CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing

Fangfei Liu[1], Qian Ge[2,3], Yuval Yarom[2,4],
Frank Mckeen[5], Carlos Rozas[5], Gernot Heiser[2,3], Ruby B. Lee[1]

[1] Department of Electrical Engineering, Princeton University, email: {fangfeil,rblee}@princeton.edu
[2] NICTA, email: {qian.ge,gernot}@nicta.com.au
[3] UNSW Australia
[4] School of Computer Science, The University of Adelaide, email: yval@cs.adelaide.edu.au
[5] Intel Labs, email: {frank.mckeen,carlos.v.rozas}@intel.com

ABSTRACT

Cache side channel attacks are serious threats to multi-tenant public cloud platforms. Past work showed how secret information in one virtual machine (VM) can be extracted by another co-resident VM using such attacks. Recent research demonstrated the feasibility of high-bandwidth, low-noise side channel attacks on the last-level cache (LLC), which is shared by all the cores in the processor package, enabling attacks even when VMs are scheduled on different cores. This paper shows how such LLC side channel attacks can be defeated using a performance optimization feature recently introduced in commodity processors. Since most cloud servers use Intel processors, we show how the Intel Cache Allocation Technology (CAT) can be used to provide a system-level protection mechanism to defend from side channel attacks on the shared LLC. CAT is a way-based hardware cache-partitioning mechanism for enforcing quality-of-service with respect to LLC occupancy. However, it cannot be directly used to defeat cache side channel attacks due to the very limited number of partitions it provides. We present CATalyst, a pseudo-locking mechanism which uses CAT to partition the LLC into a hybrid hardware-software managed cache. We implement a proof-of-concept system using Xen and Linux running on a server with Intel processors, and show that LLC side channel attacks can be defeated. Furthermore, CATalyst only causes very small performance overhead when used for security, and has negligible impact on legacy applications.

1. INTRODUCTION

Infrastructure-as-a-service (IaaS) is a cloud computing paradigm where the cloud provider leases out computing resources to cloud customers/tenants in terms of virtual machines (VM). Multiple VMs may execute on the same physical machine and share all its underlying hardware resources. Recent research has demonstrated that this sharing may allow a malicious VM to extract sensitive information from co-resident VMs, a serious security concern for cloud adoption [22, 29, 34, 45, 46]. The last-level cache (LLC) is one of the most dangerous shared resources since it is shared by all of the cores in a processor package, yet it allows fine-grained, high-bandwidth, low-noise cross-core attacks [22, 29, 42].

Due to its large size, partitioning the LLC seems like a promising strategy for mitigating its use for side channel attacks. Hardware cache partitioning architectures have been proposed in [11, 33, 40], but these targeted L1 cache side channel attacks, not LLC attacks. Software based page coloring techniques [25, 27, 36] have also been proposed, but these are incompatible with superpages (without which performance will degrade for large workloads), and have some other issues that we will discuss later in Section 2 and Section 7.

Recently, Intel introduced a performance optimization feature called the Cache Allocation Technology (CAT) [20, 21]. CAT supports associating cache ways with up to four *Classes of Service* (COSs), such that the replacement of a cache line is allowed only if the code initiating the replacement executes within the corresponding COS.

We propose to use CAT to defend against LLC side channel attacks. A direct approach is to use CAT to provide partial or complete isolation of COS for partitioning the L1 cache [11] . However, there is a fundamental problem with this direct partitioning approach— CAT supports only four COSs— too few for security purposes, especially in cloud computing, where the number of VMs on a cloud server can be much higher.

In this work, we propose an alternative approach, called CATalyst, for using CAT as a defense for LLC side channel attacks. On the one hand, we use CAT as a coarse-grained mechanism to partition the LLC into secure and non-secure partitions, where the non-secure partition can be freely used by any applications. On the other hand, the secure partition is loaded with cache-pinned *secure pages*. Thus, the LLC becomes a hybrid between a hardware-managed and a software-managed cache: the non-secure partition remains hardware-managed, while the secure partition is software-

978-1-4673-9211-2/16/$31.00

managed, leveraging CAT to enforce the cache-pinning of secure pages.

We present a proof-of-concept implementation using a Xen hypervisor running Linux-based VMs on a cloud server using commodity Intel processors. Our implementation provides applications with an explicit interface for using secure pages to store security-sensitive code and data pages. The implementation ensures that malicious VMs cannot evict the sensitive data in these secure pages from the cache, thereby preventing LLC side channel attacks on these pages.

We evaluate the implementation and show that CATalyst completely mitigates the attacks, while introducing negligible performance overhead (on average 0.7% over the SPEC benchmark suite and 0.5% for PARSEC.)

This paper makes the following contributions:

- A new mitigation scheme called CATalyst for LLC side channel attacks using recently-introduced hardware performance features in commodity processors. (Section 3.)
- A concrete implementation of the design, showing how secure page cache-pinning in the LLC can be achieved. (Section 4.)
- Security evaluation (Section 5) and performance evaluation (Section 6) of the design, showing that LLC cache side channel attacks can be defeated with minimal performance overhead.

2. BACKGROUND

2.1 Virtualization

Virtualization is the key underlying technology for IaaS clouds. Virtualization supports running multiple isolated environments, called *virtual machines* (VMs). Each VM provides the software within it with the illusion of executing on its own machine. A *virtual machine monitor* (VMM) manages the underlying hardware resources, which not only allocates those resources to the VMs but also provides the isolation boundaries among VMs.

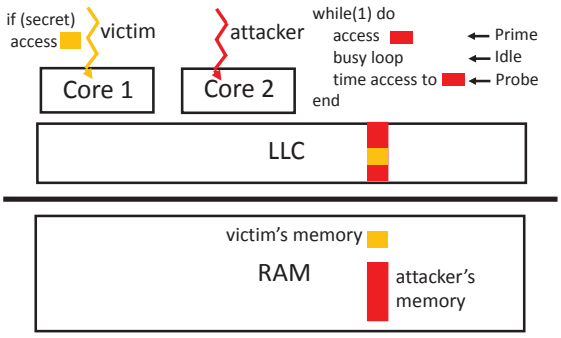
To maintain the illusion of VMs executing on isolated hardware, the VMM relies on hardware support. Whenever the software in a VM executes a sensitive instruction, i.e., an instruction that may result in breaking the illusion, the processor generates a *VM exit* that traps into the VMM. The VMM then emulates the operation of the sensitive instruction to maintain the illusion.

There are two levels of address-space virtualization in virtualized environments. The first maps the virtual addresses of a process to a guest's notion of physical addresses, which are emulated by the VMM. The second maps guest physical addresses to host physical addresses. The two levels of address translation can be achieved in two alternative ways. Without specific hardware support, the VMM can maintain a shadow page table for each process to store the translation from the virtual address to the host physical address. The VMM needs to intercept the guest OS (operating system)'s page table updates in order to keep the shadow page table coherent with the guest page table, which is very costly. This overhead can be removed with hardware support for *nested page tables*. For example, Intel introduced an *extended page table* (EPT) mechanism [20], which features a second page table that directly stores the translation from the guest physical address to the host physical address for each guest VM. The page table walker needs to traverse both the guest page table and the EPT to get the translation from the virtual address to the host physical address.

