2011 IEEE Symposium on Security and Privacy

HomeAlone: Co-Residency Detection in the Cloud via Side-Channel Analysis

Yinqian Zhang
University of North Carolina
Chapel Hill, NC, USA
zhang@unc.edu

Ari Juels
RSA Laboratories
Cambridge, MA, USA
ajuels@rsa.com

Alina Oprea
RSA Laboratories
Cambridge, MA, USA
aoprea@rsa.com

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

***Abstract*—Security is a major barrier to enterprise adoption of cloud computing. *Physical co-residency* with other tenants poses a particular risk, due to pervasive virtualization in the cloud. Recent research has shown how side channels in shared hardware may enable attackers to exfiltrate sensitive data across virtual machines (VMs). In view of such risks, cloud providers may promise physically isolated resources to select tenants, but a challenge remains: Tenants still need to be able to *verify* physical isolation of their VMs.**

We introduce *HomeAlone*, a system that lets a tenant verify its VMs' exclusive use of a physical machine. The key idea in HomeAlone is to invert the usual application of side channels. Rather than exploiting a side channel as a vector of attack, HomeAlone uses a side-channel (in the L2 memory cache) as a novel, defensive detection tool. By analyzing cache usage during periods in which "friendly" VMs coordinate to avoid portions of the cache, a tenant using HomeAlone can detect the activity of a co-resident "foe" VM. Key technical contributions of HomeAlone include classification techniques to analyze cache usage and guest operating system kernel modifications that minimize the performance impact of friendly VMs sidestepping monitored cache portions. Our implementation of HomeAlone on Xen-PVM requires no modification of existing hypervisors and no special action or cooperation by a cloud provider.

***Keywords*-Cloud computing, Infrastructure-as-a-Service (IaaS), co-residency detection, side-channel analysis**

I. 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. Security, however, is a widely cited impediment to enterprise adoption of public clouds, i.e., clouds administered by third parties [45]. By relinquishing control over their IT resources, cloud tenants expose themselves to the security choices—and mistakes—of their providers. Because many tenants share common pools of hardware, the cloud makes strange bedfellows. Businesses may find themselves sharing adjacent or overlapping computing resources with partners, suppliers, competitors, or attackers.

Strong isolation among tenants is therefore 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, other potential sources of information leakage often remain. Virtual Machines (VMs) 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*, i.e., data leakage through implicit channels. Recent research has demonstrated how hostile VMs can potentially extract sensitive data, such as passwords and cryptographic keys, from other VMs resident on the same physical machine by using memory caches (L2) as side channels [38].

For such reasons, 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 [39]. (Ultimately, 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 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.

A. HomeAlone

In this paper we 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. Our implementation of HomeAlone for the Xen paravirtualization (PVM) architecture 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 L2 memory 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.

B. Technical Challenges and Contributions of HomeAlone

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 L2 cache in a virtualized environment is never entirely quiet, and measurement of its activity (via techniques described below) 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 our classifier 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 operating system (OS) of the friendly VMs. By taking advantage of the double indirection layer in memory virtualization in Xen-PVM, 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 L2 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 L2 cache as an avenue for attack. So, while the L2 cache has been demonstrated to be an easily exploitable side-channel attack vector, it is one whose abuse HomeAlone is well-positioned to detect.

Paper organization: In Section II, we describe the cloud scenarios envisaged for use of HomeAlone and the accompanying threat model. We explain the characteristics of the L2-cache as a side channel in Section III. We detail the design of HomeAlone in Section IV and its implementation in Section V. In Section VI, we evaluate the detection accuracy of HomeAlone on demonstration workloads and the performance impact of HomeAlone. We discuss in Section VII a number of issues that bear on the use of HomeAlone in practice. Section VIII reviews related work. We conclude in Section IX.

II. Background and System Model

A. Cloud Infrastructure Service

Cloud services are often taxonymized—based on the abstraction layer they export—as Platform as a Service (PaaS), Software as a Service (SaaS), or Infrastructure as a Service (IaaS). PaaS offers an application-development environment but abstracts away lower software layers such as the OS. SaaS presents an application-level interface to the tenant. Our focus in this paper is on IaaS.

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.

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/or managed by the tenant itself.

The security concerns surrounding cloud computing arise primarily in public clouds. (They carry over, however, 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 [38].

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 demand physical isolation as part of their SLAs [18]. Others use only resource instances that are meant to provide such isolation, such as full-physical-machine instances with Amazon Web Services [11].

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 remediate 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.

