CAULDRON: A FRAMEWORK TO DEFEND AGAINST CACHE-BASED SIDE-CHANNEL ATTACKS IN CLOUDS

BY

MOHAMMAD AHMAD

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisers:

Assistant Professor Rakesh B. Bobba
Professor Roy H. Campbell

ABSTRACT

Cache-based side-channel attacks have garnered much interest in recent literature. Such attacks are particularly relevant for cloud computing platforms due to high levels of multi-tenancy. In fact, there exists recent work that demonstrates such attacks on real cloud platforms (*e.g.,* DotCloud). In this thesis we present `Cauldron`, a framework to defend against such cache-based side-channel attacks. `Cauldron` uses a combination of smart scheduling techniques and microarchitectural mechanisms to achieve this goal. We are able to demonstrate improved defenses against both cross-core side channel attacks that target shared caches as well as same-core attacks. Furthermore, `Cauldron` is transparent to the user – requiring no modification (or even recompilation) of users' application binaries by integrating directly with the popular container runtime framework, Docker. Preliminary evaluation results show that the proposed approach is effective for cloud computing applications.

To my parents, Afzal and Qudsia Mujtaba, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisers, Assistant Professor Rakesh Bobba and Professor Roy Campbell for their guidance and support during the course of my degree.

I would like to thank Konstantin Evchenko, Dr. Sibin Mohan, Read Sprabery, Abhilash Raj and Rodrigo Branco, all of whom collaborated with me on this project.

I would like to thank my family for their love and encouragement. In particular, I would like to acknowledge Sadia Ali and Syed Ali for their advice and support at every step of the journey.

I would like to thank my closest friends at UIUC, Zainab Rahil, Muhammad Huzaifa, Aneeq Zaman, Rashid Tahir, Zain Yousaf and Shayan Saeed for making my time here so enjoyable. Lastly, I would like to thank my undergraduate advisor, Dr. Fareed Zaffar for encouraging me to pursue graduate studies. I remain indebted to him for his support.

My degree has been funded by AFOSR and AFRL through the University Center of Excellence on Assured Cloud computing, by DOE through the SEL SDN Project and by the Sohaib and Sara Abbasi Fellowship.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Contributions 2

CHAPTER 2 BACKGROUND 4
2.1 Virtualization Technologies 4
2.2 Cache Architecture 7
2.3 Cache-based Side-Channel Attacks 10

CHAPTER 3 RELATED WORK 14
3.1 Cache-based Side-Channel Attacks 14
3.2 Defenses Against Cache-based Side-Channel Attacks 15

CHAPTER 4 APPROACH 18
4.1 System Model 18
4.2 Attack Model 19
4.3 Cauldron Design 19
4.4 Cauldron Implementation 23

CHAPTER 5 EVALUATION 25
5.1 Security Evaluation 25
5.2 Performance Evaluation 27

CHAPTER 6 FUTURE WORK 30

CHAPTER 7 CONCLUSION 32

REFERENCES 33

CHAPTER 1

INTRODUCTION

Cloud computing has rapidly gained adoption over the last few years as organizations have tried to reduce the complexities and costs associated with deploying and maintaining a reliable computing infrastructure. It is equally popular with small companies and start-ups that have relatively fewer resources as with large organizations including the US federal government. In fact, the federal government has instituted a CloudFirst policy and expects to spend about a quarter of its \$80 billion IT budget on cloud computing services [21]. Furthermore, a report by the Carbon Disclosure Project on cloud computing shows that large US companies (those with more than \$1 billion in revenues) are set to increase their adoption of cloud computing from 10% to 70% of their IT spending over the next decade and realize a savings of \$12.3 billion in energy costs alone. On the other hand, many cloud service companies have developed a "multi-tenant" policy where jobs from different clients can be simultaneously, yet transparently (to each other), executed to better utilize the underlying hardware resources.

