Scheduling, Isolation, and Cache Allocation: A Side-Channel Defense

Read Sprabery*, Konstantin Evchenko*, Abhilash Raj†, Rakesh B. Bobba†, Sibin Mohan* and Roy Campbell*
*University of Illinois at Urbana-Champaign †Oregon State University
Email: {spraber2, evchenk2, sibin, rhc}@illinois.edu {rajab, rakesh.bobba}@oregonstate.edu

***Abstract*—Despite the isolation mechanisms that are available to cloud service providers, like virtual machines and containers, the problem of side-channel vulnerabilities due to shared caches and multicore processors remains a threat. We present a hardware-software mechanism that improves the isolation of cloud processes in the presence of shared caches on multicore chips. Our technique can enable cache-side-channel free computing for Linux-based containers and virtual machines by combining the Intel CAT architecture that enables cache partitioning with novel scheduling techniques and state cleansing mechanisms. We evaluate our system using a CPU-bound workload and demonstrate cache-side-channel-free computation that is correct by construction. Our system allows Simultaneous Multithreading to remain enabled and does not require application level changes.**

I. Introduction

Cache-based side-channel attacks (*e.g.,* [1]–[3]) are a threat when a diverse set of users share hardware resources. Attacks focus on differences in timing while accessing processor caches. Attacks on cloud computing environments, especially Infrastructure-as-a-Service clouds (*e.g.,* [4]–[7]), show that secrets can be extracted across co-located Virtual Machines (VMs) and containers. Container frameworks such as Docker [8] that share the underlying kernel are even more susceptible to such attacks [9].

Initial attacks focused on gaming schedulers at the OS and Virtual Machine Monitor layers [2], [3], [5]. Such approaches focused on resource sharing of L1 and L2 caches within a single processor core via Simultaneous Multithreading (SMT) [1]. Multicore processors introduce cache-based side-channels via the Last-Level-Cache (LLC), thus making defenses much harder [7], [9].

Many defenses against cache-side-channel attacks in cloud environments have been proposed (*e.g.,* [10]–[22]). Existing solutions are insufficient in the following ways. Shannon's noisy-channel coding theorem states that information can be transmitted regardless of the amount of noise on the channel [23]. Hence, probabilistic defenses (*e.g.,* [10]–[12]) may decrease the bit-rate of attacks, but cannot fully eliminate them. Defenses that *eliminate* such attacks, rather than frustrate techniques employed by the attacker, are more desirable. SMT must be disabled for some solutions [12], impacting performance and utilization. In addition to a guaranteed defense, the solution must not severely impact *(i)* the performance of the applications or *(ii)* utilization of the machine. Defenses must minimize the performance cost of enforcing isolation to remain practical. Disabling hyperthreading (*e.g.,* SMT) can have a high impact on machine utilization. To the best of our knowledge, every cloud provider enables hyperthreading. Costly rewrites may be required for other defenses [13]–[15].

Solutions must be *easy to adopt*. History has shown that solutions requiring additional development time are less likely to be adopted (as shown in the Return Oriented Programming community [24]). Hardware based approaches are difficult to deploy as they require vendor support and fabrication of new chips [16]. Violating x86 semantics by modifying the resolution, accuracy, or availability of timing instructions require changes to all applications running on the machine [17]–[19]. Global compiler and page-coloring cache-partitioning [20]–[22] transforms introduce high overhead. JIT techniques allow local optimization, but performance remains problematic [25].

In this paper, *we present a hardware-software framework designed to eliminate side-channel attacks in cloud computing systems that use multicore processors with a shared LLC*. The proposed framework uses a combination of *Commodity-off-the-Shelf (COTS) hardware features* along with *novel scheduling techniques* to defend against cache-based side-channel attacks. In particular, we use Cache Allocation Technology (CAT) [26] which allows us to *partition last-level caches at runtime*. CAT, coupled with *state cleansing* between context switches and selective sharing of common libraries, removes the source of cache-timing-based side-channel attacks between different security domains. We implement a novel scheduling method as an extension to the Completely-Fair-Scheduler in Linux to reduce the overheads inherent due to state cleansing operations. Our solution provides a *transparent* way to eliminate cache-side-channel attacks while still working with *hyperthreading enabled* (SMT) systems. It works with containers and other schedulable entities that rely on the OS scheduler. To the best of our knowledge, this work is the first to provide transparent protection of applications without disabling hyperthreading.

