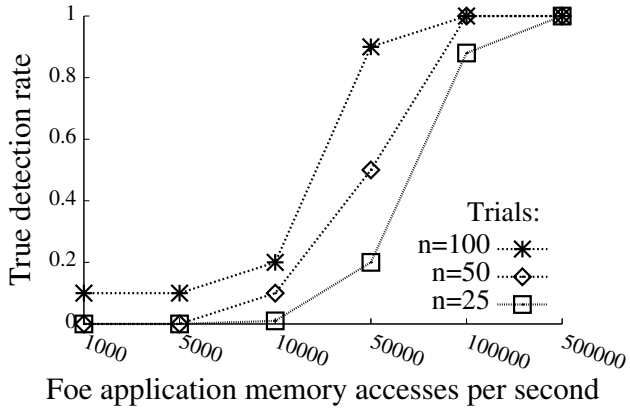


Figure 4.3: True detection rate of multi-probe detector of Section 4.2.2 with four friendly VMs ($\alpha = 1\%$).

Recall that the shortcoming of our single-probe detector was revealed when we unpinned VMs, allowing them to migrate among the available cores. We consider only this case here; all VMs are unpinned. We introduced four friendly VMs on our platform, the configurations of which were exactly the same as those in experiments for Figure 4.1, namely one `apache2` server and three unloaded VMs.

We first collected the results of 20,000 PRIME-PROBE trials employing $1/16^{th}$ of the cache with no foe present. The delay between PRIME-PROBE trials was chosen uniformly at random between 1 and 5 seconds. To confirm our ability to configure the false detection rate, we conducted a 10-fold cross-validation, in which we partitioned these 20,000 results into 10 equally sized sets and then tested on each set after training on the remainder (with $\alpha = 1\%$). Each testing set was broken into non-overlapping windows of n PRIME-PROBE trials ($n \in \{25, 50, 100\}$), each window yielding a foe or no-foe classification. The false detection rate that we observed was indeed less than 1% for each value of n.

We then added a foe, using the same toy program as in Section 4.2.1. Figure 4.3 shows the true detection rate for testing performed in the same fashion, after training on the previously collected 20,000 trials with $\alpha = 1\%$. For each value of n, the sum of all n-trial windows was 2000 PRIME-PROBE trials, meaning that the curves for smaller values of n show averages over a greater number of windows. In this figure, the memory access rate of the foe application is indicated on the x-axis. Our multi-probe classifier improves substantially over the single-probe classifier, for the same false

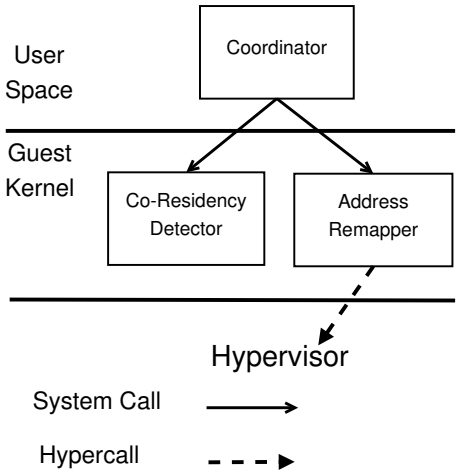


Figure 4.4: Architecture of HomeAlone within one guest VM

detection rate. The tradeoff is that our multi-probe classifier takes longer to evaluate, due to its use of multiple PRIME-PROBE trials separated by random intervals.

4.3 Implementation

In order to execute the detection technique described in Section 4.2, the cloud customer must modify the VMs that it deploys to the cloud (the "friendly VMs"). In this section we describe our proof-of-concept implementation, which we developed within 64-bit PVOps Linux kernel 2.6.32.16 for Xen. Our modifications have been tested with the Xen 4.0.1-rc2 hypervisor.

Our implementation consists of a suite of tools that is installed within each friendly VM. As shown in Figure 4.4, this suite includes a user-level *coordinator* and two kernel extensions in each guest OS kernel, namely an *address remapper* and a *co-residency detector*.

4.3.1 Coordinator

Each friendly-VM coordinator works in user space and is responsible for coordinating the detection task with coordinators in other friendly VMs residing on the same physical cloud host. We presume that coordinators can determine the friendly VMs residing on the same physical host, either because this has been configured by the cloud customer deploying these VMs or by detecting which friendly VMs do so (e.g., see (Ristenpart et al., 2009)).

A *detection period* is begun by one coordinator, here called the *initiator* for the detection period, sending commands (in our implementation, using TCP/IP) to the other friendly-VM coordinators. This command indicates a randomly selected *color* that defines the cache sets in each cache on the host that will be used during this detection period for executing PRIME-PROBE trials. A coordinator that receives this message must invoke its local address remapper (via a system call) to vacate use of those cache sets (to the extent that it can; see Section 4.3.2), so that execution of this VM will minimize pollution of those cache sets during the detection period. We note that each coordinator makes no effort to determine whether it is presently on the same cache as the initiator (if the host has multiple caches), either during initiation or at any point during the detection period. Rather, each coordinator uses its address remapper to vacate the cache sets of the color indicated by the initiator, for any cache used by its VM for the duration of the detection period. Upon receiving confirmation from its address remapper that those cache sets have been vacated, the local coordinator sends a confirmation to the initiator.

Once the initiator has received confirmation from the friendly VMs on the host, it creates a *token*, selects a friendly VM on the host uniformly at random, and passes the token to the selected VM. The selected VM now becomes the *token holder* and will act as the monitoring VM for one PRIME-PROBE trial. More specifically, the token holder alerts the other coordinators of its impending trial and then contacts its local co-residency detector to perform the PRIME-PROBE trial. Once the co-residency detector has completed the trial and returned the trial result r, the token holder collects an activity profile F from the friendly VMs. Each entry of this activity profile characterizes the I/O activity (bytes passed through `sys_read` or `sys_write`) of the friendly VM from which the entry was received, since that VM received the alert preceding the PRIME-PROBE trial. Finally, the token holder induces a random delay (to improve independence of trials; see Section 4.2.2) and then selects the next token holder uniformly at random (again, for independence) from the friendly VMs on the host. When passing the token to the new token holder, the sender includes all trial results r and corresponding activity profiles F collected in this detection period so far.

After n trials have been performed, the new token holder can evaluate the results and activity profiles to determine whether to declare that a foe is present on the machine, using the technique described in Section 4.2.

4.3.2 Address Remapper

The address remapper is provided a color, which defines cache sets that need to be avoided due to their planned use in the pending detection period. To avoid the use of these cache sets, the address remapper colors each physical memory page (c.f., (Lynch et al., 1992; Bugnion et al., 1996)) by the (unique, in our implementation) color of the cache sets to which its contents are mapped, and then causes its VM to avoid touching cache sets of the designated color by causing it to avoid accessing physical memory pages of the same color.

A straightforward way of causing its VM to avoid these memory pages would be to to alter the view of memory that the guest OS perceives. For instance, we can "unplug" the memory pages that need to be avoided, by indicating that such pages are unusable in the page descriptor structure in the guest OS. A drawback of this approach is that it breaks up physical memory as perceived by the OS, so that the OS no longer has access to a large, contiguous memory space. For example, the buddy memory allocator used in Linux maintains an array of lists, the j-th entry of which collects a list of free memory blocks of size 2^j pages, where $j = 0, 1, \ldots, 10$. Therefore, "unplugging" memory pages of one color will result in empty lists for $j \geq 6$ in the case of 64 page colors, since a block of $2^6 = 64$ pages (or larger) will contain one page of each color. Others have cautioned against this in other contexts, due to serious performance issues that it may cause (Chisnall, 2007).

Instead, we take advantage of the additional indirection layer in the mapping from virtual to physical memory introduced by virtualization. The Xen hypervisor provides a *pseudo-physical* address space to each guest virtual machine and maintains the mapping from pseudo-physical to physical memory. Because physical memory is allocated at page granularity in Xen, the memory allocated to each VM is not guaranteed to be actually contiguous, but the contiguous pseudo-physical address space in each guest virtual machine provides the illusion to the guest OS that it is running on an intact physical memory. In paravirtualized virtual machines, whereas the pseudo-physical address space is the one that is used across the operating system, the guest OS is also aware of the corresponding machine address of each page, which is embedded in the page table entry for the hardware MMU to look up during translation (i.e., translation is done directly from guest virtual address to real machine address). This design leaves us an opportunity to modify the pseudo-physical-to-machine-address mapping to avoid touching certain physical pages while keeping the

guest OS' view of memory layout unchanged. In particular, to remap the machine address of a single pseudo-physical page, the address remapper issues a hypercall to the hypervisor indicating the new machine address and then modifies the guest OS' copy of this mapping. So as to prevent accesses to these mappings while they are being reconfigured, the address remapper disables interrupts and preemption of its virtual core and suspends its guest OS' other virtual cores (if any) prior performing the remapping.

In the process of address remapping to avoid using physical pages of the specified color, the address remapper needs to copy page contents out of pages that need to be avoided and then update page tables accordingly. To provide a destination for these copies, a pool of memory pages is reserved when the guest OS is booted. This pool should be large enough to hold an entire color of memory. During the remapping process, the address remapper copies each physical page of the specified color to a page in the reserved memory pool of a different color, and then updates the page tables accordingly by issuing hypercalls to the hypervisor. One caveat is that if a page of the specified color corresponds to a page table or page directory that is write protected by the hypervisor, then this page cannot be exchanged and has to be left alone. These pages, and a few other pages that cannot be moved, are the primary cause of the remaining cache noise in our PRIME-PROBE trials.

To summarize, the remapper performs the following steps. It enumerates the pseudo-physical pages that are visible from the guest OS. For each page P, the machine address of the page is determined to figure out whether it is the designated color. If so, in which case this page would ideally be remapped, the remapper examines the page table entries pointing to P and also the page descriptor structure. In several cases—e.g., if P is reserved by HomeAlone, write-protected by the hypervisor, or a kernel stack page currently in use—then the remapper must leave the page alone. Otherwise, the remapper identifies a new page (of a different color) from its pool and exchanges the machine address of P with that of this new page (via a hypercall). Prior to doing so, it copies P's content to this new page if P was in use. The remapper updates the kernel page table (also by hypercall) and, if P was used in user space, then the remapper updates the user-space page tables. The performance of this algorithm will be evaluated in Section 4.4.

This implementation constrains the number of colors in our scheme and thus the granularity at which we can select cache sets to avoid. Let w denote the way-associativity of the cache; m be the number of cache sets; c be the size of the cache in bytes; b be the size of a cache line in bytes;

p be the size of a page in bytes; and k denote the maximum number of page colors. Each b-sized block of a page can be stored in a distinct cache set, and avoiding a particular cache set implies avoiding every page that includes a block that it could be asked to cache. Since the p/b blocks of the page with index i map to cache set indices $\{i(p/b) \bmod m, \ldots, (i+1)(p/b) - 1 \bmod m\}$, the most granular way of coloring cache sets is to have one color correspond to cache sets with indices in $\{i(p/b) \bmod m, \ldots, (i+1)(p/b) - 1 \bmod m\}$ for a given $i \in \{0, \ldots, \frac{m}{p/b} - 1\}$. Since $m = c/(w \times b)$, the number k of colors that our implementation can support is

$$k = \frac{c/(w \times b)}{p/b} = \frac{c}{w \times p}$$

On our experimental platform, an Intel Core 2 Quad processor, the L2 cache is characterized by $c = 6\text{MB}$, $w = 24$, and $b = 64\text{B}$, and Linux page size is $p = 4\text{KB}$. Thus the number of page colors in our system is $k = 64$.

4.3.3 Co-Residency Detector

The co-residency detector, which is implemented as a Linux kernel extension, executes the PRIME-PROBE protocol for measuring L2 cache activity. To PRIME the cache sets to be used in the PRIME-PROBE trial (i.e., of the color specified by the coordinator), the co-residency detector must request data from pages that map to those cache sets. To do so, at initialization the co-residency detector allocates physical pages sufficient to ensure that it can PRIME any cache set.

When invoked by the coordinator, the co-residency detector PRIMEs the cache sets of the specified color, and then waits for the PRIME-PROBE interval. In our experiments, this interval is configured empirically to be long enough for a reasonably active foe to divulge its presence in the cache but not so long that core migration of the monitoring VM becomes likely. In our experiments we use a PRIME-PROBE interval of 30ms.

The co-residency detector is tuned to improve its detection ability in several ways. First, on our experimental platform, every cache miss causes one line to be filled with the requested content and another to be filled through prefetching; i.e., a cache miss fills two cache lines in consecutive cache sets. As such, our co-residency detector PROBEs only every other cache set. Second, to eliminate noise due to the TLB, the co-residency detector flushes the TLB before its PROBE of each cache

set, so as to ensure a TLB miss. Third, the co-residency detector disables interrupts and preemption during the PRIME-PROBE protocol to limit activity that might disrupt its detection.

4.4 Evaluation

In this section, we deploy HomeAlone on a small private cloud in which four friendly VMs are running on one physical host virtualized with Xen. The host is the same as that employed in the experiments of Section 4.2.

The applications that we employ in our VMs are taken from the PARSEC benchmarks (Bienia et al., 2008; Bienia and Li, 2009). PARSEC is distinguished from most other suites in focusing on multithreaded benchmarks representative of diverse, emerging workloads, and so we take it as representative of future cloud computing workloads. In particular, we utilized the following benchmarks from PARSEC.

- `blackscholes`: This benchmark simulates financial analysis and, in particular, calculates the prices of a portfolio of options using Black-Scholes partial differential equations.
- `bodytrack`: This computer vision application tracks a 3D pose of human bodies and represents video surveillance and character animation applications.
- `canneal`: This is a benchmark using cache-aware simulated annealing to design chips that minimize routing costs; it is representative of engineering applications.
- `dedup`: This benchmark is short for "deduplication", which is a compression approach that combines global and local compression in order to obtain a high compression ratio; it is used to simulate next-generation backup storage systems.
- `facesim`: This benchmark simulates human faces and is representative of applications like computer games that employ physical simulation to create virtual environments.
- `streamcluster`: This benchmark was developed for solving online clustering problems and is included for its representation of data mining algorithms.
- `x264`: This is an H.264/AVC video encoder that can be used to simulate next-generation video systems.

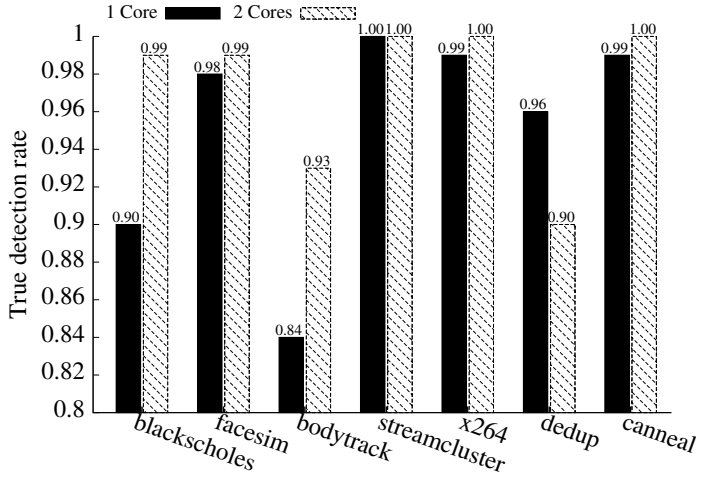


Figure 4.5: True detection rates for different foe applications ($n = 25$, $\alpha = 1\%$)

Each benchmark was provided the "`native`" input designated for the benchmark. In addition to these PARSEC benchmark applications, in some tests we employed an `apache2` web server on which we induced a workload as described in Section 4.2.1.

4.4.1 Detection

To test the effectiveness of our co-residency detector, we trained our classifier on a workload that included four friendly VMs, one running `apache2`, one running `facesim`, one running `streamcluster`, and one running `blackscholes`. Each VM was given one 1GB of memory and one virtual core. We do not claim that this request profile, or that this mix of applications, is representative of any particular cloud tenant workload. We simply used this mix of applications to capture a broad range of reasonably intensive activities.

Training consisted of collecting results from 20,000 PRIME-PROBE trials on $1/16^{th}$ of the cache, each pair separated by an interval chosen independently and uniformly from between 1 and 5 seconds. Training was performed as prescribed in Section 4.2.3 and tuned to a false detection rate of $\alpha = 1\%$. We confirmed this false detection rate using a 10-fold cross validation as in Section 4.2.4 with $n = 25$.

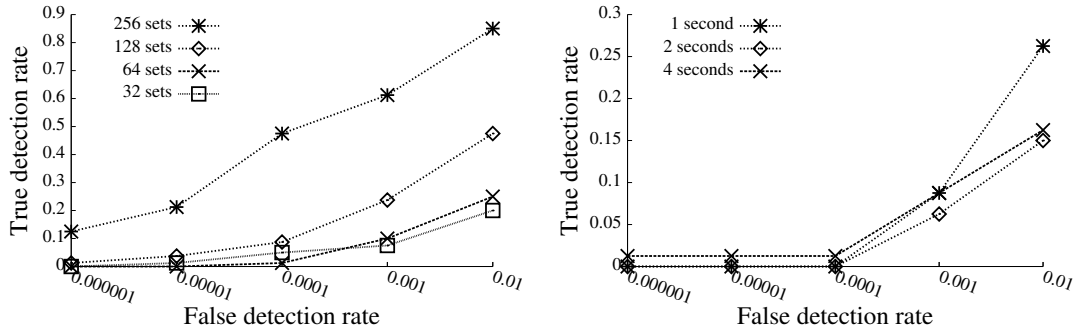
Detecting benign foe VMs. After training, we conducted seven runs with the same friendly workload and one foe VM. In each of the seven runs, the foe executed one of the seven PARSEC benchmark applications. Each run yielded 2000 PRIME-PROBE trials on $1/16^{th}$ of the cache, each

pair again separated by a random interval between 1 and 5 seconds. Nonoverlapping subsequences of $n = 25$ trials were then classified using our detector. The true detection rates observed are shown in Figure 4.5. As shown there, for the single-core foe VMs, true detection rates ranged from roughly 84% (`bodytrack`) to 100% (`streamcluster`). Except for `dedup`, true detection rates improved slightly when the foe VM employed two cores. The improvement of detection rates during the foe VM's use of multiple cores is possibly due to increased contention for the physical CPU resources. We believe that the variation in true detection rates across foe applications is caused by the different features of these applications, e.g., their CPU usage patterns and I/O intensities. Future research may help determine the relationship between detection rates and application properties.