B. 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., [38].) 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.[1] We consider two scenarios: (1) The "foe" VM is benign, i.e., oblivious to its co-residency, or at least not attempting to 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 [21]. 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.

C. 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.

[1]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.

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.

III. Cache-based Side Channels in Virtualized Infrastructures

A. Caches in Modern Architectures

Modern processors benefit from multi-level caches to reduce latencies incurred by accesses to main memory. 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 amortize 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 denoted *cache lines*, with fixed size between 8 and 512 bytes. Typical caches are *set-associative*. A w-way set-associative cache is partitioned into m sets, each with w lines. Each block 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 a memory block has a unique location in the cache (corresponding to $w = 1$), and *fully associative* caches in which a memory block can be mapped to any location in the cache (corresponding to $m = 1$). Increasing the degree of associativity usually decreases the cache miss rate, but it increases the cost of searching a block in the cache.

B. Cache-based Timing Channel

Cache-based timing channels have been widely studied in various contexts (see Section VIII). 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.

In our study, we consider the cache-based timing channel constructed by measuring the cache load of a monitored entity V that shares a common cache with the monitoring entity U. A basic construction of such a timing channel is what we call the PRIME-PROBE protocol:

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 this paper we focus on last-level caches on x86 platforms; for our experimental platforms, these are L2 caches. There are several hardware features that impact the cache-based timing channel we consider.

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. With the hardware-based TLBs implemented by the x86 architecture, TLB misses are expensive (as high as 100 cycles). 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. Thus, the TLB can add significant noise to the timing measurements of the PROBE phase.

Hardware prefetching: Modern CPUs implement several optimizations: they reorder accesses to the cache and 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 [38].

Collisions from the instruction cache: The L2 cache is shared by data and instructions, and thus data blocks can be evicted in favor of instructions. Instruction caching generally has a small effect on the timing measurements [35].

Multi-core architectures: On multi-core hosts, 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.

Simultaneous multithreading (SMT): CPUs supporting SMT allow multiple threads to execute simultaneously on the same CPU core and share the same cache hierarchy. This feature enables the monitoring entity to execute while the monitored entity is running and to detect the activity of the monitored entity with a finer time resolution.

C. Implications of Virtualization

Virtualization has been widely adopted in cloud computing for its flexibility, elasticity and ease of management. While virtualization provides logical isolation of virtual machines running on the same physical machine, it has been shown that the cache-based timing channels are still viable in virtualized environments [38], [36], albeit with reduced bandwidth. In this section we discuss several implications of the Xen paravirtualization (PVM) architecture to the cache timing channel, as our implementation of HomeAlone is based on Xen-PVM. Many of these features are common to hardware-assisted virtual machine (HVM) functionality as supported by the Intel VT-x/VT-i [32] and AMD-V technologies, as well.

Background activity: 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, even when no foe VM is present.

Processor virtualization: Virtual machines time-share a physical machine and they run as scheduled by the hypervisor scheduler. Xen implements multiple scheduling algorithms (e.g., BVT, the credit scheduler). The default credit scheduler uses a fixed 30ms time slice to allocate VMs. Whether a monitoring VM U can observe the cache activity of another, targeted VM V through the PRIME-PROBE protocol depends on being scheduled on a core that shares a cache with V and on both U and V remaining there for sufficiently long.

I/O virtualization: In a virtualized environment, a guest OS does not have direct control of I/O devices. In the Xen architecture, `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 in a modification of cache access patterns.

IV. Designing a Co-Residency Detector

The PRIME-PROBE timing channel described in Section III-B 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 IV-A, 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 IV-B and discuss training this multi-probe classifier in Section IV-C. We perform a cursory evaluation of the classifier’s detection capability in Section IV-D; this evaluation will be augmented with additional evaluations in Section VI.

Much of our discussion in this section is informed by experiences with the platform on which we performed the experiments reported in this paper. 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 $l = 64\text{B}$, 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.

A. 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 V.) 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 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.

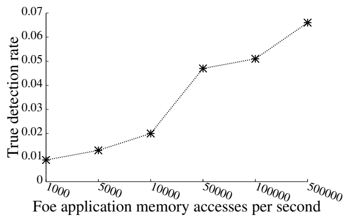


Figure 1. True detection rate of the single-probe classifier when false detection rate is configured to $\alpha = 1\%$. Four friendly VMs and one foe VM. All VMs unpinned. (Section IV-A)

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 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 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 V), it appears to be impossible to control the cache activity of Dom0 from within a VM.