We contribute the following through a framework that: **C1** Can eliminate cache-based side-channel attacks for schedulable units (*e.g.,* containers, vCPUs) **C2** Allows providers to exploit hyperthreading, **C3** Requires no application level changes, and **C4** Imposes modest performance overheads

II. System and Attack Model

System Model: We consider Platform-as-a-Service (PaaS) or Infrastructure-as-a-cloud (IaaS) environments. Such environments allow for co-residency of multiple appliances (*e.g.,*

containers, VMs) belonging to different security domains. We assume that the cloud computing infrastructure is using commodity-off-the-shelf (COTS) components. In particular, we assume that the servers have multi-core processors with multiple levels of caches, some of which are shared and that there is a runtime mechanism to partition the LLC.

In this paper, we use an Intel Haswell series processor that has a three-level cache hierarchy: private level 1 (L1) and level 2 (L2) caches for each core (64KB and 256KB respectively) and a last level (L3) cache (20MB) that is shared across cores. For cache partitioning, we rely on Intel *Cache Allocation Technology* (CAT) [26] allowing us to partition the L3 cache. The CAT mechanism is configured using model-specific registers (MSRs) at runtime using software mechanisms. On this specific processor model the maximum number of partitions is limited to *four* but newer generations support more [26]. Intel CAT technology has been available since 2014 and continues to be available on processor lines belonging to the Haswell, Broadwell, and Skylake micro-architectures. Our technique applies to any processor that supports partitioning of the LLC.

***Attack Background and Overview**:* While there exist many security threats to public cloud environments (*e.g.,* [27], [28]), the focus of this paper is cache-based side-channel attacks (*e.g.,* [5]–[7], [9]). An attacker process first needs to co-locate itself (*i.e.,* get assigned to the same physical server) with the victim in the infrastructure. Methods to both achieve co-residency [4] and to thwart co-residency (*e.g.,* [11], [29]–[31]) have been discussed in the literature. We assume that an attacker is able to co-locate with the victim; we focus on thwarting the actual side-channel attacks. Our framework complements approaches that frustrate co-residency attacks.

There are primarily two techniques to exploit cache based side-channels discussed in the literature: "Prime+Probe" attacks [1] and "Flush-Reload" attacks [3]. It is important to note that while these techniques are popular, they are only possible because of measurable interference. Our solution aims to prevent these specific techniques and others that leverage the cache as a side-channel by targeting the ability to measure interference across security domains.

Osvik *et al.* [1] provide an overview of Prime+Probe and Gullasch *et al.* provide an overview of Flush-Reload [3]. Zhang *et al.* summarizes the similarities in attack methods [5]. The key functionality that enables these attacks are the allocation of cache lines for Prime+Probe and the ability to flush a specific shared line for Flush-Reload attacks. Addressing these key components removes the side-channel.

***Threat Model** :* We consider both cross-core (*i.e.,* attacker and victim running on different cores on the same processor) and same-core (*i.e.,* attacker and victim running on the same core) side-channel attacks. We assume the attacker is capable of achieving co-residency with a victim, can allocate an arbitrary number of resources, and game both the cloud level scheduler (placement) and the operating system level scheduler (preemption). However, we assume that the cloud infrastructure provider is trusted. That is, while the cloud scheduler may be gamed, we assume that both the attacker and victim are authenticated with the cloud provider (e.g., for billing purposes). Additionally, we assume that the host kernel is trusted. We limit the scope of this work to addressing cache-based side-channel attacks since they can be launched by any tenant without having to exploit any vulnerabilities and without compromising the underlying software or hardware.

III. System Architecture

Our framework logically partitions a host server into an *isolated region* and a *shared region* as illustrated in Figure 1. Tenants must indicate to the cloud provider whether or not their containers need isolated execution. Containers requiring isolated execution will be executed with a separate cache partition on the host server; all other containers will be executed with a shared cache partition. The 'isolated execution' designation guarantees that processes within the designated containers will not share cache resources with *(i)* processes from any container belonging to another tenant (or security domain) or with *(ii)* processes in any container not designated for isolated execution irrespective of their ownership. Consider the deployment of a web service using a micro-service architecture. We envision a system where the tenant indicates the load-balancer (usually an HTTP reverse proxy running with OpenSSL) should run in the isolated region.