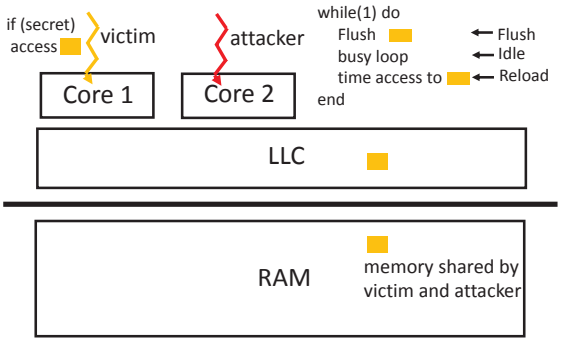
To improve performance, most VMMs allow breaking the illusion in certain cases through the use of *paravirtualization* [2]. The VMM provides some services in terms of *hypercalls* that are typically not part of the instruction set of the architecture, which the OS in the VM can use.

Emulating sensitive instructions provides a logical isolation between VMs. Instructions that normally affect the logical state of the machine are emulated so that their effect is visible only within a VM. However, the emulation does not achieve temporal isolation. That is, some instructions take a different amount of time to execute on a VM. There are several causes for the lack of temporal isolation. In this work we are interested in timing differences due to sharing the LLC, which allow mounting LLC-based side channel attacks.

2.2 LLC-based side channel attacks



(a) The PRIME+PROBE attack



(b) The FLUSH+RELOAD attack

Figure 1: Two LLC-based attacks.

Cache side channel attacks can infer secret information processed by a victim through measuring its cache usage patterns. While L1 and L2 caches are core-private, the LLC is shared between cores, hence LLC-based attacks are also executable when the victim and the attacker are not executing on the same core. Furthermore, as co-resident VMs share

the LLC, LLC-based attacks can operate cross-VM. Recent work has shown that LLC attacks are powerful enough to extract fine-grained secret information with high resolution and low noise [22, 29, 42]. We now describe the two common techniques used in LLC-based attacks: PRIME+PROBE and FLUSH+RELOAD.

2.2.1 *The* PRIME+PROBE *attack*

The PRIME+PROBE attack exploits resource contention between two processes, allowing an attacker process to measure the cache usage of a victim process. Figure 1a illustrates this attack. The attacker first constructs an *eviction buffer* that consists of cache lines in the same cache set as a target victim cache line. In order to observe when the victim process accesses the target cache line, the attacker performs repeated measurements consisting of three steps. First, the attacker primes the targeted cache set by accessing the cache lines in its eviction buffer. This, effectively, fills the monitored cache set with the attacker's data. Then, the attacker busy-waits for a set time. Lastly, the attacker probes the cache set by measuring the time to access the cache lines in the eviction buffer. If the victim accesses the target cache line between the prime and the probe steps, the target cache line will evict an attacker line from the cache, resulting in a longer probe time than otherwise.

2.2.2 *The* FLUSH+RELOAD *attack*

The FLUSH+RELOAD attack is applicable when the attacker and the victim share the memory page containing the target cache line. Page sharing is commonly implemented in both the OS and the VMM, e.g., as shared libraries or through memory deduplication [39]. Figure 1b illustrates this attack. FLUSH+RELOAD, like the PRIME+PROBE attack, consists of three steps. Firstly, the attacker flushes the target cache line from the cache, e.g., using the `clflush` instruction. Then, the attacker busy-waits for a set amount of time. Lastly, the attacker measures the time to reload the target line. If the victim accesses the target cache line between the flush and the reload steps, the line will be cached and the reload will be faster than otherwise.

2.3 Page coloring

Page coloring is a software mechanism that partitions the cache by cache sets at page-size granularity. Memory pages are given different colors, and only memory blocks in pages with the same color can be mapped into the same cache set. The page frame allocation is changed so that pages that are contiguous in the virtual address space will have different colors. Since the granularity of page coloring is 4KB small pages, it is incompatible with superpages, without which performance may degrade due to extra TLB misses [35]. Also, page coloring does not support memory deduplication across VMs, potentially increasing memory pressure [39]. Page coloring is inherently coarse-grained, significantly reducing the amount of cache available to any VM, which may lead to performance degradation.

2.4 The Intel sliced LLC

Starting with the Nehalem microarchitecture, Intel introduced the sliced LLC architecture that splits the LLC into multiple *slices*. Each of the slices operates as a standard cache, indexed by part of the physical address. The cores and the cache slices are interconnected by a bi-directional ring bus, and there is one slice for each core.

To map a physical address to a cache slice, the processor uses an unpublished hash function, which distributes the physical addresses uniformly among the slices [19]. Not disclosing the details of the hash function hampers side channel attacks as well as defenses [43]. In order to find the eviction buffer for mounting a PRIME+PROBE attack, the attacker needs to know which memory locations map to which cache sets—information which is partially encoded by the hash function. In order to implement page coloring as a defense, the system designer must ensure that pages of different colors do not map to the same cache set. Without understanding the hash function, the system can only rely on address bits used for indexing the set within a slice, reducing the number of available colors.

The hash function can be reverse-engineered by finding conflicting memory addresses, experimentally or by using the performance counters of the processors [30]. When the number of cores in the processor is a power of two, the hash function is linear [18, 23, 29]. For other core counts, the hash function cannot be linear [29, 43].

3. LEVERAGING CAT TO DEFEAT LLC ATTACKS

3.1 The Intel Cache Allocation Technology

CAT is a hardware-based mechanism, introduced in Intel Haswell server processors in 2014, which supports configurable partitioning of the cache ways of the LLC. Unlike page coloring, CAT is not affected by the hash indexing algorithm of the LLC and supports superpages. CAT is designed to enforce quality-of-service (QoS) through isolating cache footprints, e.g., by guaranteeing cache capacity to high-priority applications.

On current Haswell servers, CAT supports up to four classes of service (COS), with a bitmap defining the LLC ways a COS can access. The bit mask can be configured in software by writing to one of four model-specific registers (MSRs), one for each COS. Intel imposes two constraints on configuring the COS: the bit mask must contain at least two ways and the allocated ways must be contiguous. CAT associates COS to logical processors. As a result, all the applications running on a logical processor share the same COS. Importantly, CAT only restricts the ways that can be used to serve an LLC miss, applications can still hit on cache lines that are cached in cache ways that are not assigned to them.

3.2 Challenges

CAT could be trivially used to completely partition the cache, by assigning disjoint COSs to different VMs (Figure 2.). However, such a partitioning is too coarse-grained, as the hardware only supports four COSs, and four security domains are insufficient for realistic cloud workloads. Furthermore, the system could not dynamically balance the cache allocation according to the needs of VMs, which might harm the performance of VMs with large working sets, especially when most VMs have small working sets.

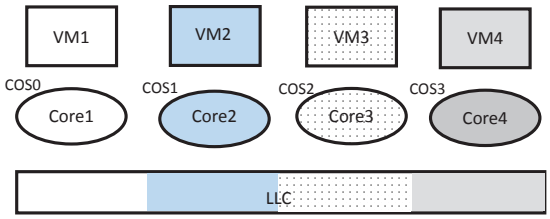


Figure 2: Straightforward way-partitioning of the LLC between cores (only 4 partitions).