While Infrastructure-as-a-Service (IaaS) clouds enabled by hardware virtualization have been dominant, Platform-as-a-Service (PaaS) offerings enabled by operating system (OS) level virtualization techniques are fast emerging as a lightweight and high-performance alternative [23, 25]. OS virtualization technology, hereafter generically referred to as *Containers*[1], provides a lightweight execution environment with better performance and less overhead than VMs [56, 29].

This rapid adoption and the multi-tenant nature of cloud computing provide a greater incentive for attackers to target such systems. Recently, cache-based side-channel attacks have received much attention. It has been shown

[1]Note that user-space instances in a virtualized OS are refereed to as Containers in many technologies, other names such as virtual engines, virtual private servers and jails are also in use.

that such attacks are capable of extracting fine-grained information such as cryptographic keys even in the cloud environment [43, 35, 57, 59]. Some of the attacks have also been demonstrated on public cloud infrastructures [60]. In fact, with the adoption of *lightweight virtualization* methods such as Linux containers [25, 23, 18] such attacks can become easier.

In this thesis, we present the `Cauldron` framework to defend against cache-based side-channel attacks in cloud environments based on the aforementioned lightweight virtualization, *viz.,containers*.
`Cauldron` is:

i) easy to deploy,

ii) transparent to end users and

iii) provides increased security against such attacks.

The framework uses a novel combination of hardware and software mechanisms to achieve these goals. Specifically, `Cauldron` does not require developers to modify their applications or libraries. This makes the application development process much easier and it can also be used for legacy applications. In this thesis, we also demonstrate how to integrate our framework into the Docker [23] container runtime. The solutions proposed in this thesis are not specific to Docker and are generally applicable to other OS virtualization frameworks (container runtimes) such as rkt and LXC [18, 25]. In fact, they can be applied to cloud environments employing hardware virtualization technologies such as Xen and KVM [27, 10].

Previous approaches were either limited to defending against same-core attacks [61, 52], limited by the performance of software-based cache-partitioning [39, 49, 31] or required modifications to applications [39, 42] (refer to Chapter 3 for more details). In contrast, `Cauldron` defends against both same-core and cross-core attacks, doesn't require changes to end-user applications or libraries, and is easy to deploy while incurring only reasonable overheads.

1.1 Contributions

We design and implement `Cauldron`, a framework to defend against cache-based side-channel attacks in the cloud environment. In our security evaluation, in Chapter 5, we show that it is able to protect against both same-core and cross-core cache-based side-channel attacks. By implementing `Cauldron` as a loadable kernel module we are able to develop an easy to deploy solution. Furthermore, we design our framework to ensure that it does not require any changes to be made to user applications, containers or libraries and integrates cleanly with a popular container runtime, Docker. Finally, we show that `Cauldron` is able to achieve all of the aforementioned advantages using commodity off-the-shelf hardware, while incurring only a reasonable overhead.

The remainder of this thesis is structured as follows: in Chapter 2 we provide a background on containers, the Intel cache architecture along with an introduction to cache-based side-channel attacks. Related work is discussed in Chapter 3. In Chapter 4 we present our system and attack models along with the `Cauldron` implementation. We evaluate the `Cauldron` framework in Chapter 5. A discussion on the advantages, limitations and future work is covered in Chapter 6 and we conclude in Chapter 7.

CHAPTER 2

BACKGROUND

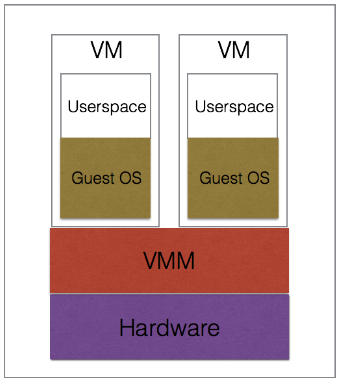
Although virtualization technologies attempt to stop any leakage of information across clients sharing the same underlying hardware, prior work, such as [59, 60, 43, 35] has shown that these virtualization technologies are susceptible to cache-based side-channel attacks. In order to fully understand such attacks, in this chapter, we provide a background of virtualization technologies, with a focus on containers, the Intel cache architecture and cache-based side-channel attacks.