Our design leverages *(i)* Intel CAT, processor affinities (*e.g.,* CPU pinning), and selective page sharing to provide *spatial isolation* and *(ii)* co-scheduling with state cleansing to provide *temporal isolation* for designated containers.

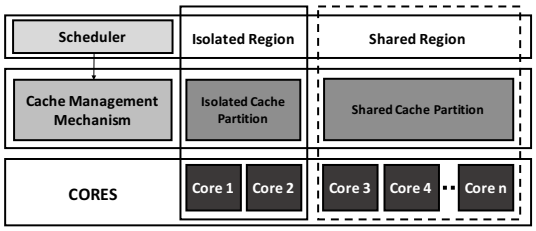


Fig. 1: System Overview

***Hardware Enforced Spatial Isolation**:* Intel's CAT [26], currently available in COTS processors, is designed to improve the performance of latency sensitive workloads by allowing the LLC to be partitioned into distinct regions. A processor can only evict cache lines within its own assigned LLC partition, thus reducing the impact of processes running on other cache regions. In particular, note that the ability to allocate cache lines (priming in Prime+Probe attacks) in a cache shared with the victim and the ability to evict cache lines being used by the victim process (flushing in Flush-Reload attacks) are key steps in cache-based side-channel attacks. Ensuring potential victim and attacker processes run on cores associated with different LLC cache partitions eliminates cross-core Prime+Probe attacks. We discuss our mitigation for the shared-memory attack vector below.

Cores in the system are associated with partitions such that each core is assigned to one and only one LLC partition.

We refer to these partition-core combinations as *isolated* and *shared regions*. The maximum number of cache partitions available with Intel CAT is fixed for a given micro-architecture. The machine used for our testing allows for up to 4 distinct partitions. The configuration of size, number of active partitions, and core to partition assignment occurs in software and can be adjusted on demand. In this paper, we evaluate a single isolated region and a single shared region, though the technique applies to multiple isolated regions.

Intel CAT, primarily designed to improve fairness of cache sharing and performance of latency sensitive workloads, allows cache hits across partition boundaries to maximize the benefits of shared memory. If the victim and the attacker processes share memory (*e.g.,* because of layered file systems used in container frameworks), an attacker can carry out a Flush-Reload attack. Since the attacker previously flushed the cache lines, a cache hit indicates that the victim executing in a different core (and LLC partition) has used or is using the library. While CAT limits the granularity of information an attacker can glean across partition boundaries, timing observations are still possible thus the side-channel is not entirely eliminated. Furthermore, attacks within an isolated partition continue to be viable. These will be addressed in the following subsections.

The partial protection against cache-side-channel attacks obtained through spatial partitioning comes at the cost of reduced LLC cache size. Fortunately, reduction in cache size has been shown to have relatively little impact on modern cloud workloads [32]. Minimal performance sensitivity to LLC size has been reported for cache sizes above $4-6$MB with modern scale-out and server workloads that are typical in cloud environments [32].

Selective Page Sharing: Hardware-assisted spatial partitioning does not eliminate cross-core Flush-Reload style attacks when the attacker and the victim share memory pages. Modern container deployments have one primary source of shared memory. We limit our discussion to Docker as it is one of the most popular choices for building container images and running them on Linux platforms, but these concepts also extend to other container frameworks. Many different containers may use the same libraries and base components, thus a way was needed to reduce disk and memory usage of the common building blocks. Docker uses Union File Systems (UFS) so processes inside of a container can access a file system composed of a stack of layers each uniquely identified by a cryptograph hash. Common layers can be shared using a given layer hash.

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

To thwart Flush-Reload style attacks across cache partitions, we eliminate cross-domain page sharing through selective layer duplication. That is, for containers requiring isolated execution, our system allows sharing of layers only among containers belonging to the same tenant (or security domain) but not otherwise. This is a reasonable trade-off as it enables isolation between different security domains while limiting the increase in memory usage. The increase in memory usage will be a function of the number of tenants running on the server rather than the number of containers. We do not prevent traditional sharing of layers for containers running within a single security domain.