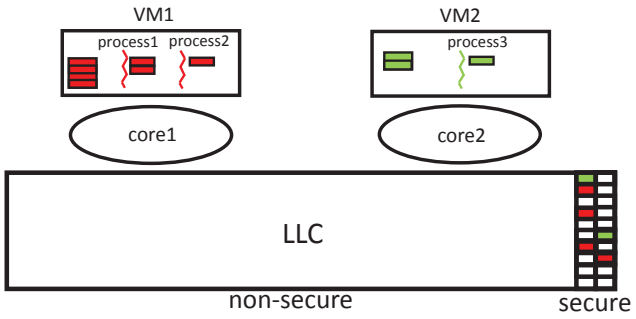


Figure 3: CATalyst architecture, with finer-grained, page-level partitioning of 2 ways of the LLC.

3.3 Threat model and design goals

We target the co-residence threat in public IaaS clouds. Without losing generality, our threat model assumes that each VM is a trust domain, i.e., cloud customers do not trust each other. The attacker owns a VM, having complete control of both OS and applications in the VM. The victim is a co-resident VM that shares the machine with the attacker VM. Our goal is to ensure that running it in a public cloud platform is as secure as running a VM on a private machine. We assume that the OS and the applications in the victim's VM are trusted. We also trust the cloud provider and any software under its control. In particular, we trust that the VMM manages the VMs, their resource allocations and the CAT correctly.

We focus on LLC-based cross-VM attacks. The attacks based on core-private caches (e.g., L1 and L2 caches) are extremely difficult to perform in the cross-VM scenario, since they need to exploit the vulnerability of the VMM scheduler (e.g., the Xen credit scheduler) to achieve fine-grained sharing of the same core for two VMs [46]. Additionally, different hardware or software solutions have already been proposed for L1-based side channel attacks. These include secure cache designs [11, 28, 40, 41], scheduler-based defenses [38] and software-based cache cleansing mitigations [44]. Furthermore, cloud providers have the option not to schedule two VMs from different cloud customers on the same core, but VMs still share the LLC even when they are scheduled on different cores. Our goal is providing a system-level mechanism that offers strong security guarantees to cloud providers and their customers. We want a practical solution that is applicable to commodity hardware. Furthermore, we want a scalable solution that can handle a large number of VMs and does not incur prohibitive performance costs. Lastly, we do not want a solution that increases the risk of denial-of-service (DoS) attacks.

3.4 Our solution: CATalyst

The key idea of CATalyst is that we use CAT not only as a cache partitioning mechanism but also as a pseudo-locking mechanism which pins certain page frames in the LLC. These cache-pinned page frames are called *secure pages* that can be used to store security-sensitive code and data by remapping them to the secure page frames.

As shown in Figure 3, we first use the CAT hardware to create two partitions: a secure partition and a non-secure partition, where the secure partition can be much smaller than the non-secure partition. We dedicate the secure partition to store cache-pinned secure pages. The non-secure partition remains hardware-managed, and thus transparent to the software. The secure partition, however, becomes a software-managed cache that is allocated as the secure partition at VMM boot time. In this way, our solution converts the LLC into a hybrid between a hardware-managed cache (using CAT) and a software-managed cache, providing finer granularity partitions, and locking of secure pages (described below). Although the effective LLC capacity for applications that do not use secure pages is reduced to the size of the non-secure partition, the performance degradation is small, owing to the large size (tens of MB) of the LLC in modern processors, and to the relatively small size of the secure partition, which can be as small as two out of many (e.g., 20) LLC ways.

Secure pages are assigned to VMs upon request at VM launch time. Allocation is done at a granularity of a page, effectively increasing the number of cache partitions to the number of pages in the secure partition. We can, therefore, support many more mutually-distrusting VMs than the number of COS. The VMM guarantees that a secure page is not shared by multiple VMs. Yet, within a VM, assigned secure pages are fully managed by the guest OS and can be time-shared by all the applications in the VM.

Security guarantees: CATalyst provides two security guarantees:

1) *No eviction of secure pages by malicious code.* We guarantee that once security-sensitive code or data is mapped to secure pages, they are kept in the LLC. That is, malicious code cannot cause eviction through cache line contention. Accesses to security-sensitive code or data always hit in the LLC, thus achieving constant-time accesses, which do not leak information to an attacker observing the shared LLC cache state. Note that unlike some previous work [26], we do not require explicit software preloading of security-sensitive code or data to achieve access time guarantees (due to cache hits). Instead, by remapping the security-sensitive code or data to the secure pages, we implicitly perform preloading.

2) *No overlapping of secure pages between different active VMs (security domains).* By preventing sharing of secure pages between VMs, we guarantee that the FLUSH+RELOAD attack cannot operate on security-sensitive code or data. This also prevents a malicious VM from flushing the secure pages belonging to another VM.

To avoid DoS attacks by malicious VMs over-requesting

secure pages, the VMM limits the maximum number of secure pages that a VM can request, for example, to no more than 8 pages.

Scalability: The number of cache partitions that our system can support is essentially the number of pages in the secure partition. Assume that there are N_{slice} LLC slices, the size of each way in a LLC slice is S bytes, and the number of reserved ways for the secure partition is k. In a typical Intel processor, N_{slice} is the number of cores in the processor and S is fixed at 128 KB, which is 32 page frames. Therefore, the secure partition contains $32 \times k \times N_{slice}$ page frames. Due to the restrictions of CAT allocation, we have $k \geq 2$. Hence, the smallest possible secure partition has 64 page frames per processor core.

Looking at Amazon EC2, as an example of a typical public cloud provider, we find that the smallest supported instance, the T2.micro, has a burstable CPU allocation of 10% of the performance of a single core [3]. If we ignore the VMM overhead, assume no larger instances are used and allow for some over-provisioning, we can estimate an upper bound of 16 VMs per CPU core. Even with such an upper bound, 64 secure pages per core can still accommodate four secure pages per VM. A few secure pages are usually sufficient for protecting the security-sensitive code and data since these are usually small.

3.5 Overview of the CATalyst system

The CATalyst system consists of the following phases:

1) **VMM boot time:** The VMM reserves a pool of physically contiguous page frames, and loads these pages into secure partitions using CAT. We allocate all secure pages in the secure partition at boot time as this implies a one-off overhead that is drowned by the boot latency. We show what is involved in loading secure pages into the secure partition with high security guarantees, and we will discuss this in detail in Section 4.

2) **VM launch time:** At VM launch time, the VMM allocates secure pages to the guest physical address space of the guest VM according to its request. The VMM guarantees that VMs do not share a secure page.

3) **VM run time (process):** At VM run time, the guest kernel manages the secure pages without interventions from the VMM. The guest kernel exposes two system calls to the process, for mapping and unmapping the secure pages. If a process wants to protect its sensitive code or data, the guest kernel only needs to remap the sensitive code or data to the secure pages, which implicitly does preloading and locking of these pages.

4) **VM terminate time:** The allocated secure pages are reclaimed and scrubbed when a VM is terminated, and released to the VMM.

In the following section, we will discuss in detail how the secure pages are managed, how to expose secure pages to the guest VM, and how to reclaim the secure pages. The implementation of the system calls and how an application uses the system calls will also be described in detail.