2.1 Virtualization Technologies

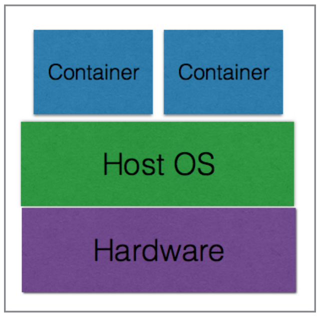
Today's Infrastructure as a Service (IaaS) clouds are powered by hardware virtualization while, on the hand, Platform as a Service (PaaS) offerings are increasingly adopting containers for tenant isolation. We discuss both virtualization technologies in detail below.

2.1.1 Hardware Virtualization

In IaaS clouds, such as Amazon EC2 [1], Google Compute Engine [8] and Azure Virtual Machines [3], clients can spin up virtual machines (VMs) that attempt to emulate physical machines. As shown in the Figure 2.1a, each VM has its own operating system along with libraries and application binaries. Multiple VMs run on top of a Virtual Machine Monitor (VMM) which in turn runs on physical hardware. The VMM is responsible for multiplexing hardware resources, such as CPU and memory, between the client VMs that it hosts. Popular VMM implementations include Xen and KVM [27, 10].



(a) Hardware virtualization stack



(b) OS virtualization stack

Figure 2.1: Comparison between the hardware and OS virtualization stacks

2.1.2 Containers

While hardware virtualization remains popular, public PaaS offerings are increasingly being powered by container runtimes [60]. In such PaaS offerings client jobs are run inside containers, as opposed to VMs. Containers can be seen as light-weight VMs. As shown in Figure 2.1b, unlike VMs, containers do not have their own OS and instead all containers running on the same underlying host, *i.e.*, co-located containers, share the host OS. Furthermore, where appropriate, libraries are also shared between co-located containers to increase memory efficiency.

The increase in popularity of containers can be attributed to their light-weight nature because of which, the time to start a container is on the order of milliseconds in contrast to VMs which can take several seconds to launch. Similarly, containers also incur a lower performance overhead when compared to VMs [56, 29]. Powerful container runtimes, such as Docker [23] and rkt [18] have made containers easy to manage, build and deploy. While, container registries, like Docker Hub [7] and Quay [16], have made containers easy to share, further increasing adoption.

In order to virtualize the OS, a container runtime needs to be able to allocate resources to containers and be able to isolate the view of the system from within containers. To achieve these two goals, container runtimes leverage Linux kernel functionality, namely, control groups (cgroups) [38] and kernel namespaces [14] to build containers. Cgroups provide a resource management solution while namespaces enable resource isolation. To provide added security, a number of container runtimes support mandatory access control policies by either integrating with Security Enhanced Linux (SELinux) [19] or AppArmor [2].

We discuss cgroups and namespaces in greater detail below.

Linux Control Groups (cgroups)

Cgroups enable fine-grained resource management of user-defined groups of processes [38]. Cgroups present a powerful abstraction for system administrators by allowing them to limit the amount of system resources, such as memory and disk I/O, available to different user-defined groups of processes. Cgroups were merged into the Linux kernel mainline with the kernel version 2.6.24 in early 2008 [6].

Cgroups can be created, deleted and updated using the cgroup virtual filesystem. Each cgroup has an associated task set that includes the processes that are part of the cgroup and on which the cgroup resource allocations apply. Cgroups are arranged hierarchically, where a child cgroup inherits properties from its parent cgroup. Take for example, a cgroup that is assigned a 40% share in CPU time and has two child cgroups both of which are assigned equal shares of CPU time. In such a scenario, the child cgroups will each be allocated 20% of the actual CPU time. There can exist multiple cgroup hierarchies in system, where each hierarchy is assigned to a different system resource [17]. For instance, separate cgroup hierarchies can be maintained for memory and CPU time. Note that, processes not assigned to any cgroup are added to the root cgroup.