An interesting limiting case for detecting benign foe VMs is a foe VM that runs nothing more than a guest OS. We briefly experimented with the possibility of detecting such a foe VM. In particular, we ran HomeAlone against an "idle" Linux foe VM (Ubuntu 10.04) and an "idle" Windows 7 foe VM, i.e., VMs with no actively running applications. HomeAlone proved effective even in this challenging setting: It achieved almost a 15% true detection rate against the Linux foe, and a 70% true detection rate for the Windows foe. (In both cases, $\alpha = 1\%$ and $n = 25$.) While further experimentation is warranted, these preliminary results perhaps provide rough lower bounds for the true detection rates of benign foe VMs of these types.

Detecting adversarial foe VMs. We further evaluated HomeAlone by studying its effectiveness against *adversarial* foe VMs, as described in section 4.1.1. The adversarial foe VMs we considered actively attempted to exfiltrate data from friendly VMs by themselves running the PRIME-PROBE protocol on portions of the L2 cache. Furthermore, the adversary's targeted collection of cache sets was fixed, as we expect an adversary would generally need to target the same cache sets for a substantial duration to exfiltrate meaningful information from friendly VMs.

The detection accuracy of HomeAlone depends on how frequently the foe VM executes PRIME-PROBE trials and on the number of cache sets in the intersection between the regions probed by HomeAlone and by the foe VM. Figure 4.6 shows the true and false detection rates over a range of adversarial foe VM PRIME-PROBE frequencies and amounts of cache-set overlap. The experimental parameters used for detection (e.g., n, α, PRIME-PROBE interval, total number and time between PRIME-PROBE trials by HomeAlone) were selected as in our detection experiments above.



(a) True detection rates for an adversarial foe VM executing continuous PRIME-PROBE cycles. Each curve represents a different number of cache sets overlapping with HomeAlone.

(b) True detection rates for an adversarial foe VM, where the number of cache sets overlapping with HomeAlone is fixed at 256. Each curve corresponds to a different delay between adversarial PRIME-PROBE trials.

Figure 4.6: ROC curve for detecting adversarial foe VMs with different aggressiveness ($n = 25$, $\alpha = 1\%$).

As illustrated in Figure 4.6(a), for an adversary that performs PRIME-PROBE protocols back-to-back with a minimal intervening delay, detection accuracy improved as the overlapping cache region grew. With as few as 32 overlapping cache sets, HomeAlone achieved a 20% true detection rate with a false detection rate of 1%. When the full $1/16^{th}$ of the cache monitored by HomeAlone overlapped with the foe VM's region of activity, the true detection rate rose to 85%. As seen in Figure 4.6(b), the true detection rate of HomeAlone increased, as expected, with the foe VM's PRIME-PROBE frequency. Such detection is possible, however, only when the foe VM executes PRIME-PROBE protocols with sufficient frequency and scope. A sufficiently inactive foe VM, i.e., one probing a small portion of the cache (e.g., 32 cache sets) with low frequency (e.g., every 10 seconds) will likely escape detection. The bandwidth of the resulting side-channel, though, would render meaningful data exfiltration challenging.

Responding to detections. When co-residency is detected by HomeAlone, the customer whose friendly VMs are at risk has several options available to respond. If the customer is not immediately concerned about attacks on friendly VMs (e.g., if the customer employs HomeAlone primarily to detect service-provider misconfigurations as opposed to truly hostile foe VMs), the customer might simply attempt to confirm the detection to a higher degree of assurance. For example, the friendly VMs could increase the portion of the cache they use for detection, increase n, or leverage multiple n-sized tests as described below. If this additional testing confirms the presence of foe VMs, then the

customer should presumably report this problem to the cloud provider. If some of the customer's VMs contain highly sensitive data that warrant more immediate reaction to a detection, then the customer might suspend processing of that data while the aforementioned steps are performed to confirm the detection.

Probability amplification. In most of our tests, the separation of the true detection rate from the false detection rate (of $\leq 1\%$) was substantial. This separation can be leveraged to substantially improve HomeAlone's sensitivity—both its true detection rate and its false detection rate—using the known technique of *probability amplification*. In this approach, a series of N detection periods (each of n trials) is executed, each yielding a binary detection hypothesis ("foe present" / "foe absent"). A *meta-classifier* is applied to these N outputs. The output of the meta-classifier ("Foe Present" / "Foe Absent") is based on the fraction of "foe present" results across runs, according to a statistical test that we briefly describe.

Let α denote the false detection rate for a run of HomeAlone and β the true detection rate (with the requirement that $\alpha < \beta$). Let z denote the number of "foe present" outputs over the N runs; $\mathbb{E}[z] = \alpha N$ with no foe present, while $\mathbb{E}[z] = \beta N$ with a foe truly present. The meta-classifier then outputs "Foe Present" if $z \geq (\alpha + \beta)N/2$, i.e., z exceeds a threshold defined as the mid-point between expected values under the two hypotheses; it outputs "Foe Absent" otherwise. (The threshold can be adjusted, of course.)

Assuming that the outputs of individual HomeAlone runs are statistically independent (even partially independent), this meta-classifier can achieve very high detection rates and very low false detection rates for moderate values of N. For example, assuming complete independence, a single-run true detection rate of $\beta = 84\%$ (the lowest we observed for the PARSEC benchmarks) and a false detection rate of $\alpha = 1\%$, with $N = 10$, the meta-classifier detection rate would be $> 99.8\%$, with a false detection rate $< 2.5 \times 10^{-8}$. When HomeAlone is used in particular to detect cloud configuration errors (and thus a long-persisting foe), it is feasible to support many more detection periods.

The degree of independence between runs increases with the length of time between them. Run-independence can also be reinforced, we expect, with a resampling of the cache color monitored by HomeAlone. Further research would be required to characterize the statistical dependence between

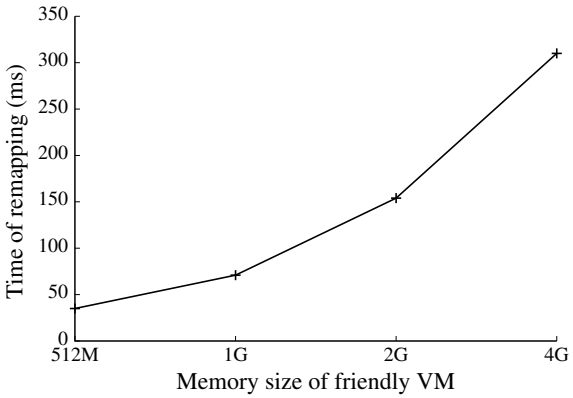


Figure 4.7: Runtime of remapping $1/16^{th}$ of memory for one VM, as function of memory size. Note that x-axis is log-scale.

runs and to determine the most appropriate tradeoff between execution time and sensitivity for probability amplification.

4.4.2 Performance

In this section, we examine the overhead induced by HomeAlone when avoiding $1/16^{th}$ of the cache.

Overhead of address remapping. At the beginning of a detection period, HomeAlone can change the region of the cache being avoided by friendly VMs by transmitting a randomly chosen cache color to all friendly VMs. This mechanism is useful to conceal the monitored region from an active foe that tries to escape detection. (Such a foe is discussed more in Section 4.5.) However, changing the cache color induces performance overhead caused by the address remapping procedure (see Section 4.3.2). In Figure 4.7, we show the overhead of address remapping in our (unoptimized) implementation, as a function of the total memory size, assuming 16 colors (and so each color constitutes $1/16^{th}$ of the memory).

In our implementation, applications running on friendly VMs are paused during remapping. So, the costs shown in Figure 4.7 are not inconsequential to applications. That said, a more refined implementation could perform remapping incrementally (e.g., one or a few pages at a time), permitting applications to run between remapping increments. As such, remapping need not incur a large contiguous pause in activity, but rather the remapping costs can be amortized over a longer interval and interleaved with application execution.

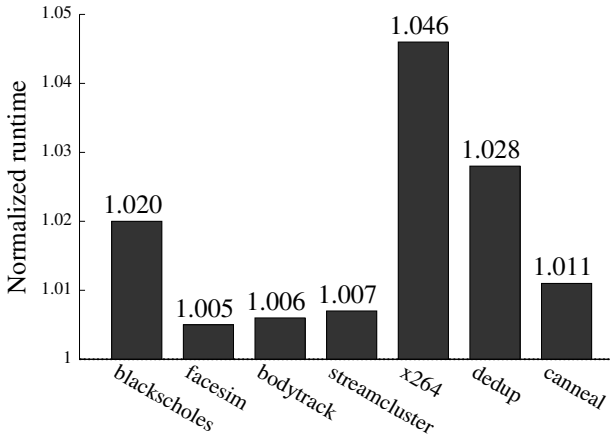


Figure 4.8: Normalized performance of benchmark applications during detection periods

Overhead during detection periods. During detection periods, applications inside friendly VMs continue to run, but the VMs do not utilize the entire cache. In addition, a detection thread runs inside the monitoring VM, and the coordinators of the VMs interact to perform PRIME-PROBE trials (see Section 4.3). In this section we show the overhead that this induced on the seven PARSEC benchmark applications during detection periods.

To measure these costs, we first ran each benchmark 10 times without HomeAlone; in each of these runs, the benchmark ran alone on the platform but within a VM with one virtual core.[3] In 10 subsequent tests, we ran the benchmark in one VM (with HomeAlone) that participated with three other, unloaded VMs in our foe detection protocol. Notably, this involved avoiding $1/16^{th}$ of its cache, conducting PRIME-PROBE trials at random intervals chosen between 1 and 5 seconds, and coordinating detection across these VMs. (These tests did not include remapping. As discussed above, this happens before the detection period and the costs can be amortized over an arbitrary amount of time in advance.) We then computed a normalized runtime of each benchmark when run with HomeAlone enabled, by dividing the average runtime of the benchmark when run with HomeAlone by the average runtime of the benchmark when run without it.

The results, shown in Figure 4.8, suggest that there is modest performance degradation in the benchmarks we examined. The benchmark that suffered the most, namely `x264`, did so by approximately 4.6% on average. We believe the reasons for the modest overhead of HomeAlone

[3]All benchmarks were run in a virtual machine with 1GB of memory, except for `dedup` and `canneal`, which were given 3GB of memory to avoid frequent swapping.

are multi-fold: First, in rare instances do applications utilize the entire cache, and thus avoiding $1/16^{th}$ of the cache impacts the performance of most applications minimally. Second, we conjecture that due to artifacts of virtualization, avoiding a portion of the cache is less disruptive to application performance than it would be in a traditional environment.

4.5 Discussion

In this section we briefly consider several issues that may affect how our techniques are applied in practice.

Machine migration during detection. Our experiments assumed that the number of friendly VMs is constant during detection periods. The unexpected machine migration of a friendly VM to or from the host, or the instantiation of a new friendly VM on the host, could potentially produce a false detection. If this is a possibility, then additional measures will be needed to report these events and, if necessary, disregard any detections based on observations with which these events may have interfered. As discussed in Section 4.3.1, our techniques already assume the ability to coordinate across friendly VMs on the same host. These additional measures to address changes in the population of friendly VMs are simply an extension of that requirement.

Hardware-assisted virtualization. With HVM technology, the hypervisor can utilize hardware assistance to better isolate one guest OS from another (Neiger et al., 2006; Dong et al., 2006). Although major computing infrastructure providers like Amazon and Rackspace still support PVM guests, we expect a move to HVM in the future. Some technical differences between HVM and PVM alter the cache-based side channel for our purposes.

The most acute complication comes from virtualization of the MMU. In HVM, only pseudo-physical memory addresses are visible to guests and stored in the guest page table entries. The mapping from the pseudo-physical to the machine address space is done through a shadow page table maintained by the hypervisor (Dong et al., 2006). In HVMs, guests do not have direct control of the physical memory addresses, and this impacts our cache coloring technique used for avoiding certain cache regions during detection.

To our advantage, more and more hardware-assisted virtual machines seek paravirtualized functionality. Hypercalls, traditionally used only by PVM, are now used in HVM for better per-

formance. Examples include hypercalls that allow guests direct control of device drivers (see `http://www.flexiant.com/`), and hypercalls that make the real machine address visible to guests. We thus believe that minor modifications would make our detection techniques viable in cloud environments with HVM guests.

Evading detection. As shown, HomeAlone detects a foe VM whose activities are significantly evidenced in the L2 cache during its execution. A foe VM with knowledge of HomeAlone could try to limit its cache footprint in order to evade detection. Since HomeAlone selects a different cache region (color) in each detection period, to escape detection the foe would presumably need to lower its utilization of most or all of the cache or else discern the color being used by HomeAlone and avoid only those portions of the cache. To discern the color, however, the foe would presumably need to probe the cache, an activity that HomeAlone is designed to detect. More generally, HomeAlone is well positioned to detect side-channel attacks via the cache (e.g., of cryptographic keys), and so a foe that avoids the cache, either in whole or in part, to evade detection sacrifices a significant attack vector to do so. Of course, it can make use of other timing channels—e.g., the instruction cache (Aciiçmez, 2007; Aciiçmez et al., 2010), the branch target cache (Aciiçmez et al., 2007c,a), or shared functional units (Wang and Lee, 2006; Aciiçmez and Seifert, 2007)—but these channels require SMT, which is not supported in some clouds, and far less has been shown about the efficacy of these channels. Moreover, it may be possible to extend HomeAlone to monitor those channels as well.

4.6 Summary

With the growing movement of sensitive applications to clouds, there is increasing demand for physical isolation of tenants' workloads (e.g., (Carlson, 2010; AmazonAWS, 2010)). In this chapter we have developed an approach called HomeAlone by which a tenant of an IaaS cloud can detect if this isolation is violated, without requiring cooperation from the cloud service provider. In addition to providing the first such capability of which we are aware, our approach is novel in utilizing cache timing channels as a *defensive monitoring* technique, in contrast to the significant body of literature that uses them as an attack vector.

We detailed the design of our cache timing classifier for detecting the co-residence of "foe VMs" with a tenant's own "friendly VMs" and how we overcame significant obstacles to make this detection viable. We also implemented our detector within Linux for Xen, and demonstrated that our detector impacted performance modestly (less than 5%) in a range of benchmark applications. Foe detection tests indicate that reasonably active, benign foes can be detected in 25 PRIME-PROBE trials of $1/16^{th}$ of the cache with a true detection rate ranging from 84% up to 100%, while permitting a false detection rate of only $\sim 1\%$. For similar parameter settings, foe VMs that attempted to exploit the cache as a side-channel were detected with rates ranging from 15% to 85% in our tests, depending on the frequencies with which they probed and the extents to which the cache sets they probed overlapped those monitored by HomeAlone.

As an initial example of using side channels to monitor for co-resident foes, we believe our work opens up new directions for research, both in better classifiers for cache timing behavior and in use of other side channels. And, while we believe that avoiding detection by HomeAlone imposes significant penalties on a foe VM—namely avoiding its own cache and thus dispensing of a potent attack vector of its own—we anticipate and welcome additional progress in testing the limits of this approach.

CHAPTER 5: OS-LEVEL SIDE CHANNEL MITIGATION[1]

In the previous chapters, we have explained in details a fine-grained side-channel attack that extract cryptographic keys across the VM boundaries, and a defensive mechanism that exploits a similar cache-based side channel to verify the status of physical isolation of VMs in public clouds. In this chapter, we present another OS-level defensive approach which injects random noise into the share CPU caches to mitigate side-channel threats. We call the system proposed in this chapter *Düppel*. Specially, we present the design of Düppel in Section 5.1 and Section 5.2. We will then describe Düppel's implementation in Section 5.3 and its evaluation in Section 5.4. At last, we discuss possible extensions and limitations in Section 5.5 and summarize in Section 5.6.

5.1 Design Goals

The anticipated attack scenario is in public IaaS clouds, where the attacker has full control of a VM co-located with the Düppel-protected VM and is capable of exploiting per-core caches as side channels to exfiltrate sensitive information. We assume the underlying CPU is based on x86 architecture and is *not* equipped with SMT. Such hardware configurations dominate modern public clouds. The two VMs in question may time-share the same CPU core, thus time-sharing the L1 instruction cache, L1 data cache, the unified L2 cache (if any), BTBs, TLBs and other caching architectures in a CPU core. We further assume the attacker can obtain a copy of the software stack running in the Düppel-protected system and so can experiment with it to learn the cache behavior that it induces. The cloud provider controls the hypervisor, which operates as is—it neither facilitates the side-channel attacks nor does anything to thwart such threats.

In light of this threat model, we have the following goals for Düppel:

Goal 1. Düppel should mitigate side-channel attacks via time-shared caches.

[1]This chapter is excerpted from Zhang and Reiter (2013).

The implementation of Düppel that we develop here addresses side channels using the L1 (or, if any, L2) per-core caches, which are time-shared caches. The principles for addressing side channels in these caches should apply to addressing them in other time-shared caches, as well.

Goal 2. Düppel should not require any hypervisor modification.

An important design principle of Düppel is to permit its adoption by cloud tenants on modern cloud infrastructures without any additional support from the cloud providers.

Goal 3. Düppel should not require modifying applications or libraries.

Though many secrets that might be targeted in side-channel attacks are handled in applications and third-party linked libraries, the sheer number and diversity of applications and libraries makes modifying all of them an unattractive proposition. Instead, a goal for Düppel is to protect such secrets without modifications to these components. We therefore adopted the guest OS kernel as the (sole) location in which to implement our techniques.

Goal 4. Düppel should induce little performance overhead.

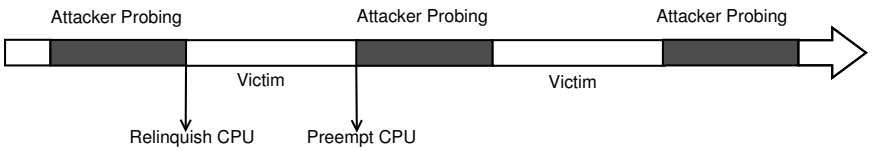
To be used in practice, Düppel must impose little performance burden on the system. This may require Düppel to operate in different modes during its life cycle.