For VMs, the kernel same-page merging (KSM) module in Linux, used for memory deduplication, is the main source of shared pages. However, KSM and memory de-duplication in general come with their own security risks (*e.g.,* [33]–[35]). Given the serious security concerns surrounding the use of KSM, we leave it disabled. Disabling memory deduplication is the default for commerical applications [36].

Note that selective page sharing, combined with hardware-assisted spatial partitioning, eliminates cross-core cache-side-channel attacks *across partitions*. Selective page sharing removes the ability for an attacker to measure interference after flushing a given address and CAT partitioning removes the ability for an attacker to prime a victim's cache across partition boundaries. Cache-side-channel attacks from within an isolated partition due to multi-core and SMT continue to be a threat and will be discussed next.

State Cleansing: Even with containers running in isolated partitions, an attacker allocated to the same isolated partition as the victim might be able to *(i)* observe the victim's LLC usage if scheduled to run on a different core than the victim but associated with the same partition and *(ii)* even observe the victim's L1 and L2 usage if running on the same physical core as the victim [5].

To thwart these attacks *we propose to cleanse the cache state when context switching between processes (containers) belonging to different security domains*. That is, if a process from one security domain, SD_1, runs on a core, then processes belonging to another domain, SD_2, must either run on a core assigned to a separate partition or state-cleansing of the shared resources (the shared caches) must be performed on the partition during the transition between processes from SD_1 to processes from SD_2. There currently exists no hardware instruction for per-partition cache invalidation. More details on how state cleansing can be achieved are in Section IV. However, state cleansing alone does not prevent attacks from an attacker process that is running in parallel with the victim either on the same-core through SMT, or running on a different core but in the same partition.

A naïve solution would be to assign just one core to the isolated partition, disable hyper-threading and perform state-cleansing on every context-switch. The performance cost of such an approach is unattractive. A mitigation would be to create multiple isolated partitions with a single-core assigned to each. However, the number of cache partitions is finite (4 in our case) and such an approach would further fragment

the LLC and hamper performance for the shared partition. Furthermore, many cloud workloads are multi-threaded and leverage additional cores when available. Thus, a mechanism is needed to mitigate the attack while assigning multiple logical-cores to an isolated region.

Co-scheduling for Temporal Isolation: To address this threat, we use a *novel scheduling technique for temporal separation of security domains sharing a single cache partition*. Co-scheduling container processes belonging to a given security domain across multiple processors amortizes the cost of state cleansing, but introduces additional complexity.

Scale-out workloads with many threads, those commonly deployed on cloud infrastructure, motivate this approach. As thread counts for a security domain increase, the number of threads able to run per domain at any given time will be high. This allows us to drive up utilization of cores assigned to a partition and *only flush the partition when changing to the next domain*. The complexities stem from needing to synchronize all isolated cores during domain changes, thus any implementation of co-scheduling has to guarantee an exclusion property. No task belonging to security domain SD_X can run on an isolated processor while a task from another domain, SD_Y, is running on a processor associated with the same isolated partition. Before a task from SD_X can run, a state cleansing event must occur. Multiple cores can be utilized at once within a security domain, but then state-cleansing must be performed as every core assigned to a given partition context switches to another security domain. The next security domain cannot run on any isolated processor until this process is complete.

IV. Implementation

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

Co-Scheduling: Co-scheduling *can* enforce isolation between security domains if the implementation is precise. By precise, we mean that any form of "loose" or "lazy" co-scheduling is unacceptable. For example, consider a naïve implementation of co-scheduling as outlined in Figure 2.

Figure 2 is a schedule instance in which $ORG1$ has 2 threads and $ORG2$ and 3. The example is a situation in which 2 cores are associated with an isolated partition and are running containers belonging to two security domains. These cores are two physical cores or one physical core presented as two to the operating system as is the case with SMT. The defining characteristic is the shared cache.