As mentioned earlier, to manage resource allocation, container runtimes leverage cgroups. A separate cgroup is created for each container. All processes running inside a container are added to the task set of its associated cgroup. Generally, four system resources, CPU, memory, disk I/O and the network are managed using cgroups. The memory cgroup tracks and limits the memory pages being used by a cgroup. Weights are assigned to each cgroup in the CPU subsystem hierarchy to determine the ratio in which CPU time will be divided between cgroups. The disk I/O cgroup subsystem can be used to track and throttle I/O operations performed by cgroups. Finally, the network cgroup enables the assignment of traffic generated by a cgroup to different traffic priority classes.

Namespaces

Namespaces isolate the view of processes within a namespace of a certain global resource [14]. Any updates made to the global resource by a process are only visible to other processes within the same namespace but remain invisible to processes outside of the namespace [11]. PID, Mount, IPC, User, UTS and Network namespaces are supported by the Linux kernel.

All six of the supported namespaces are utilized to create an isolated view of the system for each container. By creating a separate PID namespace for each container, processes within a container are limited to viewing other processes belonging to the same container. An isolated view of the filesystem hierarchy is created using the Mount namespace. The UTS namespace is used to create a separate hostname identifier for each container. Separate interprocess communication resources, such as IPC semaphores and message queues are created using the IPC namespaces. By leveraging the Network namespaces a separate network stack is created for every container. This includes separate routing tables, IP table rules and network interfaces. Finally, User namespaces are utilized to allow different mappings of a process's user ID inside and outside of a container [20].

2.2 Cache Architecture

There can be many types of caches present in a given system, such as CPU, application and database caches. In this thesis we only concern ourselves with CPU caches and we hereafter refer to CPU caches as simply, *caches*. Since caches are central to the attacks and defenses discussed in this thesis, we provide an introduction to them here.

For data to be processed, it needs to brought into the processor. Processors are capable of processing data much faster than it can be read from main memory. This makes input to the processor the bottleneck [13]. This forms the motivation for caches, which are a small, high-speed memory that stores recently accessed data from main memory. Their goal is to reduce the time for the processor to access memory. In order to be effective, caches exploit spatial locality *i.e.,* if a memory location is accessed, memory locations around that location are likely to be accessed in the near future. Keeping this is mind, each entry in the cache is 64 (or 32) bytes and is referred to as a *cache line.*

Caches fetch complete cache lines from memory instead of single memory locations.

As compared to main memory, caches are much smaller in size. This is because caches are more expensive to build. Since caches are smaller than the main memory, a mapping from lines in main memory to locations in the cache is required. One approach is to allow any line in memory to map to any location in the cache. Such an approach requires that when searching for the presence of a memory address in the cache, the complete cache needs to be checked. The approach is made infeasible by the need for a very large number of comparators in its implementation [13]. Caches implementing this approach are referred to as fully-associative caches. At the other end of the spectrum are directly-mapped caches where each line in memory maps to exactly one location in the cache. Although, this approach is easier to implement, the in-flexibility in mapping significantly hampers performance. Set-associative caches lie somewhere in between fully-associative and directly mapped caches.

The set-associative approach, divides the cache into equally sized sections, referred to as *ways*. The number of *ways* supported in the cache is referred to as the cache-associativity. A line in memory maps to exactly one location in each of the cache *ways* and together those locations are referred to as a cache set. For instance, in a 4-way set-associative cache, each line in memory maps to four locations in the cache, one in each cache *way*. By allowing a more flexible mapping scheme, set-associative caches are able to outperform directly mapped caches. In particular, they reduce *conflict misses i.e.,* cache misses that occur when there are empty spaces in the cache, just not in the required cache set. As compared to fully-associative caches, set-associative caches are cheaper to implement.