5.2 Cleansing Time-Shared Caches

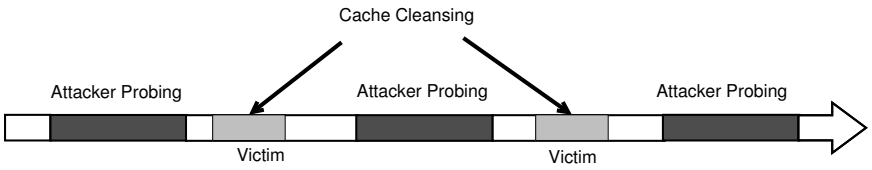
Because attacks on time-shared caches must involve cache PROBing via CPU preemption, the intuition behind Düppel is that a guest VM (and the OS that runs on it) can protect its own execution against side-channel attacks by adding noise to these caches very frequently so that side-channel readings by the attacker are confounded by noise added between PRIMEs and PROBEs.

5.2.1 Basic Design

Düppel employs periodic cache cleansing to mitigate side channels in time-shared caches, e.g., L1 data/instruction caches and unified, per-core L2 caches, if any. The principles for addressing these caches could be used to address branch predication caches, as well. The basic idea is illustrated in Figure 5.1. As described in Section 2.1.3, PRIME-PROBE protocols on time-shared caches work as illustrated in Figure 5.1(a): the attacker periodically preempts the victim's VCPU by gaming the hypervisor scheduler and then PROBEs the cache, which takes some time due to cache/memory



(a) Attackers exploits time-shared caches by preempting victim VCPUs.



(b) Periodic time-shared cache cleansing.

Figure 5.1: Attack and defense on time-shared caches

access latency. After the attacker relinquishes the physical CPU, the victim VCPU is scheduled for a short period until the attacker preempts it again.

Periodic cache cleansing, as illustrated in Figure 5.1(b), works by cleansing the time-shared cache (ideally) between the attacker's PROBEs. Specifically, cleansing involves Düppel itself PRIMing the cache in random order until all of the entries have been evicted. The cache cleansing process is accomplished by an kernel thread that is periodically invoked by Düppel.

Key to the success of this technique is a fine-grained resolution timer in the guest OS kernel that can be periodically triggered to execute cache cleansing. There are several possible timers that might be utilized for this purpose:

- **Hardware Timers:** On x86 platforms, the hardware timers include the programmable interval timer (PIT), the local advanced programmable interrupt controller timer (Local APIC), the high precision event timer (HPET), and the advanced configuration and power interface (ACPI) power management timer. Performance monitoring units (PMU) can also be regarded as a type of hardware timer, since they can be set to trigger upon hardware events that occur with predictable rates. In modern IaaS clouds, e.g., Amazon EC2 and Rackspace, most guest VMs run on paravirtualized Xen platforms. However, in such settings, none of the above hardware timers can be set to trigger by a guest VM.

- **Software Timers:** A software timer may be as complex as a Linux high-resolution timer (hrtimer) or as simple as a piece of code that busy loops and periodically invokes the execution of the timer handling code using interrupts. These interrupts can include network I/O inter-

rupts and inter-processor interrupts, the latter of which was used by Zhang et al. (2012). In paravirtualized VMs, although IPIs are also virtualized (as are I/O interrupts), they offer more stable and shorter interrupt delays. The delay of an IPI timer can be as low as 3μs, while that of the hrtimer is usually higher than 15 to 20μs.

The timers used in Düppel are hrtimers and software timers generated by IPIs. Periodic cache cleansing activities can operate in two modes: *sentinel* mode and *battle* mode. The sentinel mode is used when frequent cache cleansing is not critical, and thus the interval between two cleansings can be longer—as long as Düppel can detect abnormal activities and switch to battle mode quickly. The hrtimers are employed in sentinel mode because it induces lower performance overhead (conforming to GOAL 4). The much more rapid IPI timers are used in the battle mode, where intervals between cache cleansings are required to be as short as possible and the performance overhead is less of a concern when under active side-channel attacks.

5.2.2 Optimizations

We discuss in this section a few design optimizations that reduce the performance impact of Düppel.

5.2.2.1 Limiting the Protection Scope

Düppel's periodic cleansing of the L1 cache impacts the performance of the victim's application. As such, ideally Düppel would be triggered only when necessary. Fortunately, we can limit the scope of protection by allowing the users of Düppel to define specific operations as sensitive so that cache cleansing is triggered only when sensitive operations run. For instance, in a TLS-protected web server, cryptographic routines in the OpenSSL library might be considered sensitive operations. The goal is to automatically enable Düppel when needed and disable Düppel when sensitive operations finish.

Düppel implements this mechanism by exploiting CPU page-level execution protections. Düppel first marks the memory pages that contain the instructions of the sensitive operations as non-executable. Thus every time these instructions are executed, page faults trigger Düppel. Then Düppel can disable the execution protection to prevent further page faults and start the cache cleansing timer

to trigger cleansing each of the time-shared caches N times (Section 5.2.1). At the end of the N-th cleansing, the execution protection is re-enabled.

If N is very small, unfinished sensitive operations will trigger the page fault again to initiate another N timer interrupts, and the overhead of frequently changing page-level protections can be huge. If N is too big, Düppel will keep running after the sensitive operation ends. In our design, Düppel dynamically adjusts N with the following algorithms: N changes value in the range of $[N_{min}, N_{max}]$, each time by adding Δ. Δ may vary in the range of $[\Delta_{min}, \Delta_{max}]$ and $[-\Delta_{max}, -\Delta_{min}]$. If the page fault takes place right after (i.e., no more than a threshold of *L* cycles) re-enabling execution protection, Δ doubles if it was positive or becomes Δ_{min} otherwise; if the page fault happens later than *L*, Δ doubles in the negative direction or becomes $-\Delta_{min}$.

Alternative Approaches. Instead of exploiting page execution protection, another way to dynamically enable and disable Düppel is to modify the user level application. For example, to protect dynamically linked libraries, the most convenient way is to modify the procedure linkage table entries of the relevant processes so that every library call related to the protected library function will first go through a wrapper function. Similarly it is possible to use preloaded libraries to overwrite (to provide a wrapper for) the library routines. In order to protect sensitive operations in the executables, one either needs to instrument the functions in the executables or insert breakpoints and use `ptrace` to intercept `SIGTRAP` signals sent to the protected process. These approaches, however, violate GOAL 3 and so are not adopted by Düppel.

5.2.2.2 Skipping Unnecessary Cache Cleansings

We further optimize Düppel to skip unnecessary cache cleansings. Before cleansing the L1 cache (and any other time-shared caches), Düppel first determines if the VCPU it is running on has been preempted since it last ran on the VCPU. If preemption took place, it then determines if the process interrupted by the current timer interrupt is related to the sensitive operations to be protected. If both answers are yes, Düppel will cleanse the cache; otherwise, it skips the cache cleansing step.

A paravirtualized Xen guest VM can detect VCPU preemption by exploiting shared data structures between the guest and the hypervisor. In particular, Xen uses a shared memory page that contains a value of the timestamp counter to help the guest VM keep track of the system time. The

counter value is updated at every hypervisor context switch if it is different from the version stored in the hypervisor, which is only changed every one second. At the end of each cache cleansing, Düppel modifies the shared counter value by one, which compared to the 64-bit counter value is rather small. (Greater changes may confuse the guest OS.) The next context switch will force a change of the counter value, and therefore will be detected by Düppel. One caveat is that the guest can voluntarily relinquish the VCPU, as well, which occurs when the VCPU is idle or at certain points during VCPU setup. Düppel instruments the code locations at which the guest VM kernel issues hypercalls to relinquish the VCPU, to help distinguish voluntary context switches and VCPU preemptions.

5.3 Implementation

We implemented Düppel as a kernel component (1.4K lines of C code) in paravirtualized Linux that runs on Xen guest virtual machines. Specifically, our prototype implementation modifies Linux kernel v2.6.32. It operates on both Xen 4.0 hypervisors in our lab setting and Xen 3.0 hypervisors (as reported) in Amazon Web Services.

Architecture. We provided an entry in the `procfs` file system which allows user-space tools to input user specified parameters to Düppel. These parameters include names of the protected applications and libraries. (It is currently possible to protect a subset or all of the libraries in one or multiple processes.) Our implementation extends the memory region management component of the Linux kernel to monitor the creation, deletion and splitting of the memory regions related to the protected applications and libraries. A list of pointers to these memory regions is maintained by Düppel. This list is adjusted dynamically by Düppel with the creation and termination of the protected processes. Read-Copy-Update (RCU) synchronization mechanisms are used to guarantee exclusive modifications and wait-free reads to this data structure with very low overhead. All memory pages belonging to the memory regions maintained in the list are disallowed to be executed by modifying the corresponding page table entries. This can be enforced by modifying the default page protection flags of the memory regions, which will be propagated to all newly mapped memory pages due to on-demand paging (Bovet and Cesati, 2005). Although the same memory pages can also be mapped to other processes in the system, they can be executed normally since they have separate page tables.

We also instrumented the page fault handler so that every page fault goes through an additional check: faults caused by user-space execute accesses to pages that are already in memory are passed to Düppel to further examine if the faults take place in the protected memory regions. As such page faults are otherwise rare, the performance overhead caused by modifications to the critical page fault handling procedure is minimal.

Workflow of Düppel. The main logic of Düppel works as follows: Düppel traps the page faults caused by executing the protected memory regions, and it then enables the page execution for all pages belonging to the list of memory regions. Then it initiates the periodic cache cleansing. After N cleansings have been performed, Düppel stops and enables page-execution protection in the page tables, indicating the end of a protection epoch.

Additionally, a counter recording the times a VCPU is preempted is maintained by Düppel to determine in which mode it should operate cache cleansing, i.e., sentinel mode or battle mode (Section 5.2.1). If the counter is greater than a threshold—empirically determined as 10 in our evaluation (see Section 5.4)—Düppel enters the battle mode; otherwise it runs in sentinel mode. The counter is updated as an exponential moving average (or EMA) of the number of VCPU preemptions in the current and previous time epochs, with $\lambda = 0.5$ where λ is a constant that determines the depth of memory of the EMA (Hunter, 1986) . Each time epoch is roughly $1\mathrm{ms}$ (rounded down to a number of CPU cycles that is a power of 2 for efficiency). Even if the attacker is aware of Düppel's parameters and refrains from preempting the victim (and so PROBing) less than 10 times per ms to cause Düppel to stay in sentinel mode, the L1 caches will nevertheless be cleansed at least 50 times per millisecond.

Cache cleansing modes and operations. When operated in sentinel mode, Düppel induces periodic timer interrupts and then tries to detect preemption (see Section 5.2.2.2) on all VCPUs. Once a preemption on a VCPU is detected, it sends an IPI to launch the next timer on that VCPU. As more preemptions are detected, Düppel enters battle mode, in which it will send IPIs to all the other cores that are available to process the IPI interrupts. The reason for sending IPIs to all such VCPUs (vs. only one) is to avoid cases where the VCPU being sent an IPI is not running (e.g., is preempted or out of credit to run), thereby causing the cache-cleansing interrupts to pause.

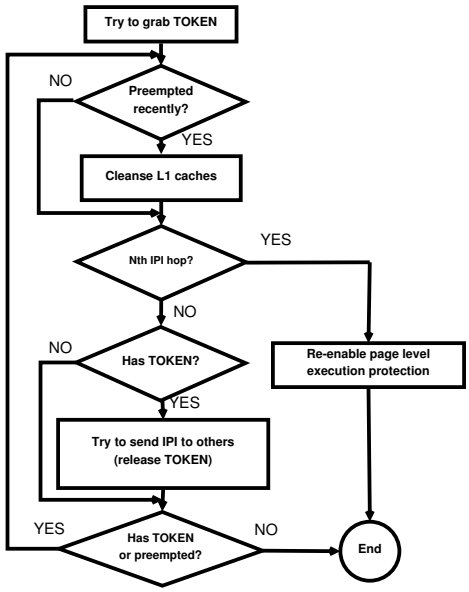


Figure 5.2: A token-based IPI synchronization algorithm.

We implemented a token-based VCPU synchronization algorithm in Düppel (shown in Figure 5.2) to avoid race conditions. An atomic variable serves as the token and one or more VCPUs simultaneously in the IPI context will try an *atomic exchange* to grab the token. So, only one VCPU will get it. After the cache cleansing job is done (or skipped), the VCPU with the token will try to send IPIs to all other VCPUs that have finished the last-round cache cleansing to pass along the token. As such, releasing tokens may not always be successful. Failure in passing the token or detection of another VCPU preemption at this point will force it to go over the cache cleaning cycle again. If the preemption counter goes below the threshold value, Düppel jumps back to sentinel mode.

5.4 Evaluation

5.4.1 Security Evaluation

In this section, we report the results of our security evaluation of Düppel in a lab environment. Our lab testbed was equipped with a single-socket quad-core Intel Core 2 Q9650 processor with an operating frequency of 3.0GHz. It had two levels of caches: both L1 data and instruction caches were 8-way set-associative and 32KB in size; two 24-way 6MB unified L2 caches, each served two CPU cores. All caches had 64-byte cache lines. We ran a Xen 4.0 hypervisor on the hardware; Dom0, the management domain in Xen, was given a single VCPU.

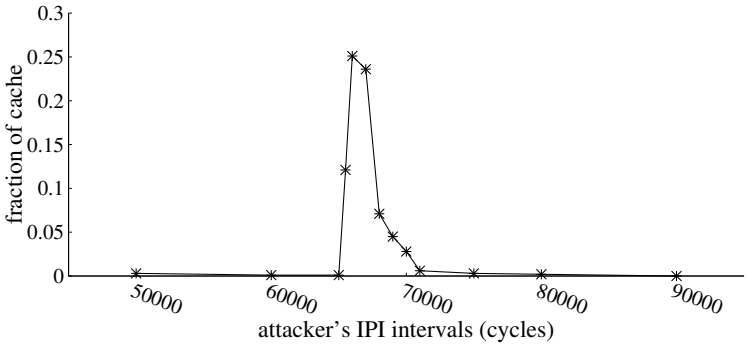
To evaluate Düppel under attack, i.e., when it operates with the presence of a side-channel attacker, we assigned two VCPUs to the VM that ran Düppel, and two VCPUs to a co-resident attacker VM. We facilitated the attack by pinning the VCPUs to the physical cores so that one attacker VCPU and one Düppel VCPU shared the same L1 caches. Düppel was configured to cleanse both the L1 data cache and the L1 instruction cache.

5.4.1.1 Attacking a "Dummy" Victim

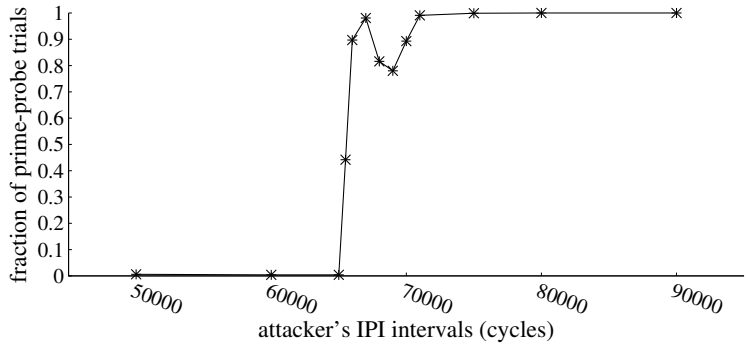
Victim. The "dummy" victim application in these experiments had two "dummy" functions composed of "nop" instructions. The instructions in each of them were mapped to different regions in the L1 instruction cache, occupying all cache lines in several cache sets. The victim application executed in a loop, alternating between these two functions. As such, the victim had distinguishable access patterns in the L1 instruction cache—all cache sets associated with the two functions were thoroughly evicted while other cache sets were almost entirely untouched.

Attacker. We leveraged the side-channel attack code by Zhang et al. (2012), which exploits IPIs to periodically preempt the victim VCPU and PRIME-PROBE the L1 instruction cache to collect timing results from each cache set. The two VCPUs in the attacker's VM have different roles: the *IPI VCPU* keeps looping and periodically sends IPIs to the *attacker VCPU*. The *attacker VCPU* is otherwise idle and only executes cache PROBing functions when triggered by receiving IPIs. The interval between two consecutive IPIs sent to the *attacker VCPU* is determined by the attacker, and is varied in our experiments as a tuned parameter. Each cache PROBing takes about 42000 CPU cycles (roughly 14μs), and one hypervisor context switch takes about 600 cycles. The IPI intervals, therefore, are the desired PRIME-PROBE intervals plus the time required for PROBing and context switches. Longer IPI intervals will give more chances to the victim to run, but multiple functions may run in the same PRIME-PROBE interval, leaving the cache too noisy to interpret. Shorter IPI intervals may cause more than one IPI to be delivered at the same time, making the PRIME-PROBE protocol fail. The range for the attacker's IPI interval that we evaluated was from 50000 CPU cycles to 90000 cycles.