Consider a situation in which $Core_1$ initiates a domain transfer upon scheduling a thread from a conflicting domain, `ORG2:THREAD1` in this example. Even if the scheduler invokes a flushing event, f_1, there remains a Δt_1 during which cross-core, cross-domain attacks could be carried out. This is seen again after $Core_1$ schedules `ORG1:THREAD1` and `ORG2:THREAD3` leading to durations Δt_2 and Δt_3 during which attacks remain feasible.

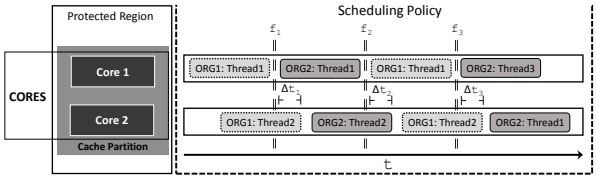


Fig. 2: Limitations of Naïve Co-Scheduling

To ensure that schedulable units belonging to different security domains run on an isolated cache partition simultaneously, we implement the core synchronization protocol shown in Figure 3. The protocol works by making the first core in an isolated partition a *leader* core. The leader core is responsible for initiating domain changes, synchronizing cores, and state cleansing. All isolated cores only schedule tasks belonging to the active security domain. Note that while only 2 cores are shown in Figure 3, the approach works with any number of cores. In Section V we evaluate the protocol with 4 cores assigned to an isolated partition.

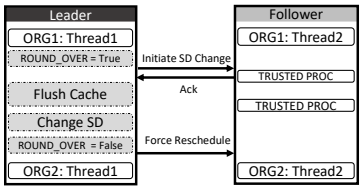


Fig. 3: Strict Co-Scheduling Protocol

Isolated cores rely on two pieces of shared state to achieve strict synchronization. The leader core is the only core that can modify the state. The `ROUND_OVER` variable indicates to follower cores that a domain change is about to occur. A timer on the leader core initiates a domain change by modifying this variable and invoking the `__schedule` function on the leader core. The `change domain` event fires every `sysctl_sched_min_granularity`, a configurable parameter exposed to administrators on Linux based systems to control system responsiveness. After setting the `ROUND_OVER` variable to `true`, the leader core issues a reschedule command via an Inter-Processor Interrupt to follower cores and waits for them to send back an acknowledgment. The acknowledgment is performed within the `__schedule` function on follower cores. When the `ROUND_OVER` variable is set, partitioned cores can only run *trusted processes*. These are only kernel tasks, including: `ksoftirq`, `watchdog`, and the idle task. Ensuring such processes can run prevents deadlocks due to `watchdog` timeouts.

After receiving an acknowledgment back from all follower cores, indicating they are no longer running tasks belonging to any security domain, the leader then flushes the cache and changes the active security domain to next domain. We iterate through a kernel linked-list for round robin style switching of security domains on the leader core. Run-queue checking is performed to ensure a domain with runnable tasks is chosen.

Having chosen the next domain, the leader core sets `ROUND_OVER` to `false` and again issues a reschedule command to follower cores. The `__schedule` function will eventually be invoked on the follower cores, but we use the reschedule command to reduce the idle time of follower cores. This protocol corrects the problem presented in Figure 2 resulting in "strict" co-scheduling as seen in Figure 4.

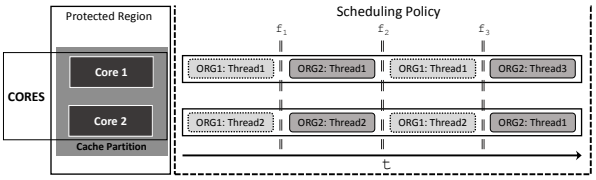


Fig. 4: Synchronized Co-Scheduling

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

One way to implement state cleansing is for the user process to invoke the *CLFLUSH* instruction, which can evict cache lines corresponding to a linear virtual address and can be invoked from user space. The kernel can invoke *CLFLUSH* on the entire virtual address space. While this is guaranteed to work across processor generations, this approach is too costly, taking up to 10x the kernel timeslice on applications we tested. An optimization is to do it only on valid virtual addresses for the task being switched out as was done in [37]. However, this can still be a large range compared to the size of a cache partition $(4 - 6\text{MB})$.