4. SYSTEM DESIGN

4.1 VMM boot time

	M9	M8	M7	M6	M5	M4	M3	M2	M1	M0
COS0	1	1	1	1	1	1	1	1	0	0
COS1	1	1	1	1	1	1	1	1	0	0
COS2	1	1	1	1	1	1	1	1	0	0
COS3	0	0	0	0	0	0	0	0	1	1

Figure 4: Example bit mask configuration for CAT as a pseudo locking mechanism.

input: Secure page P to be pinned.
C : code that loads a secure page, P

1) Disable local interrupts
2) Access C
3) Access one word in P
4) Set current logical processor to COS3
5) clflush P
6) Access P
7) Set current logical processor back to COS0
8) Enable local interrupts

Figure 5: Procedure to pin one secure page into the secure partition.

4.1.1 Pseudo locking mechanism

CAT itself is a cache partitioning mechanism. However, with careful configuration of CAT, it can also be used as a pseudo locking mechanism. Specifically, we dedicate one COS, say COS3, for security, and ensure no process except a special preloading routine can be assigned to COS3. During the initialization of CAT, we configure the capacity bit mask for each COS before locking the secure pages. Figure 4 shows an example bit mask configuration. In this example, there are 10 cache ways; way M_0 and M_1 are reserved for the secure partition and all of the remaining cache ways are used for the non-secure partition. COS3 is configured to be able to access the secure partition only, and all of the other COSs can only access the non-secure partition. We also set the MSR associated with each logical processor so that each is initially associated with COS0.

4.1.2 Preloading routine

The preloading routine (outlined in Figure 5) has to be carefully designed to ensure security. Specifically, our security guarantee is that secure pages always hit in the secure partition of the LLC. If data in a secure page was ever evicted out of the secure partition, it would become susceptible to attacks.

Once a logical processor is set to a COS, all the memory traffic initiated by that logical processor will follow the cache allocation policy of this COS. Such memory traffic includes instruction fetches and data load and stores, traffic originating from prefetchers, and even the page table walker. Therefore, we must guarantee that:

1) The preloading routine code and any non-sensitive data, e.g., the local variables on the stack, must never be loaded into the secure partition, to avoid evicting the secure pages.

2) No interrupt or exception handling is allowed while the preloading routine is executing. Otherwise, the interrupt or exception handler will execute with the logical processor setting of COS3 and may potentially evict some secure pages

out of the LLC cache.

3) No TLB miss (specifically in the unified L2 TLB) is allowed while the preloading routine is executing. This is because when the hardware walks the page tables, it may cache page-table data in the normal data-cache hierarchy. This could pollute the secure partition.

4) We also need to guarantee that the secure pages are never cached in any part of the non-secure partition.

To overcome the above challenges, we carefully write the preloading routine in assembly without using a stack or heap. This way, we only need to worry about the preloading routine's code, not data.

We first disable local interrupts to ensure that the preloading routine is not interrupted (Step 1 of Figure 5). Step 2, executed while the logical processor is still set to COS0, ensures that the preloading routine code is cached in the non-secure partition, and that the corresponding TLB entry is loaded. Since the preloading is done at boot time when only the boot core is running, we can guarantee that, once cached in the non-secure partition, the routine's code is not evicted from the cache or the TLB while preloading a page. Similarly, we also make sure no page fault or TLB miss occurs while accessing P. We achieve this by accessing one word in P to load the TLB entry for the page (Step 3). After all of these preparations, it is safe to set the current logical processor to COS3 (Step 4). Before loading P into the secure partition, we use the `clflush` instruction to evict any cached data of P from the non-secure partition (Step 5). We then traverse P (Step 6), ensuring it is cached in the secure partition. Finally, we exit the secure partition and go back to COS0 (Step 7) and re-enable interrupts (Step 8). The preloading routine consists of less than 100 lines of assembly code and achieves the same functionality as a hardware cache-pinning mechanism.

4.1.3 Preloading secure pages without self-conflicts

Ideally, a physically contiguous chunk, with size equal to the size of the secure partition (M bytes), will fit in the secure partition without conflicts. In some rare cases, this will be complicated by the use of the hash indexing scheme. Specifically, when the core count is not a power of two, we may find some conflicts within the M bytes in the physically contiguous chunk. This is because although the hash indexing scheme tries to evenly distribute traffic to each LLC slice, it is very hard to achieve this perfectly when the core count is not a power of two. To avoid LLC conflicts during the loading of secure pages, the VMM needs to allocate a chunk of physically contiguous memory that is slightly larger than M. For each page frame, the VMM needs to determine whether adding this page frame into the secure partition will cause the eviction of page frames that are already in the secure partition. There are two approaches to achieve this:

1) Reverse engineer the hash function [30, 43]. Since the VMM knows the physical address of a page frame, it is straightforward for the VMM to determine which LLC slice the page frame is mapped to, if the VMM knows the hash function.

2) The VMM can use the PRIME+PROBE technique iteratively [29]. Assume the VMM has already selected a set of page frames, F, without LLC cache line conflicts. In order to determine whether adding a new candidate page frame will result in conflicts, the VMM can first access a cache line L in the candidate page frame, and then access all the cache lines with the same page offset as L in the page frames in F. The VMM then measures the time taken to access L again. If the access latency is small, the VMM can add the candidate page frame in the set F; otherwise, adding this candidate page will cause self-conflicts since an LLC cache miss resulted in accessing L again.

We use the performance monitor unit (PMU) to verify that the secure pages are indeed pinned in the cache. The PMU allows software to access the hardware performance counters. Specifically, the VMM first allocates a large buffer (e.g., the same size as the LLC) and cleans the LLC by traversing the large buffer. Then the VMM uses the PMU to measure the number of LLC misses when randomly traversing a secure page. To count the number of LLC misses, we directly program one of the performance monitor counters (e.g., PMC0) so that it is reset to zero before we start to access the secure page; we read this counter for number of LLC misses, after traversing the secure page.

4.1.4 Other issues

Although our preloading routine is carefully designed to provide strong security guarantees, there are still several complexities to handle in a real commodity system, in order to prevent the pinned page frames from being evicted out of the LLC.

Special instructions that explicitly flush the caches: Table 1 shows some existing Intel x86 instructions that may explicitly flush a cache line out of the LLC, without enforcing CAT, which will destroy our security guarantee once such an instruction is executed. In particular, `invd` and `wbinvd` may flush the whole per-core L1 and L2 caches (and the corresponding content in the LLC). A malicious VM may use such instructions to flush arbitrary secure pages out of the LLC. Since these instructions cause VM exits, the VMM can intercept them and prevent their execution.

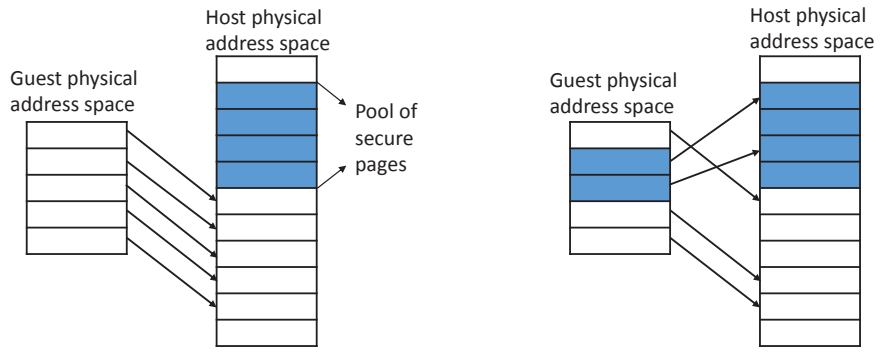
The `clflush` instruction and memory-access instructions bypassing the caches are potentially more dangerous, since they are unprivileged and can be executed by the guest without causing VM exits. However, since these instructions operate on data identified by the guest virtual address, they cannot affect another VM's secure pages, as we guarantee that VMs never share secure pages.