Effectiveness of cache cleansing. Our first attempt in evaluating the effectiveness of Düppel's cache cleansing was to measure the fraction of cache lines *not* evicted by Düppel prior to the attacker



(a) Average fraction of cache potentially occupied by victim application data upon preemption

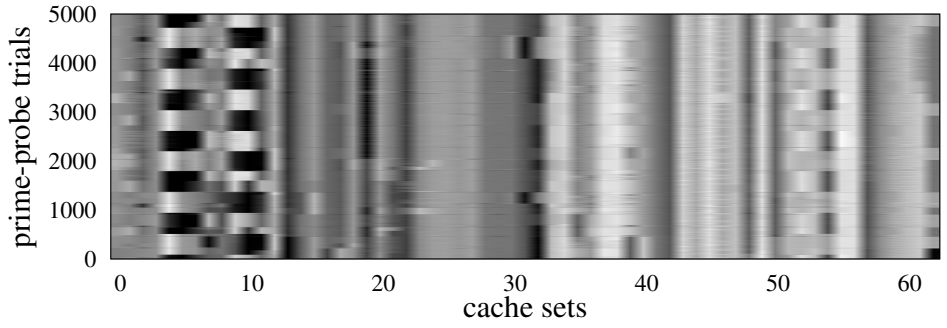


(b) IPI intervals during which the victim application executed at all

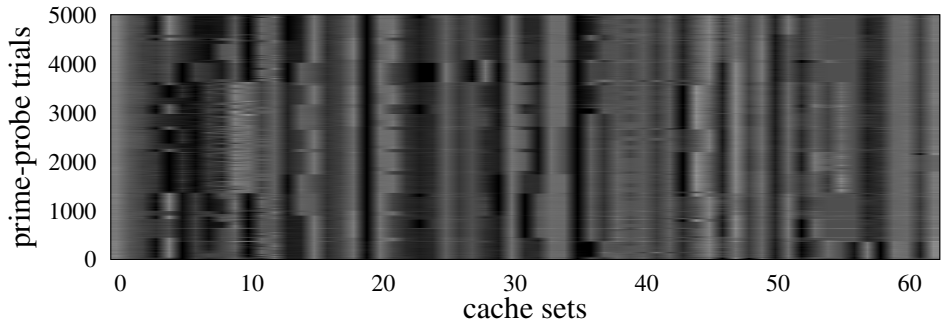
Figure 5.3: Cache cleansing effectiveness under different attacker IPI intervals

PROBing the cache. Information leaks can happen either when Düppel is not run between the attacker's PRIME and PROBE, or when Düppel is run but it does not have enough time to cleanse the entire cache before being preempted by the attacker. The results of our tests are shown in Figure 5.3(a). From the graph, we can see that when the attacker chooses a long IPI interval (>70000 cycles) or a short IPI interval (<65000 cycles), the potential information leak is minimal.

When the attacker's IPI interval is short (<65000 cycles), contention may result in IPIs issued by Düppel being significantly delayed or even "starved" by the attacker. However, recall that Düppel is designed to repeatedly cleanse the cache if it detects preemption; so even with less frequent IPI interrupts, Düppel will still consume most of the CPU time in cache cleansing to make sure the victim secret is not leaked through side channels. With longer IPI intervals (>70000 cycles), the IPIs issued by Düppel will be delivered on time and Düppel is able to finish the cache cleansing without being preempted, thus allowing the victim to run normally while being protected. However, when the IPI interval is between 65000 and 70000 cycles, Düppel only gets to cleanse a fraction of the cache before



(a) w/o Düppel



(b) w/ Düppel

Figure 5.4: Attacker's view of L1 instruction cache timings. The x-axis represents the cache sets; the y-axis represents the attacker's PRIME-PROBE trials. The darker the cell in the heatmap, the longer it takes the attacker to PROBE the cache.

being preempted. This is the worst-case scenario for Düppel's protection scheme. Figure 5.3(b) also helps illustrate this phenomenon. The y-axis in this figure is the fraction of attacker's PRIME-PROBE trials in which the victim application ran at all.

In the worst case, when the attacker's IPI interval is 66000 CPU cycles, about 25% of the cache lines are not evicted by Düppel but might contain victim application data. This provides the opportunity for some information leakage to the attacker, but the risk is far less than in the absence of Düppel. Figure 5.4 shows the attacker's cache readings from the PRIME-PROBE trials in this case. Each column in the heatmap represents the cache set indicated on the x-axis; the y-axis is the index of PRIME-PROBE readings. In Figure 5.4(a) we can see that without Düppel, the attacker can easily observe the cache pattern (the alternating cache usage on the left) of the victim application. In contrast, even with 25% of the cache lines possibly containing victim application data not evicted by Düppel, as shown in Figure 5.4(b), the victim's cache patterns are substantially obfuscated.

More quantitatively, in this worst-case scenario we measured the difference, for each cache set, of the average PROBE results (averaged over 100000 PRIME-PROBE trials) for that cache set when

one dummy function ran versus the average for that cache set when the other dummy function did. Without Düppel enabled, this difference-per-cache-set, averaged over all cache sets, was 32.7 cycles with a standard deviation of 13.2 cycles. However, cache sets to which one or the other dummy function mapped clearly divulged which dummy function had executed, exhibiting a difference-per-cache-set up to 190 CPU cycles — almost 12 standard deviations above the mean. (By comparison, the maximum difference-per-cache-set among cache sets not associated with either dummy function was only 22 cycles.) With Düppel enabled, the difference-per-cache-set, averaged over all cache sets, was 3.3 cycles with a standard deviation of 2.1 cycles. Among those cache sets to which either dummy function mapped, the maximum difference-per-cache-set dropped to 8 cycles, which is less than three standard deviations above the mean and, moreover, even less than the maximum difference-per-cache-set among cache sets not associated with either dummy function (which stayed basically unchanged from the Düppel-disabled case). In this respect, Düppel brought the timings of the cache sets occupied by the dummy functions largely "in line" with the timings of other cache sets.

5.4.1.2 Case Study: Square and Multiply

As a case study, we examined the victim application attacked by Zhang et al. (2012), namely the modular exponentiation implementation in the `libgcrypt` v.1.5.0 cryptographic library, which uses a textbook square-and-multiply algorithm. In this attack, the goal of the attacker was to extract a secret exponent used in this routine by PROBing the L1 instruction cache to infer the sequence of squares and multiplies executed. Here we focus on the accuracy of the first stage of the Zhang et al. (2012) attack, which involves classification of PRIME-PROBE timing vectors using a three-class support vector machine — one class for "Square", one for "Multiply", and one for "Reduce". While subsequent stages of the attack pipeline can filter out a small rate of misclassifications, a reasonably accuracy of the SVM classifications is necessary for the attack to work.

Training the SVM was performed using the same approach as taken in Zhang et al. (2012) work. In particular, to collect data for training, the "victim" repeatedly performed modular exponentiations, while informing the data collector via shared memory when it begins or ends a square or multiply. In this way, the data collector could associate its PROBE results of the L1 instruction cache with the ground-truth operation that was being performed when the PROBE was performed. Like Zhang

		Classification		
		Square	Multiply	Reduce
Op	Square	1740 (0.84)	43 (0.02)	298 (0.14)
	Multiply	14 (0.01)	2213 (0.94)	123 (0.05)
	Reduce	272 (0.05)	225 (0.04)	5072 (0.91)

Table 5.1: Confusion matrix of SVM classification without Düppel

		Classification		
		Square	Multiply	Reduce
Op	Square	26 (0.01)	699 (0.39)	1055 (0.59)
	Multiply	26 (0.01)	938 (0.48)	987 (0.51)
	Reduce	86 (0.01)	3367 (0.54)	2816 (0.45)

Table 5.2: Confusion matrix of SVM classification with Düppel

et al. (2012) we used a linear kernel in the SVM classifier. In each experiment, the SVM was trained with the labeled 90000 PRIME-PROBE results and then tested on an additional 10000 PRIME-PROBE results.

Table 5.1 and Table 5.2 shows the confusion matrix that resulted from an experiment in which both training and testing were conducted with Düppel disabled (Table 5.1) and one in which both training and testing were conducted with Düppel enabled (Table 5.2). With Düppel disabled, the SVM classifies the testing PROBE results with accuracy over 90%, which is well enough in our experience to enable the subsequent stages of the Zhang et al. (2012) attack. When Düppel is enabled, however, the SVM classification fails badly with an accuracy of only about 38%. In our experience, continuing the Zhang et al. (2012) attack from this point would be extremely difficult. This is, of course, not a proof that no exploitable side-channel still exists, and it is conceivable that a persistent and adaptive attacker could still make progress; however, we believe that Düppel should substantially increase the complexity of doing so.

5.4.2 Performance Evaluation

We conducted the performance evaluation of Düppel primarily in a public cloud environment, Amazon Web Services. The specification of processors in the Amazon EC2 cloud environment was not of our own choice. But through `cpuid` instructions, we determined that the platform was equipped with two 2.4GHz Intel Xeon E5645 processors, each of which had six cores per package with hyper-threading supported but disabled. Its L1 data caches were 32KB, 8-way set-associative;

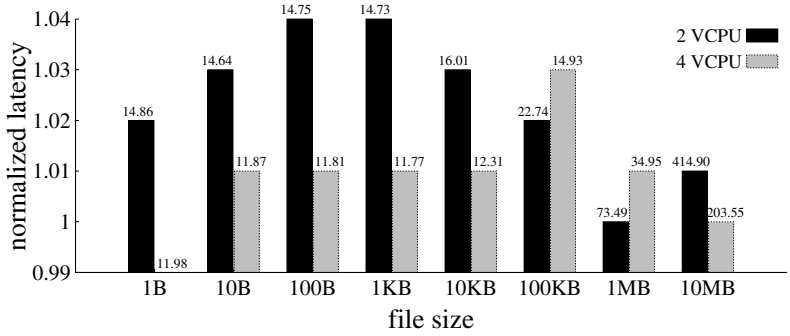


Figure 5.5: Düppel's overheads for file download latency with different file sizes. Labels on top of the bars represent the baseline latency (without Düppel) in ms.

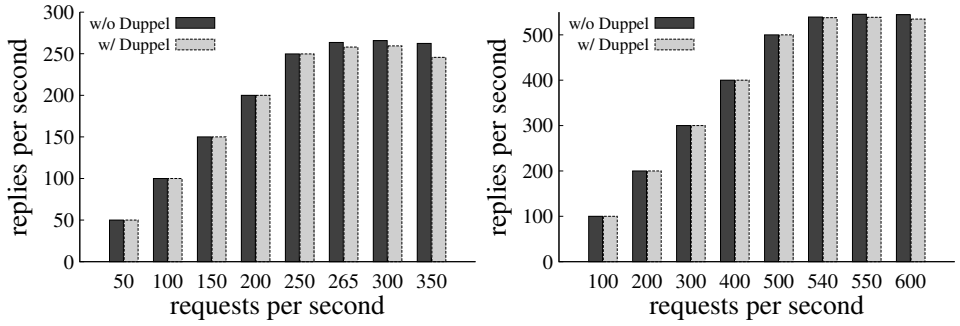
L1 instruction caches were also 32KB but with only 4 ways. One 8-way unified L2 cache was dedicated to each core, with 256KB capacity. Additionally, it had one 12MB 16-way L3 cache per CPU package. All caches had 64-byte cache lines. The shared info pages (Chisnall, 2007, Ch. 3) reported the Xen version to be `xen-3.0-x86_64`. The number of Dom0 VCPUs was unknown. As in the lab settings, we implemented Düppel in Linux kernel v2.6.32 in the cloud. Düppel was configured to protect the L1 caches and the unified L2 cache.

5.4.2.1 Securing TLS Libraries

In the following experiments we evaluate the performance overhead of Düppel when protecting `apache2` processes and the OpenSSL library (`libcrypto.so`) to which they were linked. We are particularly interested in preventing side channel attacks against the cryptographic operations provided by `libcrypto.so` during the Transport Layer Security (TLS) protocol used in `https` connections. We ran an apache 2.2.24 web server with `libcrypto.so` version 1.0.0. We ran the web server in one VM in the EC2 cloud and ran the client in another VM in the same availability zone in order to minimize network latency.

File download latency. Figure 5.5 shows results of an experiment to test the impact of Düppel on file download latency. We ran `apache bench` on the client and requested static pages with varying file sizes. The results reported are average values of 10000 requests for each file size. These experiments show that the impact of Düppel on file download latency was less than 4% in the worst case.

File download throughput. Figure 5.6 shows the file download throughput degradation induced by Düppel in our web server experiments. In these experiments, we used another web server performance



(a) 2-VCPU server (b) 4-VCPU server

Figure 5.6: File download rate in replies per second with different requests per second, with and without Düppel.

measurement tool, `httperf` (Mosberger and Jin, 1998), to produce the largest rate of requests for a 177-byte file (the default `index.html` file installed with `apache2`) for which the server could keep up. In order to saturate a server with 2 VCPUs, the requests were sent from 2 clients running `httperf` from distinct machines in the same availability zone in AWS; for the 4-VCPU server, 4 clients were employed in the test. Only one request was issued per connection and the SSL sessions were not reused. The timeout value for which `httperf` waited for the server's response was set to 1s. We confirmed by running the `top` utility that at its saturation point the server was CPU-bound; the throughput and server CPU usage both reached their limits simultaneously. Thus the throughput degradation was actually caused by Düppel rather than other factors. As this figure illustrates, the maximum throughput of the 2-VCPU server dropped from 267.1 replies per second to 258 replies per second, yielding a degradation of 3.4%. In the 4-VCPU case, the server's maximum throughput dropped only 1.4%, decaying from 545.3 replies per second to 537.7 replies per second. In all cases, the throughput overheads induced by Düppel were modest.

Local experiments. We repeated the above file download latency and throughput experiments in a more controlled environment in our lab. In these tests, the clients and server were connected through a 1Gb/s LAN. The qualitative results shown in Figure 5.5 and Figure 5.6 persisted in our local tests, e.g., with an induced latency degradation by Düppel of at most 3% and a throughput degradation of at most 2.1%.

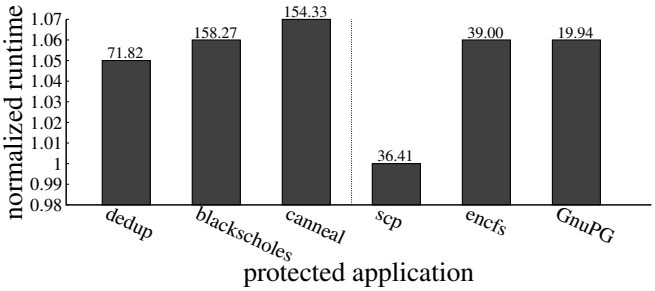


Figure 5.7: Runtime overhead of Düppel for different applications. Labels on top of the bars represent the baseline runtime (without Düppel) in seconds.

5.4.2.2 Securing Other Applications

We also evaluated Düppel in protecting other applications. We first picked three applications, `blackscholes`, `canneal` and `dedup`, from the PARSEC benchmarks (Bienia et al., 2008; Bienia and Li, 2009) to simulate different types of applications. `blackscholes` simulates financial analysis and, in particular, calculates the prices of a portfolio of options using Black Scholes partial differential equations. `canneal` uses cache-aware simulated annealing to design chips that minimize routing costs. `dedup` is short for "deduplication," which is a compression approach that combines global and local compression in order to obtain a high compression ratio. We selected these benchmark applications to represent CPU-bound applications with different amounts of memory and cache usage. The inputs to these benchmarks were `native` (see (Bienia and Li, 2009)) and the number of threads was the same as the number of VCPUs in the VM. We specified the entire executables to be sensitive, and so Düppel was always enabled while the programs were running.

In addition, we selected three applications that involve cryptographic operations, which are more likely to be of interest to attackers: `GnuPG`, `encfs`, and `scp`. `GnuPG` is part of the GNU project and implements the OpenPGP standard. We evaluated the time for it to decrypt a 1GB file encrypted using ElGamal encryption, with Düppel specified to protect `libgcrypt.so`. `encfs` is an encrypted filesystem in userspace. By encrypting a 1GB file, we evaluated the runtime of its file-encryption procedure with Düppel protecting its `libcrypto.so` library. `scp` is a network data transfer protocol built on top of secure shell (`ssh`). We transferred a 1GB file using `scp` to evaluate the latency overhead due to Düppel, again configured to protect `libcrypto.so`. All experiments above were run 10 times (except for `dedup` which was run 30 times due to large variance) on a

2-VCPU server in the AWS cloud. The average runtime degradations are shown in Figure 5.7. As can be seen there, Düppel induced less than 7% performance overhead in all cases.

5.4.2.3 Sentinel/Battle Mode Switching

As described in Section 5.2.1 and Section 5.3, a switch from sentinel mode to battle mode occurs whenever the moving average of the number of VCPU preemptions per millisecond exceeds an empirically determined, fixed threshold of 10 in our prototype. We developed a set of experiments to examine if this threshold will trigger spurious mode switches when co-located with regular, benign applications. In particular, we employed techniques similar to that described in Section 5.2.2.2 to detect VCPU preemptions. Instead of trying to detect a VCPU preemption during each timer interrupt, we tested VCPU preemptions in a loop so that every preemption was captured. We first ran experiments in our local testbed in which an application running on another VM (with only one VCPU) was pinned to share a core with the VM in which we counted preemptions. As such, the number of VCPU preemptions caused by the benchmark application on the shared CPU core was counted. Figure 5.8 illustrates the boxplot of the number of VCPU preemptions per millisecond (exponential moving average), where each box shows the first, second and third quartiles, each whisker extends to cover points within 1.5$\times$ the interquartile range, and outliers are marked with plus signs ("+"). As shown, these benchmark applications rarely caused the exponential moving average of VCPU preemptions to exceed the threshold of 10; e.g., the `apache2` web server induced a false alarm rate of 3% and all other benchmarks induced no false alarms. It is worth noting that in this test the `apache2` web server had been saturated already.

We also ran experiments in the Amazon EC2 cloud, specifically on four distinct instance types ("m1 medium", "m1 large", "m1 xlarge" and "c1 large") in each of three different availability zones in the US East region ("us-east-1a", "us-east-1c" and "us-east-1d"). The experiments were conducted from 8:00am Aug 10, 2013 to 8:00am Aug 11, 2013. Since we had no knowledge of the applications co-located with our VMs in the cloud, we assumed that the co-resident VMs and their applications were "benign" in terms of side channels. In these tests, the *maximum* false alarm rate witnessed on any of these machines was 8.22×10^{-6}.

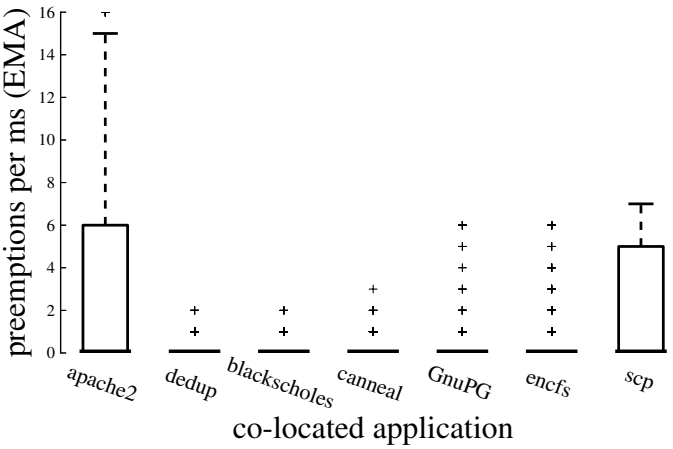


Figure 5.8: Victim VCPU preemptions per ms (exponential moving average) induced by co-located application in lab.