B. A Multi-Probe Classifier

In light of the difficulties in interpreting the results of a single PRIME-PROBE trial discussed in Section IV-A, 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 V, 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.

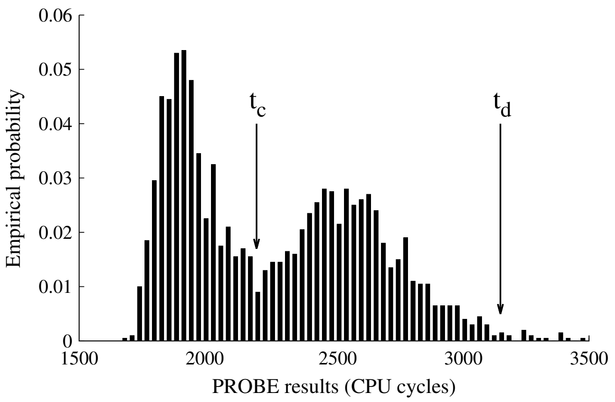


Figure 2. Distribution of timing results of PRIME-PROBE trials with no foe present.

As a first step towards our goal of detecting a foe VM based on the results of PRIME-PROBE trials, we plot in Figure 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 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 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 IV-C.

The monitoring VM collects this information F after each PRIME-PROBE trial. (We will discuss how in Section V.) 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 IV-C) 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., [19]) 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 IV-C). 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.

C. 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 paper 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., [38]).

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 paper, 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.

D. 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 IV-A. Our results here are not intended to be exhaustive; we will consider detection in the context of additional workloads in Section VI.

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 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 IV-A. Figure 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 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.

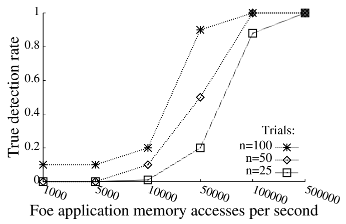


Figure 3. True detection rate of multi-probe detector of Section IV-B with four friendly VMs ($\alpha = 1\%$).

V. IMPLEMENTATION

In order to execute the detection technique described in Section IV, 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.

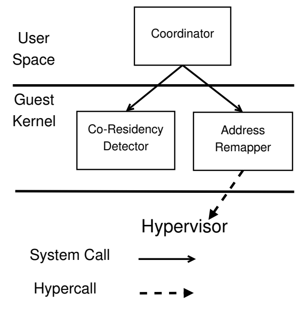


Figure 4. Architecture of our implementation within one guest VM

Our implementation consists of a suite of tools that is installed within each friendly VM. As shown in Figure 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*.

A. 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 [38]).

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 V-B), 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 IV-B) 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 IV.

B. 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., [31], [17]) 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 [20].

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 VI.

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; l 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 l-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/l blocks of the page with index i map to cache set indices $\{i(p/l) \bmod m, \ldots, (i+1)(p/l) - 1 \bmod m\}$, the most granular way of coloring cache sets is to have one color correspond to cache sets with indices in $\{i(p/l) \bmod m, \ldots, (i{+}1)(p/l){-}1 \bmod m\}$ for a given $i \in \{0, \ldots, \frac{m}{p/l} - 1\}$. Since $m = c/(w \times l)$, the number k of colors that our implementation can support is

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

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

C. 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.

VI. 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 IV.

The applications that we employ in our VMs are taken from the PARSEC benchmarks [12], [13]. 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.

1) `blackscholes`: This benchmark simulates financial analysis and, in particular, calculates the prices of a portfolio of options using Black-Scholes partial differential equations.
2) `bodytrack`: This computer vision application tracks a 3D pose of human bodies and represents video surveillance and character animation applications.
3) `canneal`: This is a benchmark using cache-aware simulated annealing to design chips that minimize routing costs; it is representative of engineering applications.
4) `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.
5) `facesim`: This benchmark simulates human faces and is representative of applications like computer games that employ physical simulation to create virtual environments.
6) `streamcluster`: This benchmark was developed for solving online clustering problems and is included for its representation of data mining algorithms.
7) `x264`: This is an H.264/AVC video encoder that can be used to simulate next-generation video systems.

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 IV-A.