Once the secure pages are reclaimed at VM termination, the VMM needs to reload these pages to ensure they are still pinned in the secure partition. We allow the VMM to execute the above special instructions, but ensure that where they target cache lines in the secure partition, the VMM immediately afterwards reloads the flushed cache lines.

Unlike the loading of secure pages at boot time, reloading them requires special care. This is because: 1) concurrent accesses to the flushed secure pages by a VM running on another core may cache a secure page in the non-secure partition (e.g., while the VMM executes `wbinvd`); 2) there is no guarantee that the preloading routine code, which is cached in the non-secure partition, will not be evicted when reloading the secure pages. In order to maintain our security guarantees we stop all other logical processors while the

Table 1: Special instructions that may explicitly flush cache.

Instructions	Description	VM exit?
invd(wbinvd)	(write back) and invalidate cache	yes
clflush	invalidate cache line with given linear address from all levels of cache hierarchy	no
non-temporal stores, e.g., movnti	write to given memory address without writing data in the cache hierarchy	no



(a) EPT mapping on VM launch **(b) EPT mapping after assigning secure pages**

Figure 6: Assigning secure pages to guest VM from a pool of secure pages. The shaded pages are secure pages.

Table 2: Summary of hypercalls.

Hypercall	Description
`long` do_alloc_secure_pages(`unsigned long` start_gpfn, `int` nr_pages)	allocate secure pages for nr_pages guest page frames starting at start_gpfn
`long` do_release_secure_pages(`void`)	release all the secure pages allocated to this VM

VMM is reloading the flushed pages. Compared to the cost of cache flushes and VM cleanup operations, the extra overhead is small.

Direct Data I/O (DDIO): DDIO is a new feature that can be found in Intel Xeon E5 and E7 series processors. It enables direct communication between Intel Ethernet controllers and adapters and the host processor's cache. Therefore, it may flush the LLC silently without the VMM being aware. Fortunately, the Intel processor allows software to configure which cache ways are allowed to be accessed by the DDIO through a bit mask. Therefore, we can configure the bit mask so that it is not overlapped with the cache ways allocated to the secure partition.

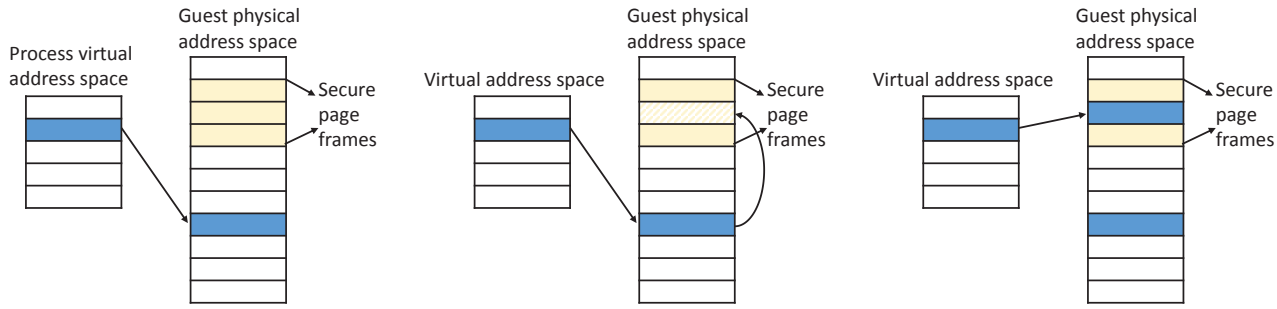
Cache coherence in multi-socket servers: So far, we have assumed a scenario with a single multicore processor package. Therefore, the intra-CMP (chip multiprocessor) coherence protocol does not impact the contents in the LLC. Especially, it is safe to keep dirty data in the LLC without writing back to the memory. However, public cloud providers often use multi-socket servers, i.e., servers that have multiple multi-core packages. The inter-CMP cache coherence protocol may destroy our security guarantee. In this case, if a processor core in one socket writes to a secure page which has been cached in the LLC of another socket, the secure page has to be invalidated. To solve this, we must set the CPU affinity of the VM's Virtual CPUs (VCPUs) to the cores of a single processor package. The secure pages of the VM are pinned to the LLC of the same processor package. Because the secure pages are not shared with other VMs, restricting the VM to the processor package which caches its secure pages ensures that no other processor package accesses the secure pages and prevents any cache coherence issues.

4.2 VM launch time and terminate time

At VM launch time, the VMM allocates secure pages to the guest VM as requested, and exposes these pages to the guest physical address space. We take advantage of the EPT mechanism and implement this operation as a kernel module in the guest kernel, without the need to modify the VM build process. When the kernel module is loaded into the kernel space, it allocates a chunk of contiguous guest page frames, and then asks the VMM to map these page frames to the secure pages.

Figure 6 illustrates the technique of exposing secure pages to the guest physical address space. In this example, all of the memory requested by the guest VM is allocated, and the EPT entries for the entire guest physical address space are populated after the guest VM is built. This is the default scenario in many virtualized systems (e.g., Xen). The kernel module invokes a new hypercall `do_alloc_secure_pages` (see Table 2) for mapping `nr_pages` guest page frames starting from frame number `start_gpfn` to secure pages. Since the EPT is already populated, the hypercall does the following operations: 1) free the original page frames that the guest physical page frames map to (and clear EPT entries); 2) allocate the secure pages; 3) modify the EPT entries to map to the secure pages.

We use a global data structure (a bit-vector-based free list, one bit for each secure page) to manage the allocation and



(a) bring in and lock backstore page **(b) copy backstore to a secure page** **(c) point PTE to secure page**

Figure 7: Mapping a virtual page to a secure page for a process.

Table 3: Summary of system calls.

System call	Description
`long` sys_map_sp(`unsigned long` start, `size_t` len)	map virtual address space starting at start, with size len to secure pages
`long` sys_unmap_sp(`unsigned long` start, `size_t` len)	unmap virtual address space starting at start, with size len from secure pages

release of secure pages. There is no global data structure for tracking which page frame is allocated to which domain. Instead, each domain keeps track of its own secure page allocation, including which secure pages (in terms of machine page frame numbers) are allocated to it, the beginning guest page frame number of the secure pages and the number of secure pages. This makes it easy to manage the release of the secure pages.

VM terminate time: At VM termination time, the secure pages are reclaimed and reloaded into the secure partition as described in Section 4.1.4, in order to prevent the flushing of secure pages by a malicious VM. Similar to launching a VM, we use a kernel module to reclaim secure page frames. Secure pages for a domain are released when the kernel module performs unloading, using a hypercall `do_release_secure_page` (see Table 2). For each guest page frame that is mapped to a secure page, the VMM does the following: 1) allocate a free machine page frame, and change the EPT to point to the new machine page frame; 2) free the secure page; 3) ensure that the secure page is pinned in the secure cache. (Recall that a malicious VM may have used `clflush` to evict the data of the secure page from the secure partition.)