5.4.2.4 Performance Overhead When Being Attacked

Cache-based side-channel attacks, especially on the L1 cache, are essentially also performance degradation attacks. Moreover, Düppel will further degrade the whole system's performance when being attacked. In Figure 5.9 we show the results of our lab evaluation of the runtime overhead on `blackscholes`, `canneal` and `dedup` benchmarks under side-channel attacks on the L1 instruction caches, with and without Düppel. The victim was pinned to run only on the core being attacked for these tests. As can be seen in the figure, the attacker alone degraded the victim's performance by up to two orders of magnitude, and Düppel only compounded that cost. Suffering such a denial-of-service may be preferable to succumbing to side-channel attacks. Moreover, since VCPUs are usually not pinned in real clouds, the duration of such a denial-of-service can be expected to be short.

5.5 Discussion

5.5.1 Extensions of Düppel

Extension to other cache side channels. In principle, Düppel can defend against side-channel attacks on other types of time-shared caches, as well. For instance, the BTB and TLB can be cleansed together with L1 caches. However, BTB attacks were only demonstrated in non-virtualized systems, while cross-VM attacks on TLBs are unlikely because TLBs are flushed on context switch.

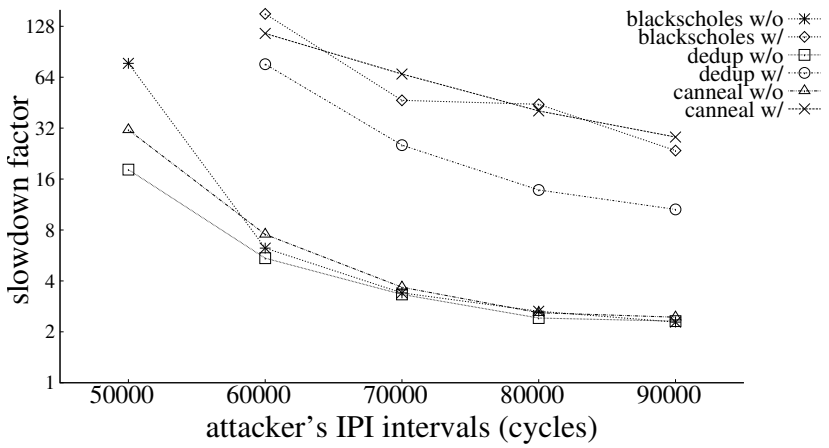


Figure 5.9: Performance overhead (averaged over five runs) when under PRIME-PROBE attacks on L1 instruction caches, w/ and w/o Düppel.

Extension to other clouds. Düppel can be implemented in the guest OS for cloud platforms other than Amazon, as well, as long as the virtualization substrate supports paravirtualized Xen guests and customized OS kernels. We expect Düppel to also work on Rackspace and GoGrid, for example.

Extension to kernel routines. Düppel can be extended to protect sensitive operations in the guest OS kernel, as well. The difficulty, however, is that the Linux kernel is monolithic, and so the code and data memory sections of kernel routines are not cleanly separated from the rest of the kernel. Düppel will have to look for all memory pages that contain kernel routines and mark them as non-executable.

5.5.2 Limitations

SMT-enabled cloud platforms. When SMT is enabled on processors, some time-shared caches can become simultaneously shared caches (e.g., L1 caches). Düppel may not be effective in this case. However, from our previous experience (Zhang et al., 2012), as well as prior documented evidence (Percival, 2005; Osvik et al., 2006), SMT-enabled CPUs are more vulnerable to side-channel attacks. Moreover, as clouds often charge by CPU computing units, with SMT-enabled cores the estimated computing units per VM would become unreliable if more than one VCPU shared the same CPU core at the same time. Therefore, we anticipate that cloud providers will not enable SMT in production clouds for both security and accounting considerations.

The simultaneously shared caches. A simultaneously shared cache, e.g., the last level cache, might also be targeted by side-channel attacks. Again, Düppel does not address this possibility. Compared

with time-shared caches, PROBing simultaneously shared caches can be done without VCPU preemption and with frequencies limited only by the cache and memory access latencies and the sizes of the caches utilized. Although fine-grained cache observations are possible in such situations, the challenges that attackers face to extract sensitive information may be prohibitive. The first challenge stems from the fact that most LLCs are physically indexed. As such, attackers must have prior knowledge of the physical addresses of the memory regions of the victim application (Mowery et al., 2012), which is usually allocated dynamically and may vary each time it is run. Second, exploiting last level caches when the victim and attacker run on different cores can be sensitive to cache coherence properties. For example, PROBing on an LLC in an exclusive caching hierarchy (as compared with inclusive caches) will not invalidate contents in the lower-level caches of another core. As such, the victim's activity may be hidden in lower level caches and never leaked to the last level cache at all. Moreover, to our knowledge, last level caches in the processors used in cloud platforms usually serve more than two cores; in this case, side-channel observations are subject to more background noise from cores that are not running the victim application. We believe for these reasons, side-channel attacks in simultaneously shared caches in virtualized SMP environments have been limited so far to extracting only coarse information (Ristenpart et al., 2009) or have needed to leverage additional hypervisor features (e.g., sharing memory pages between the victim and attacker VMs (Yarom and Falkner, 2013)).

5.6 Summary

In this chapter, we presented Düppel, a system to mitigate cache side channels in public clouds by retrofitting commodity operating systems. Düppel cleanses time-shared caches (e.g., per-core L1 and L2 caches) to confound side channels through them. We detailed our implementation of Düppel in Linux for paravirtualized Xen, demonstrated the security effectiveness of our prototype in tests on lab machines, and evaluated the performance impact of our prototype on a public cloud. To our knowledge, Düppel is the first general and efficient technique to enable tenants to defend themselves from cache-based side-channel attacks in public clouds, without help from the hypervisor or cloud operator.

CHAPTER 6: CROSS-TENANT SIDE CHANNELS IN PAAS CLOUDS

So far in this dissertation, we have explored both attacks and defenses of cache-based cross-VM side channels in the public IaaS clouds. To further support the thesis of this dissertation, in this chapter, we elaborate our study of cross-tenant side-channel attacks specifically in PaaS clouds. We consider attacks by the PaaS provider (or other malicious insiders) as out of scope. The same trust extends to any underlying IaaS provider, if the PaaS cloud runs atop an IaaS service. Should the IaaS cloud be public (e.g., EC2) then its malicious IaaS customers represent a threat to PaaS customers, but not one that we explore further. Rather we focus on other malicious customers of the PaaS cloud, and container-based isolation in particular. Thus both the adversaries and the victims in our threat model are users of a PaaS system. An attacker seeks to (i) arrange for a malicious instance it controls to be scheduled to run within a different container on the same host OS as the target victim and (ii) extract confidential information from the target victim using this vantage point.

The chapter is organized as follows. Section 6.1 describes our attack framework. Section 6.2 discusses our strategies for achieving and confirming co-location of attacker instances with victims. Sections 6.3–6.5 then detail our three attack demonstrations outlined in Section 1.4. We discuss some potential countermeasures to our attack and the ethical issues involved in conducting our evaluation in Section 6.6 and we conclude this chapter in Section 6.7.

6.1 Attack Framework

In this section, we present an attack framework that enables an attacker to design a cache-based attack to track the execution path of a victim and, in doing so, to extract a secret of interest from the victim. We will first describe the FLUSH-RELOAD-based side channels exploited in this study (Section 6.1.1), and then develop an *attack nondeterministic finite automaton* or *attack NFA* (Section 6.1.2) from the control-flow graph of an executable shared with the victim. We defer the actual demonstration of security attacks to later sections.

6.1.1 Side Channels via Flush-Reload

We leverage a type of cache-based side channel that was first reported by Gullasch et al. (2011), who demonstrated its use by an attack process to extract Advanced Encryption Standard (AES) keys from a victim process when both were running within the same OS. The attack was studied on a single-core processor and exploits the attacker's process' ability to evict data in physical memory pages it shares with the victim process from the CPU cache (e.g., via the instruction *clflush*). The technique was later extended by Yarom and Falkner to multi-core systems with a shared last-level cache (Yarom and Falkner, 2013). They refer to their attack as FLUSH-RELOAD. In this work, we further extend the use of the FLUSH-RELOAD side channels and apply it to more general attack scenarios.

Basic FLUSH-RELOAD. The basic building block of a FLUSH-RELOAD attack is as follows.

FLUSH: The attacker flushes chunks (defined in Section 2.1.1) containing specific instructions located in a memory page it shares with the victim out of the entire cache hierarchy (including the shared last-level cache) using the *clflush* instruction.

FLUSH-RELOAD interval: The attacker waits for a pre-specified interval while the last-level cache is utilized by the victim running on another CPU core.

RELOAD: The attacker times the reload of the same chunks into the processor. A faster reload suggests these chunks were in the last-level cache and so were executed by the victim during the FLUSH-RELOAD interval; a slower reload suggests otherwise.

We refer to the chunks being FLUSH-RELOADed by the attacker as being *monitored*, since FLUSH-RELOAD essentially monitors access to data in the chunk.

Flush-Reload Protocols. We next define a FLUSH-RELOAD protocol, in which the attacker process monitors a list of chunks simultaneously and repeatedly until instructed otherwise. It will first try to RELOAD the first chunk, record the reload time and FLUSH it immediately afterwards. Then it will repeat these steps on the second chunk, the third, and so on, until the last chunk in the list. Then the attacker will wait for a carefully calculated time period before starting over from the first chunk, so that the interval between the FLUSH and RELOAD of the same chunk is of a target duration.[1] From

[1]Variation in the duration on the order of one or two hundred CPU cycles may occur as the reload of a chunk does not take constant time.

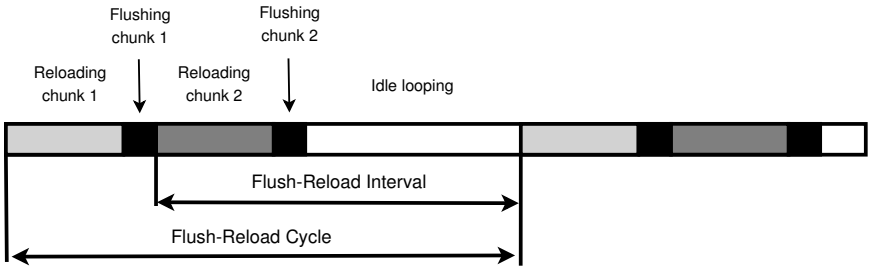


Figure 6.1: An example of a FLUSH-RELOAD protocol in which two chunks are monitored at the same time. Gray rectangles are RELOADs of two chunks and dark squares are immediate FLUSHes of the prior RELOADs.

the RELOAD of the first chunk to the end of the waiting period is called one *Flush-Reload cycle*. An illustration of the FLUSH-RELOAD protocol is shown in Figure 6.1.

Compared with the PRIME-PROBE attacks used in many previous works (e.g., (Percival, 2005; Neve and Seifert, 2007; Tromer et al., 2010; Zhang et al., 2012)), FLUSH-RELOAD attacks involve relatively less noise, since the attacker is able to tell whether the victim accessed the data in the chunk the attacker is monitoring, versus simply some data mapped to the same cache set. The technique still suffers from many sources of noise in practice.

Sources of noise. We discuss, in turn, noise due to race conditions, unobserved duplicate reloads, false sharing of cache lines, and multiple processes using the same memory pages. These affect the granularity and reliability of the attacks that we will develop in subsequent sections.

A race condition here refers to the situation where two memory loads of the same chunk are issued from two CPU cores roughly at the same time. Due to the complexity of the cache snooping protocols, the outcome of the RELOAD step can be unpredictable in such cases. The access of the shared chunk by the victim may be missed by the attacker if it overlaps with the attacker's memory load. Because the attacker increases the risk of such an overlap as it shortens its FLUSH-RELOAD interval, the attacker is limited in how far it can shrink that interval.

Another source of noise is the victim itself—a victim's first access of a chunk can be missed by FLUSH-RELOAD monitoring if it accesses the chunk a second time before the RELOAD of the attacker. This type of noise is particularly significant when applying the attack framework to count the repeated use of the same chunk, which will be discussed in our attack scenarios.

False sharing usually refers to a cache usage pattern in distributed, coherent cache systems that degrades the performance of the cache (Bolosky and Scott, 1993). Here we refer to false sharing of a cache line to refer to cases in which two separate program components share the same chunk and hence the FLUSH-RELOAD probing of one component may be misled by the execution of another. For example, the memory layout of a function rarely aligns perfectly with chunks, and the beginning and the end of a function usually share the same chunks with other functions.

As multiple processes in the operating system may share the same executables, and so the same memory pages that contain executable code, activities from processes other than the victim may trigger false positives in the RELOAD phase. Especially in PaaS cloud settings, tens or even hundreds of applications may share the same set of executables, and careful use of the FLUSH-RELOAD side channel is required.

6.1.2 From CFGs to Attack NFAs

In this section we provide a framework to leverage FLUSH-RELOAD attacks as a primitive in a larger attack strategy to trace the execution path of a victim instance during (at least part of) its execution. Specifically, we develop *attack NFAs* that prescribe the order in which different chunks should be monitored using FLUSH-RELOAD attacks, based on what has been learned so far.

The development of an attack NFA to attack a target victim begins with a control-flow graph (CFG) (Allen, 1970) of the executable shared with the victim. As usual, each node of the CFG is a *basic block* of instructions, and an edge from one basic block to another indicates that the latter can immediately follow the former in execution. Let B denote the set of basic blocks of the victim instance, and let E denote the directed edges of the CFG.

When the shared executable is loaded, its organization in memory determines a function $\mathsf{chunks} : B \to \mathbf{2}^{\mathcal{C}}$ that describes how each basic block shared with the victim (i.e., in the shared executable) is mapped to one or more chunks in the attacker instance's virtual memory. Here, $\mathcal{C}$ is the set of all chunks in the attacker's virtual memory occupied by the shared executable, and $\mathbf{2}^{\mathcal{C}}$ denotes the power set of $\mathcal{C}$. That is, each basic block in B is mapped to one or more chunks by chunks.

Like a regular NFA, the attack NFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of symbols, $\delta : Q \times \Sigma \to Q$ is a transition function, q_0 is the initial NFA state, and

$F \subseteq Q$ is a set of *accepting* states. To each state $q \in Q$ is associated a not-necessarily-unique set of chunks, denoted $\mathsf{mon}(q) \subseteq \mathcal{C}$, that contains the chunks the attacker will monitor while in state q.

The symbols Σ consumed by the NFA is the set $\Sigma = \mathcal{C} \times \mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. Specifically, the meaning of the transition $(q, (c, \ell, u), q') \in \delta$ is: while in state q and so monitoring the chunks $\mathsf{mon}(q)$, if the attacker detects the victim's use of c within the interval $[\ell, u]$ (in units of FLUSH-RELOAD cycles), then the attacker transitions to q' and begins monitoring the cache lines $\mathsf{mon}(q')$. We allow ℓ to be zero; detecting the victim's use of c in zero FLUSH-RELOAD cycles since entering state q means that c was detected in the same FLUSH-RELOAD cycle that caused state q to be entered.

In light of this intended meaning of the attack NFA, the transition function should satisfy certain constraints.

- **Observability**: If $(q, (c, \ell, u), q') \in \delta$, then $c \in \mathsf{mon}(q)$. Otherwise, an attacker in state q will not observe the victim using c. If in addition $(q', (c', 0, u'), q'') \in \delta$, then $\mathsf{mon}(q') \subseteq \mathsf{mon}(q)$, since for transition $(q', (c', 0, u'), q'')$ to become enabled with no FLUSH-RELOAD cycles after transitioning to q', c' must be monitored in q (as must other chunks included in $\mathsf{mon}(q')$ due to recursive application of this rule to additional "downstream" states like q'').

- **Feasibility**: To each state q there corresponds a basic block b such that for each transition $(q, (c, \ell, u), q') \in \delta$, there is a (possibly empty) path in the CFG from b to a basic block b' (corresponding to q') that can be traversed in no fewer than ℓ and no more than u FLUSH-RELOAD cycles and such that $c \in \mathsf{chunks}(b')$. Intuitively, it is this execution path that the attacker detects in transitioning from state q to q'.

A transition is taken out of a state at the first FLUSH-RELOAD cycle that enables a transition. Still, it is possible for multiple transitions to become enabled in the same FLUSH-RELOAD cycle, in which case all such enabled transitions are taken. In this respect, the automaton is nondeterministic. In practice (see Sections 6.3-6.5), it is important to design the attack NFA so that the number of simultaneously active states is constrained, since monitoring large numbers of chunks simultaneously poses difficulties.

Also, there is a "catch-all" transition out of every state q to a designated failure state $q_\perp$, which is taken if the FLUSH-RELOAD cycle limit u of every transition $(q, (c, \ell, u), q') \in \delta$ is exceeded

with none of the transitions being taken. Once in this failure state, the NFA stays in that state. The accepting states F of the NFA are all states other than this designated failure state.

The designated initial state q_0 represents the shared executable's entry point(s) of interest to the attacker. That is, $\mathsf{mon}(q_0) \cap \mathsf{chunks}(b) \neq \emptyset$ for each basic block b that the attacker wants to detect initially.

At this point in our research, construction of an attack NFA from the CFG and chunks is a manual process aided by static analysis of the shared executables, offline dynamic analysis (instrumentation using Valgrind, a memory debugging and profiling tool) and source code inspection. Once a set of chunks are selected to be monitored, the attack NFA can be constructed by offline training, where all (or some of the) chunks are monitored simultaneously and their sequential order and relative timing are recorded. This process depends in large part on the attack goals—in particular, which execution paths in the victim the attacker needs to detect. In Sections 6.3–6.5 we will give several examples of how to construct attack NFAs for different types of attacks.

6.1.3 Applying Attack NFAs

In our attacks, the adversary employs an attack NFA $(Q, \Sigma, \delta, q_0, F)$ to reconstruct the victim's execution path by simultaneously (i) triggering the victim's execution by sending a request to victim's web application interface, and (ii) inducing its co-located attacker application to start monitoring $\mathsf{mon}(q_0)$. While the NFA remains in an accepting state, the adversary knows the execution path of interest taken by the victim. We have found that in practice, a well designed NFA usually leads to successful identification of an execution path of the victim application.