A. 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 IV-C 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 IV-D 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,

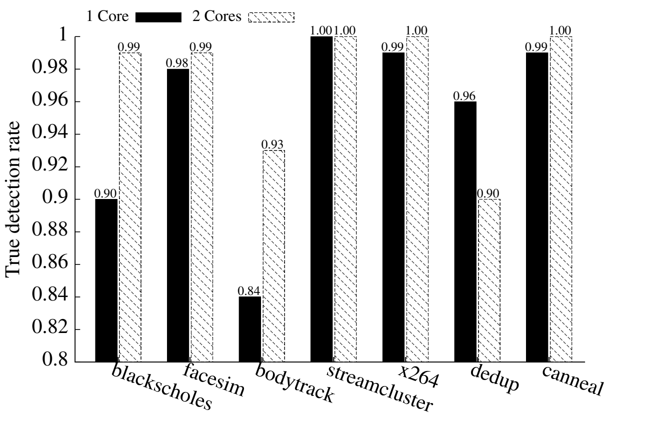
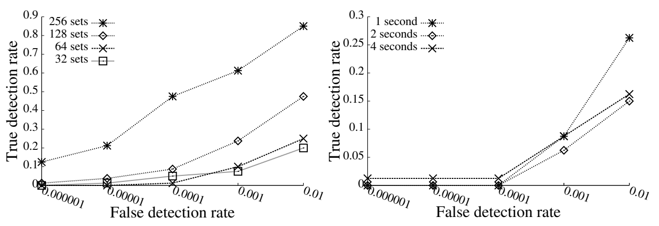


Figure 5. True detection rates for different foe applications ($n = 25$, $\alpha = 1\%$)



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

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 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 II-B. 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 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.

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

B. 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 VII.) However, changing the cache color induces performance overhead caused by the address remapping procedure (see Section V-B). In Figure 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 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.

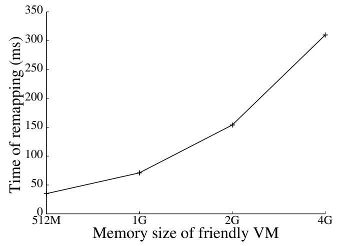


Figure 7. Runtime of remapping $1/16^{th}$ of memory for one VM, as function of memory size. Note that x-axis is log-scale.

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 V). 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.[2] 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

[2]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.

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 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 are multifold: 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.

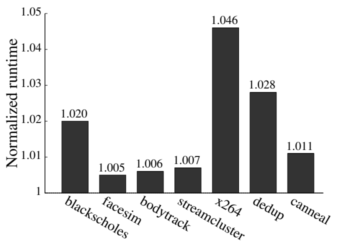


Figure 8. Normalized performance of benchmark applications during detection periods

VII. 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 V-A, 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 [32], [22]. Although major computing infrastructure providers like Amazon [1] and Rackspace [3] 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 [22]. 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 performance. Examples include hypercalls that allow guests direct control of device drivers [2], 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 [4], the branch target cache [6], [7], or shared functional units [46], [9]—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.

VIII. Related Work

Most prior work on cache timing channels has focused on their use as a side or covert channel. Here we briefly review related research and highlight its differences from our own.

Cryptanalytic techniques based on timing measurements of arithmetic operations were introduced by Kocher [27]. Subsequently, timing attacks based on shared data caches have been widely studied in the cryptanalysis of cryptographic protocols, e.g., [43], [42], [35], [8], [5], [15], [33], [34], [44], [23], [40], [25], [30], [16], [37], [49], [14], [24], [41]. The focus of this (still active) research area is to exploit the characteristics of the data cache (in particular the access latency gap between the cache and main memory) to develop cryptanalytic techniques specifically tailored to particular cryptographic implementations. In contrast, our work uses timing measurements on the L2 cache as a defensive tool. Moreover, our techniques are general in that we aim at detecting arbitrary foe VMs and we do not assume any knowledge about the foe VM implementation or workload.

Methods proposed to mitigate the threats posed by data-cache side channels generally fall into three categories. First, they include new cache designs (e.g., [46], [47], [26], [28], [48]). Second, Aviram et al. [10] have proposed to eliminate timing channels in cloud computing by forcing VM execution to be deterministic, but the success of this approach still needs to be demonstrated. Third, a promising direction is to construct cryptographic implementations that resist cache-based timing attacks (e.g., [29], [25]). Techniques such as new cache designs and forced determinism could potentially hinder the detection capability that we have developed in this paper. However, we do not anticipate that these mechanisms will be widely adopted in the near future. Another defense applicable in cloud computing is to disallow cache sharing among tenants altogether, either by grouping friendly VMs on cores sharing a cache or by partitioning the cache among VMs [36]. Of course, our techniques enable friendly VMs to detect if the service provider fails to correctly implement such cache isolation policies.