4.3 VM run time

The guest VM's kernel manages the secure pages by multiplexing them to all the processes. There are two alternatives to implement a time-shared secure page management mechanism. One option is a truly time-shared implementation, i.e., virtualizing the secure pages for user processes so that two processes can simultaneously map to the same secure page as long as they do not access it concurrently. To avoid the complexity of synchronization on multi-core processors, the guest kernel can associate different secure pages to each VCPU. A process running on one VCPU can use all the secure pages owned by that VCPU. During context switch, all the secure pages on that core must be saved to some per-process backstore pages. The contents of the backstore pages of the next running process will be loaded into the secure pages. The second option is that the guest kernel does not virtualize the secure pages so that a user process can exclusively hold secure pages until they are explicitly released. In our proof-of-concept implementation we use the second option.

The guest kernel exposes two system calls to the user process: `map_sp` and `unmap_sp` as described in Table 3. Basically, these system calls allow a user process to map (and unmap) certain virtual address space to (from) secure pages. These interfaces are flexible enough to protect arbitrary code, stack, and heap, etc.

Figure 7 illustrates the implementation of the `map_sp` syscall. The user process passes the start virtual address and length of the protected region as parameters. `map_sp` first uses mlock to make sure the memory region has the `VM_LOCKED` flag set (i.e., cannot be swapped out), and fault in (i.e. explicitly invoke page fault handler) if necessary so that the memory region is mapped to certain guest physical page frames and the corresponding page table entries are populated. These guest physical page frames are the backstore for the secure pages. We then allocate secure pages for the memory region and copy the contents of the backstore page to the secure page frame. Similar to managing secure pages in the VMM, we also use a global data structure to manage the secure pages within the guest VM. The guest kernel maintains a bit vector based free list for the secure pages. In addition, for each secure page, we use a global data structure to record its backstore page. Note that this is different from the design we use in the VMM (which must manage a much larger number of secure pages than a guest). Lastly, we modify the page table entry so that it points to the secure page; we also save the backstore page frame to the global data structure for the corresponding secure page.

The implementation of `unmap_sp` is similar: the guest kernel first needs to copy the contents of the secure page to the backstore page if it is dirty. Then it changes the page table entry so that it points to the backstore page and then

Table 4: Summary of software changes.

Application	invoke syscalls to map and unmap secure pages
Guest kernel	new syscalls
VMM	new hypercalls, boot time code, VM exit handlers

frees the secure page.

Table 4 summarizes the changes to the software stack in order to use CATalyst. It is worth noting that our system still supports legacy guest VMs and applications if they are not aware of CATalyst.

5. SECURITY EVALUATION

Table 5: Experimental environment.

Processor model	Intel Xeon E5 2618L v3
Microarchitecture	Haswell
Clock frequency	2.3 GHz
# cores	8 (8 slices)
LLC	20-way 20 MB
# ways in secure partition	2
VMM	Xen 4.5.0 (HVM)
Dom0 and DomU	CentOS 6.6
Kernel version	Linux 3.14.0
Legacy VM	4 VCPUs, 4 GB RAM
Secure VM	4 VCPUs, 2 GB RAM, 8 secure pages

Table 5 summarizes the experimental environment for our implementation and evaluation. We use a Xeon E5 based Haswell server (with only one socket) with 8 cores. The size of the LLC is 20 MB, consisting of 8 cache slices. We reserve two out of the 20 cache ways as the secure partition, holding 10% of the entire cache. This secure partition supports up to 512 (20MB/4KB/10) secure pages, which allows our system to support a large number of VMs. We implement our design on a Xen hypervisor with Linux as the guest OS.

Algorithm 1: Square-and-Multiply exponentiation.

```
input  : base b, modulo m, exponent e = (e_{n-1} ... e_0)_2
output: b^e mod m
r ← 1
for i from n − 1 downto 0 do
    r ← r^2 mod m
    if e_i = 1 then
        r ← r · b mod m
    end
end
return r
```

To evaluate the security, we select GnuPG, a free implementation of the OpenPGP standard, as the victim process. The modular exponentiation in GnuPG version 1.4.13 uses a simple Square-and-Multiply algorithm [15] that is known to be vulnerable to side channel attacks. As described in algorithm 1, the algorithm scans the exponent bits (private key) from left to right, performing a square operation on each bit. For every "1" bit, it performs an extra multiply operation. Consequently, the time interval between two consecutive squares reveals the private key: if the exponent bit is "1" there will be a multiply operation in between two square operations, resulting in longer elapsed time between square operations than when the exponent bit is "0".

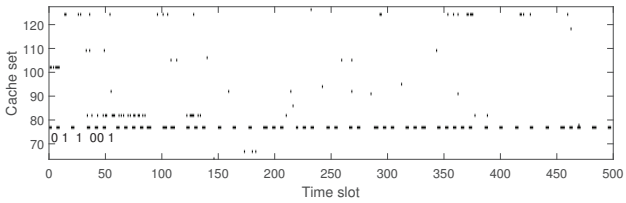


Figure 8: Clear activity pattern for cache set 78 in GnuPG without protection.

The attack first selects cache sets that potentially conflict with instructions in the square routine, i.e., they have the same page offset and hence will potentially map into the same cache set. The attack scans these selected cache sets, one by one, for 10,000 time slots, where each time slot is 5000 cycles. Figure 8 only shows cache set activity for some cache sets for 500 time slots. A dark dot represents activity in that time slot, whereas no dot represents inactivity in that time slot. For most cache sets, we only see random activity patterns. But for set 78, we see a clear string of activity (usually 3-4 consecutive dots for one square operation) followed by an inactivity period. This is a temporal access pattern that can indicate key bits: for this machine, the inactive period is 3-4 time slots for a "0" and 10-11 time-slots for a "1" in the key. Hence, a common activity pattern for set 78 is 3 time slots with activity followed by 4 time slots without activity, which is the matched activity pattern that we count in Figure 9.

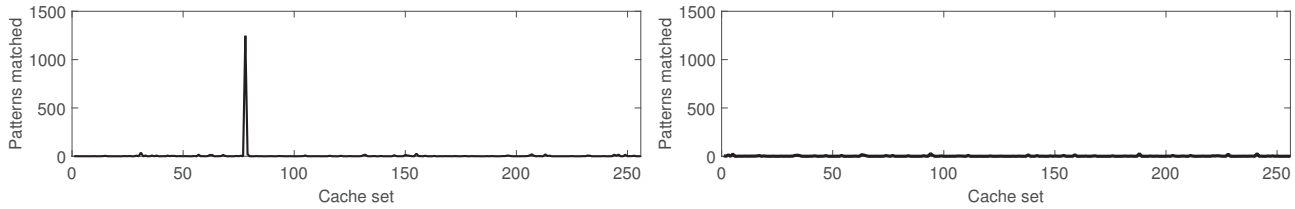
We modified GnuPG version 1.4.13 to use CATalyst. In particular, at the beginning of the modular exponentiation function, we invoke the `map_sp` syscall to map the page containing the square routine (as well as the multiply routine) to a secure page. At the end of the function, we invoke the `unmap_sp` syscall to release the secure page.