6.2 Co-Location in PaaS

To exploit side-channels in PaaS environments, an attacker must first somehow achieve co-location of a malicious instance on the same OS as a target. Ristenpart et al. (2009) explored co-location vulnerabilities in the setting of IaaS clouds. To the best of our knowledge, no one has investigated co-location in PaaS settings. We therefore provide a preliminary empirical study of the ability to co-locate an attacker instance with a victim instance in modern public PaaS clouds, leveraging our proposed attack framework to detect success.

Co-location attacks consist of two steps. First, the attacker employs some strategy for launching (typically a large number of) instances on the cloud service. Second, each of these instances attempts to perform co-location detection. For the first step, we explore only the simplest strategy in which we repeatedly launch instances that check for co-location until success is achieved.

Co-location detection. For the second step, we use a FLUSH-RELOAD side-channel to detect whether any of the instances co-locates with the victim instance. To detect co-location, the adversary sends an HTTP query to the victim instance and instructs each of the attacker instances to simultaneously monitor a certain execution path using the techniques proposed in Section 6.1. If the execution path is detected, the adversary will have some confidence that the detecting attacker instance is co-located with the victim. However, this approach may have false positives due to activities of other tenants sharing the same OS. In order to increase the confidence, two strategies can be taken: (1) induce and monitor for rare events to eliminate false positives; or (2) use multiple trials to reduce false positives and false negatives.

The execution path to be monitored may vary. In our experiments, we considered a victim instance that ran a popular PHP e-commerce application, Magento. To differentiate the query sent by the adversary from background noise, we simply used a relatively unusual query with an associated uncommon execution path. By inspecting the source code of the Magento application, facilitated by dynamic analysis using Valgrind, we found the function `xmlXPathNodeSetSort()` in `libxml2.so` and `php_session_start()` in either `libphp5.so` or `php5-fpm` (depending on the version of PHP used by the cloud) are called sequentially during a (failed) login attempt. We confirmed with dynamic analysis of other types of queries that the execution paths that traverse both functions are uncommon. Therefore, we constructed an attack NFA as shown in Figure 6.2. The edge labels indicate the monitoring set: in this case $\{2\}$ indicates `php_session_start()`, and the number of FLUSH-RELOAD periods allowed is any in the range $[1, T]$. One can of course adapt the above strategy easily to targets beyond Magento.

We observed in earlier experiments that some cloud services tend to schedule applications with different runtimes (e.g., PHP versus Ruby) on different machines. Fortunately (for an attacker) it is easy to choose the same runtime as the victim should it be known to the attacker, which we did in all

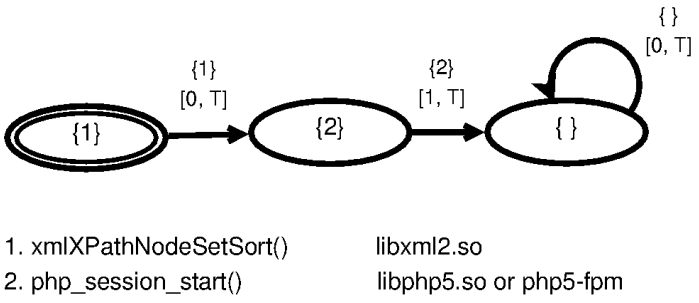


Figure 6.2: The attack NFA used for detection of co-location with PaaS Magento e-commerce instances. Initial state q_0 indicated with double ovals, and error state $q_\perp$ not shown. T is the maximum FLUSH-RELOAD cycles before transitioning to $q_\perp$.

of our experiments. If it is not known, the attacker can simply repeat the co-location attack for each runtime as there are only a handful in any given cloud.

Co-location validation. To obtain ground truth for evaluating efficacy, we took advantage of the fact that during our experiments we controlled both the attacker and victim instances. In particular, we augmented the above procedure to also have both attack instances and the target victim establish a TCP connection with an external server under our control. (Most clouds have their firewalls configured to allow outbound traffic.) This revealed the IP addresses associated with each instance; if two instances shared the same IP address they were hosted on the same (virtual) server. It is worth noting that a Network Address Translation (NAT) configuration in the cloud provider's network would hinder this approach. However, we did not observe this problem in our experiments. We also note that this co-location check could potentially be used in cases in which real attackers can obtain the IP address of the target, and so this might be directly useful by real attackers. However in many cases clients do not directly connect to PaaS instances, hitting a load balancer or HTTPS endpoint first. Thus we only used the IP comparison approach to validate that the previously described side-channel based co-location check worked.

Co-location experiments. We provide some initial proof-of-concept experiments regarding the ability of an attacker to obtain co-location with a single victim. We do so for two popular public PaaS services: DotCloud and Openshift.

The client control interfaces are different in the two services. In Openshift, a target victim instance was launched and after a certain amount of time (typically on the order of a few hours, though times varied), the adversary launched attack instances one-at-a-time (with a 30-second interval

	Trials			Miss Detection	
	1st	2nd	3rd	FP	FN
DotCloud	< 10	< 10	< 10	0.00	0.03
Openshift	98	120	5	0.00	0.49

Table 6.1: Number of sequentially launched instances before co-location

to reduce the stress of the experiments on the cloud fabric) until one obtained co-location with the victim as indicated by the attack NFA. In DotCloud, the experiments were conducted similarly, except that the control interface enabled us to launch attack instances ten-at-a-time via static scaling (vs. dynamic scaling in Openshift). We repeated this process three times for each cloud. We report in Table 6.1 the number of instances that the adversary launched before a successful co-location. As can be seen, every trial succeeded in every cloud, providing strong evidence that an attacker is very likely to be able to obtain co-location with a target. (Indeed, in the course of writing this dissertation, we never were unable to achieve co-residency with our victim instance in these clouds.) The number of trials required, however, varied greatly. Even in the worst observed case, with 120 instances in Openshift, nevertheless, co-location was obtained after 3.2 hours and at a total cost of zero US dollars, as we did not exceed the limits of the free tier.

We used this experimental data to test the accuracy of our co-location detection attack NFA. Specifically, we ran for each cloud the co-location detection 100 times using two instances which were co-located (as per IP address checks) and 100 times on instances which were not co-located (as per IP address checks). The detection rates are also shown in Figure 6.1: *FP* indicates the rate of false positives in which not-co-located instances were reported as co-located, and *FN* indicates the rate of false negatives in which co-located instances were not reported so. We believe the high false negative rate in Openshift was due to CPU resource contention, as the applications were run on a two-core VM sharing CPUs with hundreds of processes. Nevertheless, the result indicates the rare execution path represented in Figure 6.2 successfully reduced background noise; repeating the co-location test five times resulted in a false negative rate of less than 5%.

6.3 Case Study 1: Inferring Sensitive User Data

In this section and the two that follow, we present three examples in the form of case studies that demonstrate how an adversary can apply our proposed framework to accomplish a variety of real-world attacks. Our experimental environment was common to all three studies.

Experimental environment. Our evaluations were conducted in a public PaaS cloud, DotCloud. We will discuss the ethical considerations surrounding our experiments in Section 6.6.2. The software and hardware stack in DotCloud was out of our control and was not officially reported by the provider. By observing data extracted from the `procfs`, a pseudo filesystem presenting system information, and data available from the PaaS control fabric, however, we believe the applications in our experiment were run on a VM with four virtual CPUs operated by Amazon EC2 in `us-east-1a` datacenter. The physical CPU was a 2.4GHz Intel Xeon processor E5-2665, which has 8 cores sharing a 20MB last-level cache. Moreover, we believe the operating systems that support the applications were Ubuntu 10.04.4 LTS on Linux kernels version 2.6.38. The tenants were isolated with Linux containers.

In all three case studies, we created two accounts in DotCloud, designating one of them as the victim account and the other as the attacker account. As in all cases the victims are PHP applications, all attacker applications were designed to operate on the same runtime to facilitate its colocation with the victim, which was achieved as described in Section 6.2. In DotCloud, PHP was operated as `php-fpm` (version 5.4.6), which interacts with the Nginx web server (`http://nginx.org`) and processes PHP requests. In all experiments, the FLUSH-RELOAD cycle was set to be 2400 clock cycles, corresponding to about one microsecond in real time.

6.3.1 Attack Background

Our first case study explores a relatively simple attack, a good starting point for end-to-end illustration of our techniques. We show how our proposed attack framework permits inference of the responses of a victim web application to client requests. Specifically, an attacker may combine what is known as a cross-site request (CSR) with the FLUSH-RELOAD side channel to infer the number of distinct items in a user's shopping cart on an e-commerce server.

There have been various related *timing* attacks demonstrated on web privacy (Felten and Schneider, 2000; Nagami et al., 2008). Particularly similar to our case study here is a CSR-based attack described by Bortz and Boneh (2007) that likewise infers the number of distinct items in a user's shopping cart. As their attack relies on the timing of request fulfillment, they propose and implement a countermeasure that enforces uniform server response timing. The attack we present here depends instead on execution tracking via an attack NFA, and thus defeats timing-side-channel countermeasures of this kind.

Cross-site requests. The target of the attacker in this case study is, specifically, a user that is authenticated to a victim e-commerce site. We presume, however, that the attacker cannot compromise the credentials of the user for the victim site, and only makes use of the side channel to observe data retrieved by the user. A passive attacker might be unable to determine the identities of users accessing the victim site. We consider an alternative strategy in which the attacker prompts user retrieval of the target data by means of a *cross-site request* (CSR).

CSRs are HTML requests made to a third-party resource, that is, one hosted by a domain other than that serving the HTML. While there are legitimate uses for such indirection, it can also serve as a basis for requests that make improper use of a user's credentials, as in our attack here. A CSR requires that the attacker lure the user to a site that serves HTML crafted by the attacker to redirect the user's browser to the victim's e-commerce site, e.g., `<img src = "http://victim-site.com/index.php/checkout/cart/">`.

If a user Alice has been previously authenticated to the domain `www.victim-site.com`, then her browser will often obtain and cache credentials for the domain, such as cookies, and automatically re-authenticate on subsequent visits. Thus, in our attack, `www.victim-site.com` will see an authenticated request originating from Alice's browser, without awareness that the request was triggered by an attacker.

Web applications may include protections against malicious CSRs, such as requiring explicit user authorization of resource requests. Often these protections are confined, however, to what are called cross-site request forgery (CSRF) attacks, which cause state changes (known as "side-effects") in the server. The CSR we exploit for our attack here has no side effects, and will thus be allowed by many victim servers.

6.3.2 Evaluation in Public PaaS

We empirically evaluated our proposed attack in DotCloud against the Magento e-commerce application (latest version). This is an especially popular open source e-commerce application, used by roughly 1% or about 200,000 of the top 10 million websites (W3Techs, 2014) ranked by Alexa (`http://www.alexa.com`). We reiterate that our goal is for an attacker instance to determine reliably the number of distinct items in an authenticated user's shopping cart on the e-commerce site of the victim. Our attack cannot determine the quantity count for a given item.

We assume, as noted above, that the attacker can lure an authenticated user of the victim Magento website to an HTML page hosted in its own webserver, thereby triggering a CSR in which the user requests her shopping cart on the victim's web server. We simulated the user on Google Chrome (version 34.0.1847.132). We expect the attack to work on other browsers as well, which support a similar range of cross-origin requests.

Attack details and results. The attack NFA we constructed for the attacker in this example is highly dependent on the specifics of the Magento web application. We analyzed the application with Valgrind. We observe that a Zend opcode handler (which we call `handler()` for convenience[2]), which is implemented in the executable `php5-fpm`, is invoked every time an item in the shopping cart is displayed.

To count the number of items in a shopping cart, therefore, it suffices for the attacker to count the number of invocations of the `handler()` function using the FLUSH-RELOAD side channel. In our experiments, an interval of at least 20 FLUSH-RELOAD cycles elapses between the display of two distinct items. We take this interval length to be a lower bound on the time between calls to `handler()` within the NFA we construct for the attack, as depicted in in Figure 6.3.

The evaluation was performed on DotCloud as follows. The victim user placed m distinct items in her shopping cart, for $m \in \{0, 1, 2, 3, 4, 5, 6\}$. We repeated our experiment 10 times for each value of m. The number of successes for each number m of distinct items, that is, the frequency with which the attacker correctly determined m from a single trial, is shown in Table 6.2. Also shown is that when the attacker inferred m incorrectly, its inference was nevertheless very close to correct.

[2]As only the virtual address of the handler was required to construct the attack NFA, we were able to perform the attack without studying the Zend source code. Therefore, the name of the function, which is hidden in the result of an `objdump`, remains unknown to us.

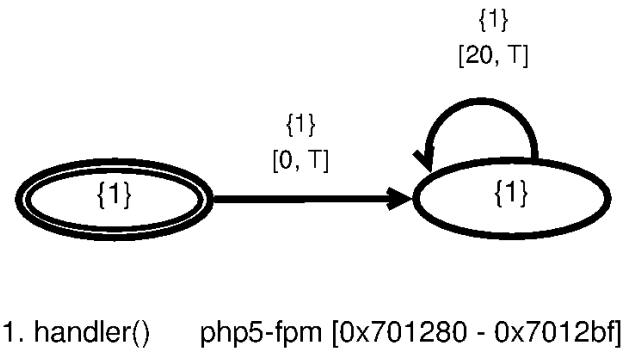


Figure 6.3: The attack NFA for case study in Section 6.3. Initial state q_0 indicated with double ovals, and error state $q_\perp$ not shown. T is the maximum FLUSH-RELOAD cycles before transitioning to $q_\perp$.

Items in cart (m) \ Items detected in cart	0	1	2	3	4	5	6	7
0	10							
1		10						
2			9	1				
3				10				
4				1	9			
5					1	9		
6						1	8	1

Table 6.2: Item count inferences by the attacker. Each table entry indicates the number of experiments yielding a given (true count, inferred count) pair over 10 trials per row. Entries on the diagonal, which predominate, correspond to correct inference.

6.4 Case Study 2: Password-Reset Attacks

In this second case study, we show how to employ our attack framework to compromise the pseudorandom number generators (PRNGs) used by many web applications in authenticating password reset requests. An attacker can exploit this ability to reset the passwords for and thus obtain control of the accounts of arbitrarily selected users.

Our attack targets the PRNG present in certain programming language runtimes (e.g., PHP), which relies upon system time (e.g., `gettimeofday()`) as a source of seed entropy. With a malicious application that is co-located with the victim application, the adversary is able to detect system calls such as `gettimeofday()`, reconstruct the internal state of the PRNG, and thereby reproduce its entire output.

The ability to mount password-reset attacks is one consequence of this PRNG vulnerability. Such attacks are of particular concern because an adversary can trigger a password reset on a web application for a user with knowledge of the user's account name or email address alone. To authenticate the user, a web application will typically use a PRNG to generate a random string R, and then embed this string in the URL of a password reset link sent to the user's registered email address. By learning the state of the web application's PRNG, a co-located attacker instance can reproduce the password reset token R, reset the password before the user does, and hijack the user's account. We stress that the adversary does not need access to the user's email to accomplish this attack. In this section, we demonstrate such a password-reset attack against PHP-based web applications in public clouds.

Weaknesses in PHP PRNGs have been previously reported, e.g., (Esser, 2008; Kamkar, 2010). A recent study by Argyros and Kiayias (2012) gave several attacks, one of which involves recovery of the seed values of the PHP system's PRNGs for password reset and so has the same goal as the attack in our own case study. Their attacks (which are against victims presenting a much smaller search space than ours, see (Argyros and Kiayias, 2012, Sec. 6.2)), however, require sending repeated requests to the victim server, which may take several minutes and may result in attack detection. In comparison, after a setup phase requiring a small brute-force attack (2^{20} offline trials), our attack requires at most four online queries to compromise a user account. It is thus almost instantaneous and scales easily to a large number of accounts.

6.4.1 Background on PRNG in PHP

The PHP runtime provides several functions by which applications can obtain or generate (pseudo)random numbers. For instance, during the process of password reset token generation, most PHP applications call APIs such as `microtime()`, `mt_rand()`, and `uniqid()`. Internally, the `microtime()` function calls `gettimeofday()` to obtain the current system time in the form of the number of seconds and microseconds since the Unix epoch (0:00:00 1 January 1970 UTC). The `mt_rand()` function, which is the interface to the PHP internal Mersenne Twister generator, automatically initializes its own internal state, if `mt_srand()` has not yet been invoked, with a random seed generated using functions `time()`, `php_combined_lcg()`, and `getpid()`. The `time()` function merely returns the number of seconds since the Unix epoch, and therefore has

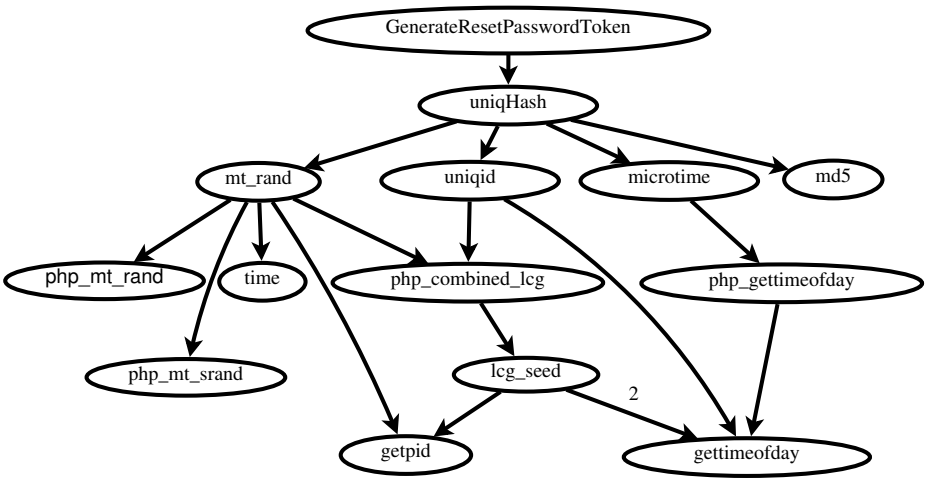


Figure 6.4: The call graph of password reset token generation in PHP applications.

low entropy. The function `php_combined_lcg()` combines two linear congruential generators (with prime periods $2^{31} - 85$ and $2^{31} - 249$) to generate a long-period pseudorandom sequence (the product of the primes). The initialization of `php_combined_lcg()` depends on the `lcg_seed()` function, which generates random seeds by calling `getpid()` once and `gettimeofday()` twice. These function calls and dependencies are shown in Figure 6.4.