Another approach is to create an *eviction set* – a set of addresses which when loaded are guaranteed to evict the entire cache partition. However, the memory address to cache line mapping is proprietary and subject to change across processor generations. Cache-side-channel attacks have to contend with this challenge and address it by reverse-engineering the memory address to cache line mapping for a given micro-architecture (*e.g.,* [7]). Apart from the one-time cost of reverse engineering, the cost of this approach is equal to loading memory the size of the cache partition from a linear address space. To evaluate the performance of such an approach, we use the memory load method.

We perform state cleansing anytime the security domain is changed, as shown by the protocol in Figure 3. To reduce performance impact in the case of other domains lacking runnable threads (due to blocking on I/O, *etc.*), flushing is only performed when the active security domain changes. If only a single security domain has runnable tasks, no flushing will occur.

Selective Page Sharing: Docker uses a UFS to present a unified view of the several different layers. We modify the AUFS storage driver. AUFS is mature and supports all of Docker's storage feature set. In a UFS, multiple directories on the host are unified in a single directory called a *union mount*, without replicating the actual contents of individual layers. Contents of all layers become visible at the *union mount*. Docker keeps a single copy of each layer on the host filesystem and AUFS mounts all layers to a single *union mount* point, which becomes the container's root file system. We modified the AUFS driver to have separate copies of each layer for each security domain. In this way, no two containers belonging to different security domains share any common layers.

V. Performance Evaluation

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

The machine is configured as outlined in Section II. We allocate 2 physical cores and 4 logical processors to an isolated cache region. The cache region is 4MB (4 cache ways out of 20 available on the system). Each security domain is assigned 4 threads, and the number of domains is varied from 2 to 8. Each domain consists of 4 CPU bound tasks, 1 for each logical processor. Measurements are taken at an interval of once per second for 100 seconds. Figure 5 shows the overhead of our system running while running 8 security domains in an isolated region. System utilization when our system is disabled is very near 100% in each case, thus is not shown in Figure 5. Similar numbers are seen when running 2 and 4 security domains. The result indicates overhead for a single logical processor are a function of the system and not of the number of security domains assigned to a partition. Follower cores can be seen idling during domain changes with overhead around 20%. From our tests, we know that flushing significantly increases the performance penalties. The leader core spends the most time executing in system space due to its responsibility to change domains and synchronize cores, so this was to be expected. In the future, we will investigate mechanisms to reduce system time on the leader core and idle time on follower cores. Hardware based per-partition flushing mechanisms, such as an enhanced `WBINVD`, would significantly reduce these overheads, though our approach would still be needed to enable multiple logical processors in an isolated region. Because the isolated region suffers $\approx$ 20% reduction in utilization as visible to the provider, the impact to tenants within the isolated region will be strictly less. Cloud computing benefits from over-subscription making these gains possible. Impacts to utilization are amortized across the security domains assigned to a given isolated region. The reduction in time spent in userspace per domain as measurable by the tenant is small. There is a reduction of 9.82%, 2.97%, and 1.68% for 2, 4, and 8 security domains in a single isolated regions respectively. The only performance impact to schedulable units in the non-isolated region is due to the reduction in cache sized from 20MB to 16MB. Reduced cache sizes have little impact on cloud workloads [32].

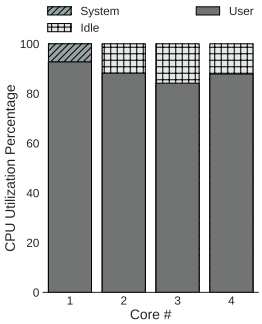
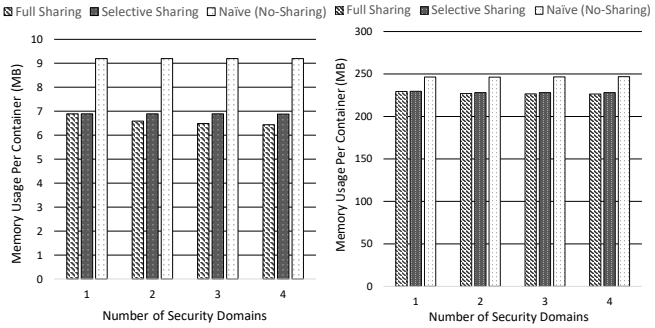