Figure 9 indicates the results of searching targeted cache traces, by counting the number of matched activity pattern described above, before and after we locked the secure page in the LLC. As shown in Figure 9a for GnuPG without protection, an attacker can easily identify the targeted cache set containing expected cache activity patterns. However, when CATAlyst is engaged as in Figure 9b, it is impossible to find such targeted cache sets.

6. PERFORMANCE EVALUATION

We measure the effect of our changes in two different scenarios. We first evaluate the costs of the reduced cache capacity on legacy applications. We then measure the effects of implementing the defense by modifying three encryption algorithms to protect their sensitive code and data. We use the two VM configurations shown in Table 5. We use the legacy VM for the legacy applications, running an unmodified OS without using secure pages. Moreover, we use the secure VM for measuring the costs of the defense.

6.1 Performance impact on legacy applications



(a) without CATalyst (clear pattern for set 78) **(b) with CATalyst**

Figure 9: Results on attacking GnuPG.

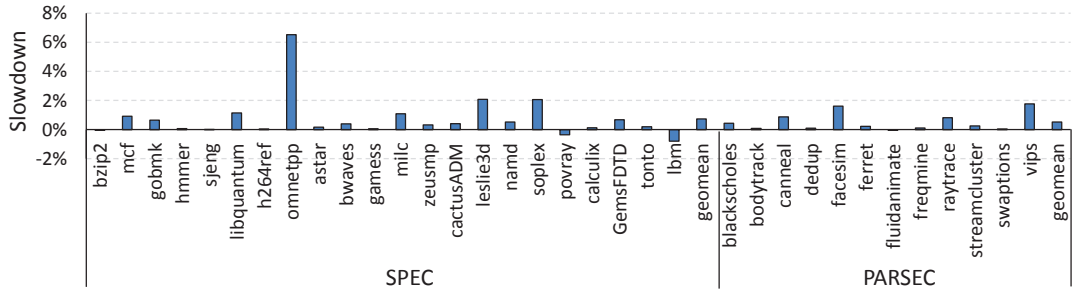


Figure 10: Slowdown for SPEC 2006 and PARSEC benchmarks.

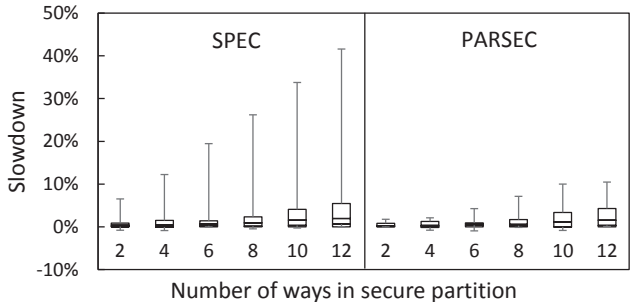


Figure 11: Box plot of slowdown for secure partition with different size.

We measure the impact on legacy applications using 22 SPEC CPU2006 [17] and 12 PARSEC [8] benchmarks. We run each benchmark three times to completion using the reference (for SPEC) and native (for PARSEC) input sets and compare the results against the baseline case—without the secure partition. Because the minimum secure partition occupies 10% of the cache, we expect some performance degradation when using our system.

Figure 10 shows the average slowdown of each of the benchmarks. We see an average slowdown of 0.7% for SPEC and 0.5% for PARSEC, with all but one benchmark being under 2%. The only exception is `omnetpp`, which has a slowdown of 6.5%. Hence, a decrease of 10% in the cache capacity incurs a very small performance degradation. Comparing with the state-of-art page coloring based scheme which gives an average degradation of 5.9% (maximum 16%) for SPEC benchmarks [25], CATalyst has much better performance for the legacy applications.

When a larger part of the cache is reserved for the secure partition, the performance deteriorates. Figure 11 shows the box plot of the slowdown distribution for various secure-partition sizes. Specifically, the ends of the whiskers represent the minimum and maximum slowdown over the 22 SPEC (or 12 PARSEC) benchmarks. The box represents the first and third quartile and the band in the box is the mean slowdown. Generally, SPEC is more sensitive to the LLC capacity than PARSEC benchmarks. For both benchmarks, we find that even when the secure partition occupies more than half of the LLC, 75% of the benchmarks suffer less than 6% performance degradation. Only very few benchmarks are sensitive to cache size, the worst suffering a performance degradation of up to 45%.

Although we only study the performance of SPEC and multi-threaded PARSEC benchmarks, we would expect that server benchmarks, scale-out workloads and real-world data center applications are even less sensitive to the LLC size than the memory-intensive SPEC and PARSEC benchmarks [13].

6.2 Overhead of security-sensitive applications

The performance of security-sensitive applications may be impacted by several factors: 1) the slowdown due to reduced cache size; 2) overhead due to mapping and unmapping secure pages; 3) potential performance improvement since frequently used code and data are pinned in the LLC.

We measure the performance effect of CATalyst on three cryptographic implementations: the OpenSSL 0.9.8.a implementation of AES encryption in cipher block chaining (CBC) mode, the GnuPG 1.4.13 and the GnuPG 1.4.18 implementations of the ElGamal decryption [12].

6.2.1 OpenSSL AES

The implementation of AES uses five 1 KB tables. Tracing accesses on these tables can recover the encryption

key [32]. To mitigate this attack, we map these tables to secure pages at the start of the encryption and unmap them at the end.

Table 6: Time to encrypt a 5 MB file using AES.

Buffer size	5 KB	50 KB	500 KB	5 MB
Baseline (ms)	81.04	79.91	79.83	82.87
CATalyst (ms)	95.26	81.96	79.96	82.83
Slowdown	17.54%	2.56%	0.16%	-0.05%

Table 7: GnuPG ElGamal decryption times.

Version	1.4.13	1.4.18
Baseline(ms)	15.36	12.77
CATalyst (ms)	15.40	12.83
Slowdown	0.25%	0.44%

The implementation uses an internal buffer for storing the data. Long streams of data are broken into buffer-sized chunks, each of these chunks encrypted separately. We measure the time to encrypt a 5 MB file containing random data with several buffer lengths. Table 6 shows the average results over 1,000 runs. With small buffers, the overhead of the map and unmap system calls is significant resulting in a slowdown of $\sim$17.5%. Increasing the buffer size dramatically reduces the overhead of map and unmap system calls. When the buffer size is larger than 500 KB, the impact is negligible. To reduce the overhead of mapping and unmapping the tables, a programmer can evoke the mapping at library initialization, therefore avoiding the overhead for every encryption.

6.2.2 GnuPG ElGamal

We test the performance with two versions of GnuPG: version 1.4.13 (discussed in Section 5) uses the square-and-multiply exponentiation algorithm; and version 1.4.18 uses the sliding window exponentiation [9]. We protect GnuPG 1.4.13 as specified in Section 5. For GnuPG 1.4.18, we protect both the multiply routine and the multipliers [29]. We map the multiply routine to a secure page at the beginning of the modular exponentiation function, and unmap it at the end of the function. We map the multipliers, which are dynamically allocated on the heap, immediately after allocation, and unmap them before deallocation.