While the range of options for seeding the PRNG in PHP systems may seem convoluted, as Figure 6.4 shows, the only sources of entropy for the PRNG seed are `gettimeofday()`, `time()`, and `getpid()`. By monitoring invocations of the `gettimeofday()` function, the adversary can immediately issue another call to `gettimeofday()` once it is called by the victim. As the adversary shares the OS with the victim web application, the result of the adversary's invocation of `gettimeofday()` will be very close to, if not exactly the same as, that returned to the victim application. The same is (even more) true of `time()`. As such, the only input to the victim PRNG that may be unknown to the adversary is the result of `getpid()`, which may assume any of 2^{16} values.

An adversary can initiate a password reset for its own account with the victim web application. As the adversary receives the corresponding secret string R, it can guess the `pid` and verify its correctness against R. Subsequent password reset attacks issued from the same connection will be served by the same process. The adversary therefore resolves virtually all entropy in the initial state of the PHP application; by continuously monitoring the invocations of `mt_rand()` and

`php_combined_lcg()`, the adversary can keep track of the evolvement of the PRNG and guess all the random numbers generated.

As described above, it is critical that the adversary monitors the initialization process of the PRNG, which takes place only once in the lifetime of a server process. A very common configuration (see `www.apache.org` and `www.php.net` for more information about PHP web server configurations) is to have one process, either an Apache process or a standalone PHP process, to serve each new request. As such, it is possible for an adversary to mount an active attack in which it triggers the PRNG initialization process for observation. To do so, the adversary can saturate existing server processes and force the victim application to instantiate new processes to serve subsequent requests.

6.4.2 Evaluation in Public PaaS

As in our previous attack, we experimented in DotCloud with Magento eCommerce application (latest version). Not only are e-commerce applications very popular, and Magento especially so as mentioned above, but they are likely targets because of the severity of the password resetting attacks against them. Our investigation of the source code of other web applications reveals that a few more widely used PHP applications are susceptible to such attacks as well, such as the latest version of WordPress (`http://www.wordpress.com`) that is reportedly used by 21.9% of the top 10 million websites.

By default, the Magento application launches two instances, a `www` instance and a `db` instance, running on separate machines. Particularly in this experiment, the parameters in `php-fpm.conf` were set so that the FastCGI Processes were created and terminated dynamically and only a small number of processes were kept when idle. We should note that this setting was not the default one in DotCloud—although it was in other public clouds we have explored. It was intended to make the attacks easier, as the requests are likely to be served by newly created `php-fpm` processes. Otherwise the attacker needs to find a way to crash the `php-fpm` process and force a restart.

Attack details and results. The strategy we employ is for the attacker to create enough "keep-alive" connections to force the creation of a new `php-fpm` process; then within the same connection, the attacker sends two password-reset requests—one request for an account under the attacker's control, another for the victim's account. As the first request results in email being sent to the account under

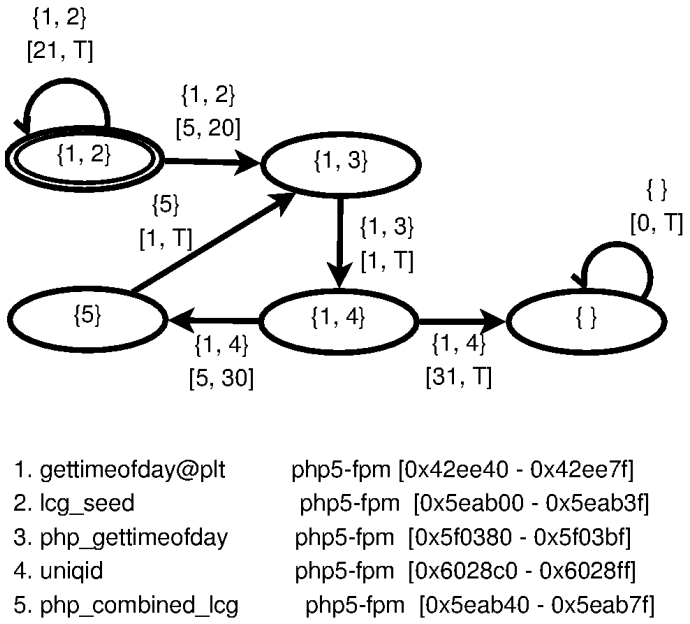


Figure 6.5: The attack NFA for case study in Section 6.4. Initial state q_0 indicated with double ovals, and error state $q_\perp$ not shown. T is the maximum FLUSH-RELOAD cycles before transitioning to $q_\perp$.

the attacker's control, the attacker can use the URL (and embedded secret R), together with the timing information collected from the side channel, to recover the `pid` of the new `php-fpm` process. Then the password reset token generated by the second request becomes entirely predictable. The attacker maintains a local copy of the PRNG modified to inject results collected from the side channel instead of those from real system calls.

As shown in Figure 6.5, five code chunks were monitored: one chunk from each of the three functions `php_gettimeofday()`, `lcg_seed()`, and `uniqid()`, which calls the entry point of `gettimeofday()` that is stored in the procedure linkage table (PLT); the first chunk of the function `php_combined_lcg()`; and the chunks that contain the entry point of `gettimeofday()` in the PLT. A complete execution path of the password reset action that initializes the PRNG in the PHP application is $2 \to 1 \to 2 \to 1 \to 3 \to 1 \to 3 \to 4 \to 1 \to 4 \to 5$ (indices as shown in Figure 6.5). The second password reset action follows the path $3 \to 1 \to 3 \to 4 \to 1 \to 4$. The attack NFA is shown in Figure 6.5.

In our experiments, the attacker and victim measurements of `gettimeofday()` sometimes differed by one bit; the response time (about 0.3μs) of the system call caused a single microsecond discrepancy. Thus, to recover the `pid` upon initialization of the PRNG the attacker needed to perform

a (trivial) offline brute-force guessing attack in a search space of size $2^{20} = 2^{16} \times 2^4$ (space 2^{16} for the `pid` and 2^4 for four invocations of `gettimeofday()`).

Once the attacker recovers the `pid`, a password-reset attack against a victim requires only two invocations of `gettimeofday()`, and thus, in our experiments, an online attack against a (tiny) space of size $4 = 2^2$. We emphasize that because the attacker is performing password reset and not password guessing, there is no account lockdown in response to an incorrect guess. So the attacker in our experiments could quickly guess the correct embedded secret R in the URL of the password link sent to the victim and then reset the victim's password.

6.5 Case Study 3: Breaking SAML-based Single Sign-On

In this final case study, we use our side-channel attack framework to instantiate a padding error oracle sufficient for mounting a Bleichenbacher attack (Bleichenbacher, 1998) against PKCS#1 v1.5 RSA encryption as used in XML. Bleichenbacher attacks allow the decryption of a target RSA ciphertext (although not key recovery). While this class of attacks has been known since 1998 and the insecurity of XML encryption in the face of a kind of Bleichenbacher attack was shown by Jager et al. (2012), implementations of PKCS#1 v1.5 persist in deployments and, instead of moving on to inherently more secure encryption, practitioners have deployed a sequence of countermeasures that prevent each attack. Current implementations are not exploitable by prior attacks, but our new attack circumvents all the existing countermeasures to (yet again) break XML encryption. We emphasize that the main takeaway is not that PKCS#1 v1.5 is inherently broken (as already known), but rather that our new side-channel attack framework and PaaS environments provide new opportunities for attackers.

6.5.1 Bleichenbacher Attacks

PKCS#1 specifies an algorithm for encryption using RSA. Recall that with RSA, one generates a key pair by choosing a modulus $N = pq$ for primes p, q and exponents e, d for which $ed \equiv 1 \bmod \phi(N)$; the public key is then (N, e) and secret key is (N, d). Let n be the length of N in bytes. With the PKCS#1 v1.5 padding scheme, one encrypts a message M of size m bytes with $m < n - 11$. Letting $r = n - m - 3$, a byte string P of length r is generated in which each byte

is randomly selected from $\{0,1\}^8 \setminus \{0\}$. Letting $X = 00 \,\|\, 02 \,\|\, P \,\|\, 00 \,\|\, M$, the ciphertext is then $C = X^e \bmod N$.

To decrypt, one computes $X = C^d \bmod N$ and then checks the padding. A padding error occurs if the first two bytes of X are not $00 \,\|\, 02$, there exists a 00 byte among the first 11 bytes, or there does not exist a 00 byte at all after the first two bytes. Decryption fails in such a case.

Bleichenbacher (1998) showed how to exploit decryption implementations that notify the sender of a ciphertext when a padding error occurs. Given a challenge ciphertext C^* encrypting some unknown message M, the attacker sends a sequence of adaptively chosen ciphertexts to the oracle, using the response to learn whether the padding is correct or not. Bleichenbacher attacks were first used against XML encryption by Jager et al. (2012), with improvements shortly after by Bardou et al. (2012). Below we use the latter's experimental results to estimate timings of the full attack.

Modern implementations attempt to prevent Bleichenbacher attacks by uniform error reporting, in which padding errors are not reported differently from other errors, and by ensuring that decryption runs in the essentially the same time when padding errors occur as when not. We will show, however, that our side-channel attack framework can be used in PaaS type settings to re-enable Bleichenbacher attacks despite such countermeasures.

6.5.2 Evaluation in Public PaaS

We demonstrate this attack in DotCloud. The target of the attack is an active open source project, SimpleSAMLphp, which implements a SAML-based authentication application in PHP that can be used as either a service provider or an identity provider. It is worth noting that recent SimpleSAMLphp implementations have provided defenses against the traditional Bleichenbacher attack by generating uniform error messages and eliminating timing differences due to invalid padding in session-key decryption. As we show in this section, however, these defenses do not prevent our attack. As we also explain, a recent change in SimpleSAMLphp to a better padding scheme (RSA-OAEP) does not prevent our attack either, as it is possible to force a rollback to PKCS #1 v1.5.

A set of protocol bindings (OASIS, b) and profiles (OASIS, c) are defined in the SAML 2.0 specification. We investigated the default protocol bindings implemented in SimpleSAMLphp for the web browser SSO profile. As shown in Figure 6.6, A web browser acting as a user agent interacts

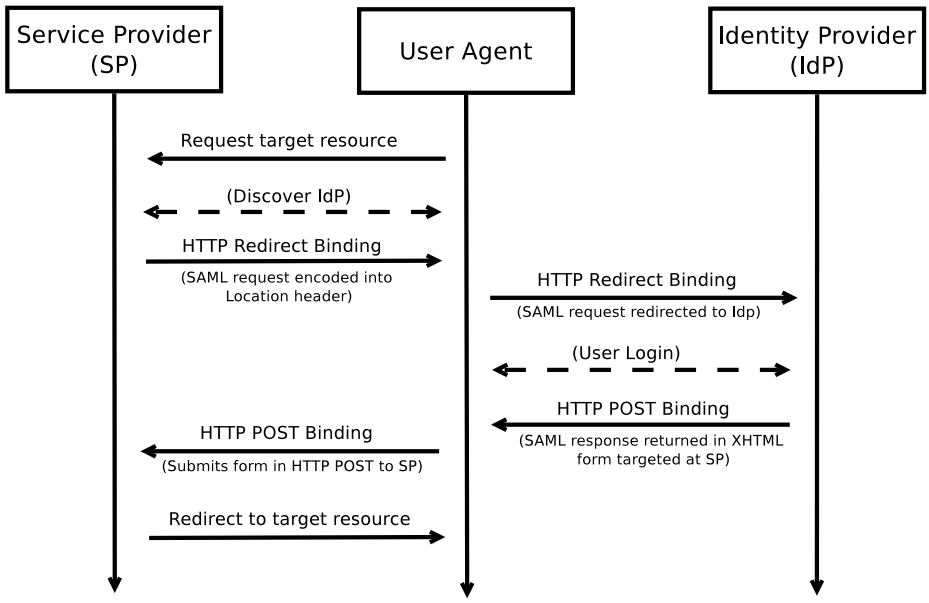


Figure 6.6: The SAML 2.0 protocol evaluated in the case study

with the service provider (SP) to access resources with the identify provider (IdP) for authentication. Upon receiving a resource access request, the SP issues an `<AuthnRequest>` message via HTTP redirect binding. The message in XML format is uncompressed and then base64-encoded in the redirect URL query string. After authenticating the user's identity, the IdP will return a SAML response message via HTTP POST binding, in which a signed and encrypted XML file is base64-encoded as a POST parameter which is then sent by the user agent to the service provider using the HTTP POST method.

The padding oracle. In the SAML 2.0 core specification (OASIS, a), XML encryption and signing work as follows. The message is first signed, and then encrypted under a symmetric session key. The session key is in turn encrypted. This means that the XML signature is only validated after performing the RSA decryption. While the default padding for encryption is RSA-OEAP, because the padding type is specified in the assertion itself, it is possible to modify the assertion and force the service provider to roll back to PKCS#1 v1.5 padding. The server generates an error whether or not the PKCS padding is correct, and timing channels have been eliminated. But we will now show how to use the side-channel attack to differentiate between code paths associated with padding errors and non-errors, enabling the mounting of Bleichenbacher-style attack.

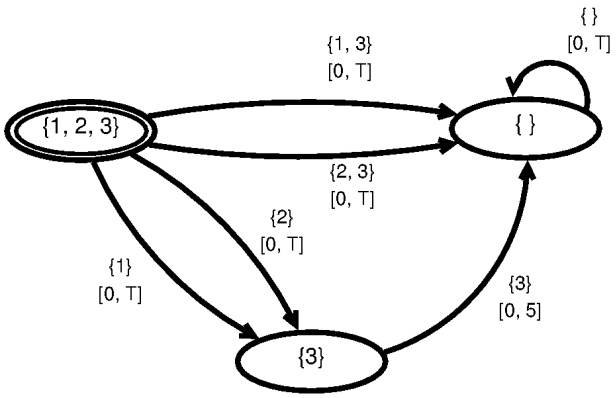
Attack details and results. The victim account operated a PHP application integrated with the latest stable version of SimpleSAMLphp. The PaaS environment ran OpenSSL version 0.9.8k and was invoked by the victim application. As such, the attacker monitored the library `libcrypto.so`, a component of OpenSSL, and specifically the function `RSA_padding_check_PKCS1_type_2()` that internally reports a padding error by calling `ERR_put_error()`. As the padding check procedure is only used during the RSA decryption, other operations do not invoke these functions and it is thus sufficient to monitor only the first chunk of each of the two functions. In practice, though, we found it helped to monitor the first two chunks of the function to increase the chance of capturing the events.

The attack NFA is shown in Figure 6.7. We continuously sent 10,000 requests with conformant padding and 10,000 requests with non-conformant padding, and report the rate of acceptance by the NFA. The results are shown in Table 6.3. The average time for making one request and padding error detection in this experiment was 0.544 seconds. Optimized attack software could achieve a much higher request rate.

The results indicate that we only had one-sided errors: an execution path accepted by the attack NFA correlated with a non-conformant padding with 100% accuracy. Therefore, the best strategy for the attacker is to send k requests to the padding oracle for each padding, and stop once an execution is accepted by the attack NFA and consider it to be non-conformant padding. If none of the k requests are accepted by the NFA, then no padding error occurred.

This approach will yield no false positives (i.e., false appearances of non-conformant padding). Given the error rate of 12% and assuming errors are independent of ciphertext values, the probability of a false negative (i.e., failure to observe non-conformant padding) in this procedure is $(.12)^k$.

Bardou et al. (2012) estimated that their modified Bleichenbacher attack against 2048-bit RSA keys could require around 335,065 queries (a likely overestimate in our view). We take $(.12)^k$ to be an upper bound on the probability of a false negative for non-conformant padding across k queries. Thus for all queries, an error bound is $335,065(.12)^k$; a choice of $k = 7$ yields an error probability of less than 1% for $335,065 \times 7 = 2,345,455$ total queries. This is about the same number of queries as the original Bleichenbacher attack, and significantly better than, for example, the same estimate for CBC-padding attack of Jager et al. (2012) that would require about 85 million queries.



1. RSA padding check PKCS1 type 2 libcrypto.so [0xb29c0 - 0xb29ff]
2. RSA padding check PKCS1 type 2 libcrypto.so [0xb2a00 - 0xb2a3f]
3. ERR put error libcrypto.so [0xcd800 - 0xcd83f]

Figure 6.7: The attack NFA for case study in Section 6.5. Initial state q_0 indicated with double ovals, and error state $q_\perp$ not shown. T is the maximum FLUSH-RELOAD cycles before transitioning to $q_\perp$.

		Attack NFA	
		Accepted	Rejected
Padding	Non-conformant	8800 (88%)	1200 (12%)
	Conformant	0 (0%)	10000 (100%)

Table 6.3: Confusion matrix for padding error detection. The attacker has only one-sided errors, 12% of the time failing to observe a padding error.

6.6 Discussion

6.6.1 Countermeasures

A key question for future research is how to design effective defenses against the attacks enabled within our proposed framework. High-overhead, general countermeasures to control-flow side channels, such as source-to-source translation (Coppens et al., 2009; Molnar et al., 2006) may be applicable in the PaaS setting, but have yet to see commercial deployment. The most general countermeasure for any side-channel attack is to prevent sharing of the exploited resource. In our setting, this would mean disallowing sharing of memory pages that serve as FLUSH-RELOAD attack vectors. An extreme realization would be a prohibition on sharing of any memory pages among different users, for instance by duplicating binary files for each user in the OS. Such a defense, however, would increase the memory footprint of each tenant, decreasing the number of tenants

that a PaaS provider could provision on a (virtual) machine and reducing machine utilization and service-provider profit. Selective memory sharing promises a more cost-effective approach; sharing of memory pages specifically carrying vulnerable code might then be disallowed. We leave the challenges of identifying, annotating, and protecting such code, as well as the development of alternative defenses, as interesting lines of future research.

6.6.2 Ethical Considerations

Our experiments involved running attacks against simulated victims using PaaS accounts setup for that purpose. The experiments discussed in Sections 6.2–6.5 were run on production PaaS platforms. This means our experiments were designed to conform with PaaS provider acceptable use, the law, and proper ethics.