Besides the data cache, other architectural side channels have been exploited in cryptanalysis; as mentioned above, these include the instruction cache [4], the branch target cache [6], [7], and shared functional units [46], [9]. It is conceivable that these or other side channels could be used for foe detection, though we leave investigation of this possibility to future work.

IX. Conclusion

With the growing movement of sensitive applications to clouds, there is increasing demand for physical isolation of tenants' workloads (e.g., [18], [11]). In this paper 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 (see Section VIII).

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.

Acknowledgements

We thank Darrell Bethea, Steve Gribble and the anonymous reviewers for suggestions that led to improvements to this paper. This work was supported in part by NSF grant 0910483 and by a gift from RSA Laboratories.

References

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] FlexiScale utility computing. http://www.flexiant.com/.

[3] Rackspace. http://www.rackspacecloud.com/.

[4] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop*, pages 110–124, August 2010.

[5] O. Aciiçmez and Ç. K. Koç. Trace driven cache attack on AES (short paper). In *Information and Communications Security, 8th International Conference, ICICS 2006*, pages 112–121, 2006.

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

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

[8] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286, February 2007.

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

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

[11] Amazon AWS. AWS case study: Numerate finds a winning combination with AWS. http://bit.ly/a0ONsQ, 2010.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[13] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[14] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Topics in Cryptology — CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010*, pages 235–251, March 2010.

[15] J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop*, pages 201–215, 2006.

[16] B. B. Brumley and R. M. Hakala. Cache-timing template attacks. In *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, 2009.

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

[18] T. Carlson. Secure cloud offerings for government. MSDN blog, http://bit.ly/b2e1sI, 24 February 2010.

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

[20] D. Chisnall. *The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, November 2007.

[21] PCI Security Standards Council. Payment card industry (PCI) data security standard v.2. http://www.pcisecuritystandards.org, October 2010.

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

[23] P. Grabher, J. Großschädl, and D. Page. Cryptographic side-channels from low-power cache memory. In *Cryptography and Coding, 11th IMA International Conference*, pages 170–184, December 2007.

[24] M. Henricksen, W. S. Yap, C. H. Yian, S. Kiyomoto, and T. Tanaka. Side-channel analysis of the K2 stream cipher. In *Information Security and Privacy, 15th Australasian Conference, ACISP 2010*, pages 53–73, July 2010.

[25] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-CGM. In *Cryptographic Hardware and Embedded Systems — CHES 2009, 11th International Workshop*, pages 1–17, September 2009.

[26] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[27] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology — Proceedings of Crypto'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[28] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *2nd ACM Workshop on Computer Security Architectures*, pages 25–34, October 2008.

[29] R. Könighofer. A fast and cache-timing resistant implementation of AES. In *Topics in Cryptology — CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008*, pages 187–202, April 2008.

[30] G. Leander, E. Zenner, and P. Hawkes. Cache timing analysis of LFSR-based stream ciphers. In *Cryptography and Coding, 12th IMA International Conference*, pages 433–445, December 2009.

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

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

[33] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006*, pages 147–162, August 2006.

[34] M. Neve, J.-P. Seifert, and Z. Wang. A refined look at Bernstein's AES side-channel analysis. In *ACM Symposium on Information, Computer and Communications Security*, March 2006.

[35] C. Percival. Cache missing for fun and profit. In *BSDCon 2005*, 2005.

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

[37] C. Rebeiro, D. Mukhopadhyay, J. Takahashi, and T. Fukunaga. Cache timing attacks on Clefia. In *Progress in Cryptology – INDOCRYPT 2009, 10th International Conference on Cryptology in India*, pages 104–118, December 2009.

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

[39] B. Stone and A. Vance. Companies slowly join cloud-computing. *New York Times*, 18 April 2010.

[40] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An analytical model for time-driven cache attacks. In *Fast Software Encryption, 14th International Workshop, FSE 2007*, pages 399–413, 2007.

[41] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[42] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems — CHES 2003*, pages 62–76, 2003.

[43] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, pages 803–806, 2002.

[44] Y. Tsunoo, E. Tsujihara, M. Shigeri, H. Kubo, and K. Minematsu. Improving cache attacks using cipher structure. *International Journal of Information Security*, 5(3):166–176, 2006.

[45] J. Vijayan. Will security concerns darken Google's government cloud? *ComputerWorld*, 17 September 2009.

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

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

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

[49] E. Zenner. A cache timing analysis of HC-256. In *Selected Areas in Cryptography, 15th International Workshop, SAC 2008*, pages 199–213, August 2009.