GnuPG encryption uses a one-time symmetric key for encryption. It uses the ElGamal encryption to store an encrypted version of the one-time symmetric key with the data. We use GnuPG to encrypt a very short file (5 bytes), measuring the time to decrypt it. To accurately measure the time taken for ElGamal decryption, we insert `rdtsc` instructions before and after invoking the ElGamal decryption. Since ElGamal decryption is only performed on the one-time symmetric key, the time is independent of the size of the processed file. Similar to AES, we repeat each measurement 1,000 times. The results in Table 7 show that the overhead of the syscall is within 0.5% for both GnuPG 1.4.13 and GnuPG 1.4.18.

7. RELATED WORK

7.1 Constant time implementations

Cryptographic primitives sometimes can be reprogrammed such that they do not leak side channel information through the cache. Such *constant time* implementations avoid secret-dependent memory accesses and branches [5, 6].

Constant time implementations of cryptographic primitives are extremely hard to get right: for example, recent work has demonstrated that a supposedly constant-time implementation of TLS in OpenSSL still has data-dependent execution time at least on ARM processors [10]. Also, even where real constant-time implementations exist, they may not always be usable. For example, the generic implementation of elliptic curves in OpenSSL has been shown to be vulnerable to side channel attacks [1, 4, 37]. For the popular NIST P-256 curve, OpenSSL provides a constant-time implementation [16]. However this implementation is not part of the FIPS 140-2 [31] certified code base, and therefore developers that need FIPS 140-2 certification must use the vulnerable generic implementation.

7.2 Secure caches

Wang and Lee [40] designed a secure cache for dynamic cache partitioning (PLcache) by introducing a lock-bit per cache line, providing a hardware mechanism for locking security-sensitive data into the L1 data cache. Domnister et al. [11] propose that each hardware thread in a processor core statically owns several cache lines in every cache set of the L1 cache. However, they target L1 caches, not the LLC, and would require hardware changes to processor chips.

7.3 Page coloring based defenses

Page coloring was initially proposed for improving system performance [7, 24] and enhancing performance isolation in real-time systems [27]. Later on, the technique has been used by system designers for mitigating cache-based timing attacks [14, 25, 36].

Shi et al. [36] designed a dynamic page coloring solution to limit cache side channel attacks. Their solution dedicates several colors for security-critical operations and does not allow other processes to use these colors when security-critical operations are in execution. This solution may potentially cause significant overhead since it needs to frequently swap pages between secure colors and other colors. Godfrey [14] implemented page coloring in Xen, and evaluated its effectiveness. STEALTHMEM [25] reserves one or more stealth pages with different colors for each core, for protecting security-sensitive data. To ensure stealth pages are pinned in the cache, it disallows using page frames with the same colors as the stealth pages, or mediates access to those pages by page faults.

Compared to these software page coloring based solutions, CATalyst has the following advantages: 1) the design is simpler since the isolation of secure and insecure partitions is enforced by the CAT, instead of by complicated software. This results in modification of less than 1,000 C/assembly lines in the VMM. 2) CATalyst is not impacted by the Intel LLC's non-linear and unknown hash indexing scheme, whereas this may significantly reduce the already limited number of available colors in page coloring schemes.

3) CATalyst is compatible with superpages and memory deduplication, whereas these performance-enhancing features may have to be disabled in page coloring schemes.

8. CONCLUSIONS

We propose CATalyst, a lightweight system mechanism for the cloud provider and cloud customers to protect security-sensitive code and data against LLC-based side channel attacks, by leveraging the CAT on Intel processors. CATalyst builds on existing commodity hardware, and can be easily deployed. It bridges the gap between protecting security applications and the existing performance-oriented cache-partitioning mechanism. CATalyst uses the CAT to partition the LLC, forming a hybrid between a conventional hardware-managed cache and a small software-managed cache. We pin a pool of page frames permanently in the cache, managed by the VMM and the guest OS. Users can load and lock security-sensitive code and data by mapping them to the cache-pinned pages. We implement a proof-of-concept of CATalyst using the Xen hypervisor and the Linux OS. Our evaluation shows that CATalyst not only effectively mitigates the LLC attacks but also introduces very small performance degradation.

Acknowledgments

We thank the reviewers for their helpful comments. This work was supported in part by DHS/AFRL FA8750-12-2-0295 and NSF CNS-1218817. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," IACR Cryptology ePrint Archive, Report 2015/1141, Nov 2015.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003, pp. 164–177.

[3] J. Barr, "New low cost EC2 instances with burstable performance," https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/, Jul. 2014.

[4] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah..., just a little bit": A small amount of side channel can go a long way," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Sept. 2014, pp. 75–92.

[5] D. J. Bernstein, "Cache-timing attacks on AES," 2005, Preprint available at http://cr.yp.to/papers.html#cachetiming.

[6] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America (Latincrypt)*, 2012.

[7] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1994, pp. 158–170.

[8] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[9] J. Bos and M. Coster, "Addition chain heuristics," in *International Cryptology Conference (CRYPTO)*, Aug 1989, pp. 400–407.

[10] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of some timing channels on seL4," in *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014, pp. 570–581.

[11] L. Domnister, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *Trans. Arch. & Code Optimization (TACO)*, vol. 8, no. 4, Jan. 2012.

[12] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology*, 1985.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 37–48.

[14] M. Godfrey, "On the prevention of cache-based side-channel attacks in a cloud environment," Master's thesis, Queen's University, Ont, CA, Sep. 2013.

[15] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, Apr. 1998.

[16] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Engin.*, vol. 5, pp. 141–151, Jun 2015.

[17] J. Henning, "SPEC CPU2006 benchmark descriptions," *Comp. Arch. News*, vol. 34, no. 4, Sep 2006.

[18] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy (S&P)*, May 2013, pp. 191–205.

[19] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, Apr 2012.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Sep 2013.

[21] *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, Intel Corporation, Apr 2015.

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[23] ——, "Systematic reverse engineering of cache slice selection in Intel processors," IACR Cryptology ePrint Archive, Report 2015/690, Jul 2015.

[24] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *Trans. Comp. Syst.*, vol. 10, pp. 338–359, 1992.

[25] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security Symposium*, Aug. 2012, pp. 189–204.

[26] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[27] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Jun. 1997.

[28] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2014.

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

[30] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *RAID*, Kyoto, Japan, Nov 2015.

[31] *Security Requirements for Cryptographic Modules*, NIST, Dec. 2002.

[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, Nov. 2005.

[33] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, p. 280, 2005.

[34] 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 *ACM conference on Computer and communications security (CCS)*, Nov. 2009, pp. 199–212.

[35] T. H. Romer, W. H. Ohllrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and memory overhead using online superpage promotion," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Jun. 1995, pp. 176–87.

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

[37] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *RSA Conference Cryptographers' Track (CT-RSA)*, Apr. 2015, pp. 3–21.

[38] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-VM side-channels," in *USENIX Security Symposium*, 2014.

[39] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Usenix Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[40] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Jun. 2007.

[41] ——, "A novel cache architecture with enhanced performance and security," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2008, pp. 83–93.

[42] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[43] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel last-level cache," IACR Cryptology ePrint Archive, Report 2015/905, Sep 2015.

[44] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013, pp. 827–838.

[45] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *IEEE Symposium on Security and Privacy (S&P)*, May 2011, pp. 313–328.

[46] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2012, pp. 305–316.