Fig. 5: CPU Overhead in Isolated Region



(a) Redis (b) Tomcat

Fig. 6: Full Sharing vs. Selective Sharing

Impact of Shared Memory Reduction: By enabling selective sharing of base layers in Docker, we expect an increase in the memory footprint of containers as there are multiple copies of certain pages that would otherwise be shared. To understand the memory growth vs. the number of security domains, we ran 2 experiments each with a web server (Apache Tomcat) and an in-memory database (Redis). We measure the amount of memory being used on a per container basis. To make sure that the code is resident in memory before we take measurements, we send 100 requests to the Apache Tomcat servers and added 1000 random key-value pairs to each Redis server.

We measure the average memory increase as we raise the number of security domains from 1 to 4 across 50 runs. Each security domain has 5 containers. The result is compared against the same number of containers running without modifications on the same host. These measurements are also compared against a naïve solution in which sharing is disabled all together. Figures 6a and 6b show that memory usage for Redis increases ≈ 0.45 MB per container in the worst case (4 security domains with 5 containers each) and for Apache Tomcat only ≈ 1.71 MB per container in the worst case. From the overall memory consumption of these processes (≈ 7 and ≈ 230 respectively) it is clear that the added memory cost per container is marginal. Selective sharing that enables layer sharing within a security domain provides substantial improvements over disabling sharing entirely.

VI. Related Works

Probabilistic [10]–[12] solutions are insufficient as they cannot fully eliminate the source of cache-based side-channel attacks. Existing approaches achieve single core isolation by disabling hyperthreading [12], [38]. Disabling hyperthreading reduces the throughput of not only the "secure" workload, but the entire machine. Cutting whole-machine utilization by even 20% (the impact of hyperthreading in 2005 [39]) is too high for cloud computing. StealthMem is able to enable hyperthreading, but the authors do not sufficiently address cross thread scheduling issues [14] and require application developers to make code changes. CATalyst [15] uses Intel's CAT technology to assign virtual pages to sensitive variables instead of software-based page coloring. Application rewrites are necessary for these approaches [13]–[15], increasing the cost of adoption. Costly hardware changes [16] face similar adoption challenges. Hardware approaches changing timing instruction behavior change the semantics of the architecture [17]–[19], forcing application changes. Cache coloring [20]–[22] approaches have impractically high overheads.

CACHEBAR [12] defends against Flush-Reload attacks by duplicating memory pages on access from separate processes. CACHEBAR provides only probabilistic guarantees for defense against a Prime+Probe attack and disables hyperthreading. Solutions like Nomad [11], while probabilistic, complement our approach. Nomad works in the cloud scheduler to reduce the co-residency of different security domains. Our solution could be used in conjunction with Nomad to provide hard isolation when co-residency restrictions are not possible.

VII. Conclusions

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

VIII. Acknowledgments

This material is funded in part by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11- 2-0084 and by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE1144245. We thank Rodrigo Branco for helpful discussion and comments, as well as the anonymous reviewers.

REFERENCES

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

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

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

[4] 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 *Proc. of the 16th ACM conf. on Computer and communications security*. ACM, 2009, pp. 199–212.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *Proc. of the 2012 ACM conf. on Computer and communications security*. ACM, 2012, pp. 253–264.

[28] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[29] Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Virtual machine allocation policies against co-resident attacks in cloud computing," in *Communications (ICC), 2014 IEEE Int. conf. on*. IEEE, 2014, pp. 786–792.

[30] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang, "Incentive compatible moving target defense against vm-colocation attacks in clouds," in *IFIP Int. Information Security conf.* Springer, 2012, pp. 388–399.

[31] Y. Azar, S. Kamara, I. Menache, M. Raykova, and B. Shepard, "Co-location-resistant clouds," in *Proc. of the 6th Edition of the ACM Workshop on Cloud Computing Security*, ser. CCSW '14. New York, NY, USA: ACM, 2014, pp. 9–20. [Online]. Available: http://doi.acm.org/10.1145/2664168.2664179

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

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

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

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

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

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

[38] M. Ahmad, "Cauldron: a framework to defend against cache-based side-channel attacks in clouds," Master's thesis, 2016.

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