Our attacks only targeted tenants running accounts that we setup and controlled, and no information about other customers was ever collected in our experiments. Our attacker instances did not conduct FLUSH-RELOAD attacks indiscriminately, but rather these were carefully timed to coincide with requests that we initiated to our victim instances. In this way, we limited the risk of our attacker instances observing activities of tenants other than our own.

It is possible that another tenant's programs made use of the same shared executable as our attacker and victim, in which case there is a concern that other tenants might experience degraded memory hierarchy performance as compared to running while co-located with different tenants. Moreover, the acceptable use policies of the clouds on which we demonstrated our attacks include general requirements that we not interfere with other users' enjoyment of their services, which could be interpreted to preclude our demonstrations if they slowed down other tenants substantially as a side effect. We therefore designed our experiments so that they do not cause *undue harm* and, specifically, do not degrade performance of such bystanders significantly more than their performance could be degraded by other workloads.

To ensure no undue harm, we ran local microbenchmarks to evaluate the possible overhead observed by a bystander due to our attacks. For example, to gain confidence that the attack of Section 6.4 would introduce minimal overhead on a bystander, in one container we constructed an attacker that, in each FLUSH-RELOAD cycle, monitored *every* chunk monitored in *any* state of the attack NFA of Figure 6.5. The "bystander" in another container ran a web server hosting a dynamic

web page that was artificially constructed to touch (i.e., execute some instruction in) *every* chunk monitored by the attacker before returning. We forced the attacker and the bystander to share the last level cache in all experiments.

We configured a separate machine in the same LAN to represent a client that repeatedly issued HTTP requests (in the same HTTP session) to the dynamic web page served by the bystander. To measure the bystander's performance degradation resulting from the attacker's activity, we instrumented the client with `httperf` and `apachebench`. In the absence of the attacker, the client received responses with an average latency of .306ms, and the throughput of the bystander was 461 requests per second. With the attacker active, the results were nearly identical: an average latency of .307ms and, again, 461 requests per second. Given the conservative nature of these experiments (with the attacker monitoring more chunks than in the actual attack, and the bystander touching all of them per request), we concluded that our attack demonstrations posed negligible risk to bystanders.

An interesting side observation from these experiments is that the minimal performance degradation induced by the attacks in this chapter offers little hope for detecting these attacks. That is, prior studies suggest that cross-tenant side-channel attacks in cloud settings can induce significant performance degradation in victim workloads, as was the case in, e.g., the attacks demonstrated by Zhang et al. (2012). It is possible, therefore, that the attacks of Zhang et al. (2012) might be detected by monitoring the performance of the victim application. While our results here do not conclusively rule out the use of victim performance monitoring to detect the attacks in this chapter, they also do not offer much promise for doing so.

6.7 Summary

We have proposed a general automaton-driven framework to mount cache-based side-channel attacks and demonstrated its potency specifically in PaaS environments. Our three case studies demonstrate that an attacker co-located with a victim can learn sensitive user data, such as number of distinct items in a shopping cart; perform password-reset attacks against arbitrary users; and break XML encryption in a SAML-based authentication application. We believe that our work presents: (1) the first exploration of cache-based side-channel attacks specifically in PaaS environments, and (2)

the first report of granular, cross-tenant, side-channel attacks successfully mounted in any existing commercial cloud, PaaS or otherwise, against state-of-the-art applications.

The attacks we illustrate are especially significant in some cases in that they bypass existing or proposed side-channel countermeasures. Our shopping-cart attack is immune to defenses proposed for analogous, timing-based side-channel attacks. Our study of RSA private-key decryption re-enables the classic Bleichenbacher padding-oracle attack despite widely deployed countermeasures against remote adversaries. These attacks call into question the traditional security model in developing software applications in hosted, shared environments, and open up new research questions in cloud computing security.

CHAPTER 7: CONCLUSION

As a cost-effective computing model, cloud computing is here to stay. The growth of clouds, however, depends increasingly on the security promises that the providers can deliver to their customers. This dissertation explores one particular type of cloud security problems—side-channel threats—and concludes that *the design of, and common practices in, modern public multi-tenant IaaS and PaaS clouds are flawed in their insufficient isolation of cloud tenants from side-channel attacks by co-located tenants*, which is evidenced by the demonstration of (1) a granular cross-VM side channel that exploits a PRIME-PROBE protocol on the L1 instruction cache to extract cryptographic keys from another VM; and (2) an NFA-based framework that exploits FLUSH-RELOAD side channels to learn the execution path of a co-located PaaS application in public PaaS cloud offerings, thereby exfiltrating sensitive information to subvert the security of the victim applications. However, we stress that *in IaaS clouds, tenants can defend against prominent side-channel attacks by themselves with modified operating system kernels*. In supporting of such statement, we designed, implemented and evaluated two defensive mechanisms—HomeAlone and Düppel—to address the side-channel threats, and showed that an IaaS tenant can conduct effective self-defense against side channels without the help of cloud providers.

We hope the work presented in this dissertation will have impact on both industry practices and academic research. We believe the demonstrated insufficiency in isolation of cloud tenants poses essential risks to the success of cloud businesses. Therefore, we anticipate the public cloud providers who will continue to provide multi-tenant services will take actions to this new security threats. The commercial adoption of HomeAlone and Düppel are yet to be seen, but we believe that security practices for isolating tenants in public clouds will change in the near future, with or without them. Our work also paves the way for several research directions. For instance, by demonstrating, for the first time, cross-VM side-channel attacks in IaaS settings and cross-tenant side-channel attacks in public PaaS clouds, our work helps the research community better understand the risks and attack

vectors of side channels in shared cloud servers, and by that means serve, at least in part, as the basis of future research that defends against such threats. Moreover, many of the techniques in this dissertation that enable the attacks or the defenses are innovative (e.g., the usage of side channels as defensive tools in HomeAlone), which, we hope, will inspire fellow security researchers to find solutions to their research problems.

There are several research questions yet to be answered in this dissertation, however. First, the quest for a clean-slate, efficient solution to the cross-VM side-channel threats in public IaaS clouds will have to continue. HomeAlone and Düppel are intended to mitigate the threats only in certain circumstances (i.e., HomeAlone verifies the status of physical isolation and Düppel conducts self-defense for cache-based side-channels only). Second, given the demonstrated security attacks in Chapter 6, the cloud industry would need to better understand the extent to which our technique is applicable, and, more importantly, a practical solution to such issues. We leave these questions to be addressed in future research.

BIBLIOGRAPHY

Aciiçmez, O. (2007). Yet another microarchitectural attack: exploiting I-Cache. In *2007 ACM Workshop on Computer Security Architecture*, pages 11–18.

Aciiçmez, O., Brumley, B. B., and Grabher, P. (2010). New results on instruction cache attacks. In *12th International Conference on Cryptographic Hardware and Embedded Systems*, pages 110–124.

Aciiçmez, O., Ç. K. Koç, and Seifert, J.-P. (2007a). Predicting secret keys via branch prediction. In *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference*, pages 225–242.

Aciiçmez, O., Gueron, S., and Seifert, J.-P. (2007b). New branch prediction vulnerabilities in openSSL and necessary software countermeasures. In *11th IMA International Conference on Cryptography and Coding*, pages 185–203.

Aciiçmez, O. and Koç, c. K. (2006). Trace-driven cache attacks on AES. In *8th International Conference on Information and Communications Security*, pages 112–121.

Aciiçmez, O., Koç, c. K., and Seifert, J.-P. (2007c). On the power of simple branch prediction analysis. In *2nd ACM Symposium on Information, Computer and Communications Security*, pages 312–320.

Aciiçmez, O., Schindler, W., and Koç, c. K. (2005). Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *12th ACM Conference on Computer and Communications security*, pages 139–146.

Aciiçmez, O., Schindler, W., and Koç, c. K. (2006). Cache based remote timing attack on the AES. In *7th Cryptographers' Track at the RSA Conference*, pages 271–286.

Aciiçmez, O. and Seifert, J.-P. (2007). Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91.

Akkar, M.-L., Bevan, R., Dischamp, P., and Moyart, D. (2000). Power analysis, what is now possible... In *Advances in Cryptology – ASIACRYPT 2000*, pages 489–502.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19.

AmazonAWS (2010). AWS case study: Numerate finds a winning combination with AWS. `http://bit.ly/a0ONsQ`.

Argyros, B. and Kiayias, A. (2012). I forgot your password: Randomness attacks against PHP applications. In *21st USENIX Security Symposium*.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Aviram, A., Hu, S., Ford, B., and Gummadi, R. (2010). Determinating timing channels in compute clouds. In *2010 ACM Workshop on Cloud Computing Security*, pages 103–108.

Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., and Tsay, J.-K. (2012). Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology — CRYPTO 2012*, pages 608–625.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171.

Bernstein, D. J. (2005). Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., and Palermo, G. (2005). AES power attack based on induced cache miss and countermeasure. In *International Conference on Information Technology: Coding and Computing*, pages 586–591.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81.

Bienia, C. and Li, K. (2009). PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *5th Workshop on Modeling, Benchmarking and Simulation*.

Bishop, C. M. (2007). *Pattern Recognition and Machine Learning*. Springer.

Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology — CRYPTO '98*, pages 1–12.

Blömer, J. and Seifert, J.-P. (2003). Fault based cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography, 7th International Conference*, pages 162–181.

Bolosky, W. J. and Scott, M. L. (1993). False sharing and its effect on shared memory performance. In *USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*.

Bonneau, J. and Mironov, I. (2006). Cache-collision timing attacks against AES. In *8th International Conference on Cryptographic Hardware and Embedded Systems*, pages 201–215.

Bortz, A. and Boneh, D. (2007). Exposing private information by timing web applications. In *16th International Conference on World Wide Web*, pages 621–628.

Bovet, D. and Cesati, M. (2005). *Understanding The Linux Kernel*. Oreilly & Associates.

Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. In *12th USENIX Security Symposium*.

Bugnion, E., Anderson, J. M., Mowry, T. C., Rosenblum, M., and Lam, M. S. (1996). Compiler-directed page coloring for multiprocessors. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255.

Callas, J., Donnerhacke, L., Finney, H., and Thayer, R. (1998). OpenPGP message format. Technical report, RFC 2440, November.

Carlson, T. (24 February 2010). Secure cloud offerings for government. MSDN blog, `http://bit.ly/b2e1sI`.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3).

Chen, S. X. and Liu, J. S. (1997). Statistical applications to the Poisson-binomial and conditional Bernoulli distributions. *Statistica Sinica*, 7:875–892.

Chisnall, D. (2007). *The Definitive Guide to the Xen Hypervisor*. Prentice Hall PTR.

Coppens, B., Verbauwhede, I., Bosschere, K. D., and Sutter, B. D. (2009). Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*, pages 45–60.

Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. (2012). Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4).

Dong, Y., Li, S., Mallick, A., Nakajima, J., Tian, K., Xu, X., Yang, F., and Yu, W. (2006). Extending Xen with Intel virtualization technology. *Intel Technology Journal*, 10.

ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4).

England, P. and Manferdelli, J. (2006). Virtual machines for enterprise desktop security. *Inf. Secur. Tech. Rep.*, 11(4):193–202.

Esser, S. (2008). Lesser known security problems in PHP applications. In *Zend Conference*.

Felten, E. W. and Schneider, M. A. (2000). Timing attacks on web privacy. In *7th ACM Conference on Computer and Communications Security*, pages 25–32.

Gandolfi, K., Mourtel, C., and Olivier, F. (2001). Electromagnetic analysis: Concrete results. In *3rd International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261.

Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. (2003). Terra: a virtual machine-based platform for trusted computing. In *19th ACM Symposium on Operating Systems Principles*, pages 193–206.

Gautham, N. (2006). *Bioinformatics: Databases and Algorithms*. Alpha Science International Ltd.

Gullasch, D., Bangerter, E., and Krenn, S. (2011). Cache games – bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505.

Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205.

Hunter, J. S. (1986). The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210.

Jager, T., Schinzel, S., and Somorovsky, J. (2012). Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML encryption. In *Computer Security — ESORICS 2012*, pages 752–769.

Jana, S. and Shmatikov, V. (2012). Memento: Learning secrets from process footprints. In *33nd IEEE Symposium on Security & Privacy*, pages 143–157.

Kamkar, S. (2010). phpwn: Attacking sessions and pseudo-random numbers in PHP. In *Blackhat USA*.

Keramidas, G., Antonopoulos, A., Serpanos, D., and Kaxiras, S. (2008). Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12:221–230.

Kim, T., Peinado, M., and Mainar-Ruiz, G. (2012). STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*.

Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – Crypto'96*, pages 104–113.

Kocher, P. C., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Crypology – CRYPTO '99*, pages 388–397.

Könighofer, R. (2008). A fast and cache-timing resistant implementation of the AES. In *2008 Cryptographers' Track at the RSA Conference*, pages 187–202.

Lauradoux, C. (2005). Collision attacks on processors with cache and countermeasures. `http://perso.citi.insa-lyon.fr/claurado/publis/Lau05b.pdf`.

Li, P., Gao, D., and Reiter, M. K. (2013). Mitigating access-driven timing channels in clouds using StopWatch. In *43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12.

Lynch, W. L., Bray, B. K., and Flynn, M. J. (1992). The effect of page allocation on caches. In *25th International Symposium on Microarchitecture*, pages 222–225.

Mahowald, R. P., Olofson, C. W., Ballou, M.-C., Fleming, M., and Hilwa, A. (2013). Worldwide competitive public Platform as a Service 2013-2017 forecast (Doc 243315). IDC Inc.

Marshall, A., Howard, M., Bugher, G., and Harden, B. (2010). Security best practices for developing Windows Azure applications.

Martin, R., Demme, J., and Sethumadhavan, S. (2012). Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture*, pages 118–129.

Meushaw, R. and Simard, D. (2000). A network on a desktop. *NSA Tech Trend Notes*, 9(4). `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

Molnar, D., Piotrowski, M., Schultz, D., and Wagner, D. (2006). The program counter security model: automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology*, pages 156–168.

Montgomery, P. L. (1987). Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp*, 48(177):243–264.

Mosberger, D. and Jin, T. (1998). httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37.

Mowery, K., Keelveedhi, S., and Shacham, H. (2012). Are AES x86 cache timing attacks still feasible? In *2012 ACM Cloud Computing Security Workshop*, pages 19–24.

Nagami, Y., Miyamoto, D., Hazeyama, H., and Kadobayashi, Y. (2008). An independent evaluation of web timing attack and its countermeasure. In *3rd International Conference on Availability, Reliability and Security*, pages 1319–1324.

Natis, Y. V. (2014). Gartner research highlights Platform as a Service (ID: G00259659). Gartner Inc.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453.

Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. (2006). Intel virtualization technology: Hardware support for for efficient processor virtualization. *Intel Technology Journal*, 10.

Neve, M. and Seifert, J.-P. (2007). Advances on access-driven cache attacks on AES. In *13th International Conference on Selected Areas in Cryptography*, pages 147–162.

NIST (2012). Challenging security requirements for US government cloud computing adoption. NIST Manuscript Publication.

OASIS. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.

OASIS. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf`.

OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`.

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of AES. In *2006 Cryptographers' Track at the RSA Conference*, pages 1–20.

Owens, R. and Wang, W. (2011). Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Conference on Performance, Computing and Communications*, pages 1–8.

Page, D. (2005). Partitioned cache architecture as a side-channel defence mechanism.

Percival, C. (2005). Cache missing for fun and profit. In *BSDCon 2005*.

Piotrowski, M. and Joseph, A. D. (2010). Virtics: A system for privilege separation of legacy desktop applications. Technical Report EECS-2010-70, U.C. Berkeley.

Qian, Z., Mao, Z. M., and Xie, Y. (2012). Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *2012 ACM Conference on Computer and Communications Security*, pages 593–604.

Raj, H., Nathuji, R., Singh, A., and England, P. (2009). Resource management for isolation enhanced cloud services. In *ACM Cloud Computing Security Workshop*, pages 77–84.

Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *16th ACM Conference on Computer and Communications Security*, pages 199–212.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2).

Rutkowska, J. and Wojtczuk, R. (2012). Qubes OS architecture. `http://qubes-os.org`.

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

Shriver, R. (2014). Survey: enterprise development in the cloud. `http://research.gigaom.com/report/survey-enterprise-development-in-the-cloud/`.

Stone, B. and Vance, A. (2010). Companies slowly join cloud-computing. *New York Times*.

Suzaki, K., Iijima, K., Yagi, T., and Artho, C. (2011). Memory deduplication as a threat to the guest OS. In *4th European Workshop on System Security*.

TrendMicro (2011). Cloud security survey global executive summary.

Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71.

Tsafrir, D., Etsion, Y., and Feitelson, D. G. (2007). Secretly monopolizing the CPU without superuser privileges. In *16th USENIX Security Symposium*, pages 1–18.

Vattikonda, B. C., Das, S., and Shacham, H. (2011). Eliminating fine grained timers in Xen. In *3rd ACM Workshop on Cloud Computing Security*, pages 41–46.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, IT-13:260–269.

W3Techs (2014). Usage of content management systems for websites. `http://w3techs.com/technologies/overview/content_management/all`.

Wang, Z. and Lee, R. B. (2006). Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference*, pages 473–482.

Wang, Z. and Lee, R. B. (2007). New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture*, pages 494–505.

Wang, Z. and Lee, R. B. (2008). A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93.

Weiß, M., Heinz, B., and Stumpf, F. (2012). A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*.

Yarom, Y. and Falkner, K. (2013). Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. `http://eprint.iacr.org/2013/448`.

Yeboah-Boateng, E. O. and Essandoh, K. A. (2014). Factors influencing the adoption of cloud computing by small and medium enterprises in developing economies. *International Journal of Emerging Science and Engineering*, 2.

Zhang, K. and Wang, X. (2009). Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *18th USENIX Security Symposium*, pages 17–32.

Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. (2011). HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328.

Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2012). Cross-VM side channels and their use to extract private keys. In *2012 ACM Conference on Computer and Communications Security*, pages 305–316.

Zhang, Y. and Reiter, M. K. (2013). Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *2013 ACM Conference on Computer and Communications Security*, pages 827–838.

Zhou, F., Goel, M., Desnoyers, P., and Sundaram, R. (2011). Scheduler vulnerabilities and coordinated attacks in cloud computing. In *IEEE International Symposium on Networking Computing and Applications*, pages 533–559.