Due to the advantages discussed above, caches in modern Intel processors are set-associative. In the following subsections we discuss the cache hierarchy and support for cache partitioning.

2.2.1 Cache Hierarchy

As shown in Figure 2.2, today's multi-core processors typically have a multi-level cache hierarchy with a shared last-level cache and private lowest-level

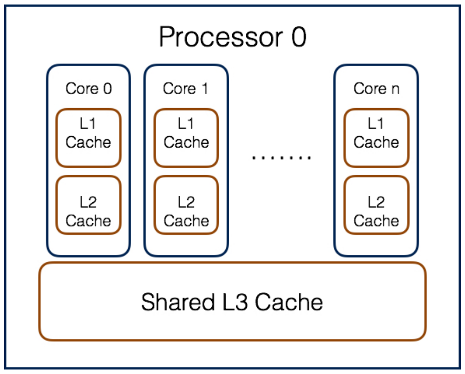


Figure 2.2: Cache Hierarchy Overview

and intermediate-level caches. In this thesis, we limit ourselves to modern Intel architectures that have a three level hierarchy [24]. The first level cache, L1, is core-private *i.e.,* each core has a separate L1 cache. Furthermore, the L1 cache is divided into two parts, the L1 instruction cache and the L1 data cache. The instruction cache is responsible for caching program instructions while data is stored in the data cache. Similar to the L1 cache, the second layer, L2 cache is also core-private but unlike the L1 cache, the L2 cache is integrated, meaning both the instructions and the data share the same cache. Finally, the L3 or last level cache (LLC) is shared among all the cores. The L3 cache is integrated as well as inclusive, meaning all the cache lines present in the lower level caches, L1 and L2, are also present in the L3 cache. As we move from the L1 to the L3 cache, the size of the cache increases but at the same time access latency increases as well.

In the remainder of this thesis we assume this 3 level cache architecture. However, the proposed defense is generically applicable.

2.2.2 Intel Cache Allocation Technology

As discussed in Section 2.2.1, modern Intel cache architectures consist of a shared L3 cache. The sharing of the L3 cache between all the cores in a processor means that a misbehaving thread *i.e.,* a thread that continuously brings new data into the L3 cache but does not reuse any data brought into the cache, can negatively impact the performance of other threads running on the system along with the cache efficiency. Other threads are negatively impacted because the misbehaving thread, continuously evicts the data they brought into the cache.

Considering cases similar to one discussed above, Intel introduced the Cache Allocation Technology (CAT) which allows the partitioning of the L3 cache between different cores on a processor. Partitions are assigned on a per-core basis and threads are limited to evicting cache lines from the partition allocated to the core that they run on. By assigning a separate partition to a misbehaving thread, the negative impacts on both, other threads in the system and cache efficiency, can be mitigated.

Partitions are divided along cache *ways*. Classes of Service (COS) are assigned *ways* of the L3 cache that they can allocate from. This assignment is done using bit masks written to Model Specific Registers (MSRs). COS can be allocated overlapping cache *ways*. Processor cores are then assigned to COS that determine which cache ways are available to threads running of the cores. No changes are required to the OS in order to benefit from CAT. Intel requires that a minimum of 2 cache *ways* are assigned to a COS. Importantly, CAT configuration can be changed dynamically by the MSRs as described above.

On the latest Haswell architecture, the number of partitions is limited to four [42]. Furthermore, since CAT has been developed with quality of service (QoS) in mind, threads are able to get a hit for data present in different partition of the L3 cache. Both these limitations impact `Cauldron` and we discuss our solutions to them in Section 4.3.

2.3 Cache-based Side-Channel Attacks

A side-channel is an information leakage channel that can be used by the attacker to gain insight into the victim application. An *access-driven* side-channel attack consists of an attacker, co-located with the victim, that monitors shared microarchitectural components, such as caches, to gain useful information about the application the victim is running [59]. In this thesis we focus on access-driven side-channel attacks that leverage caches, as they have been shown to be able to extract fine-grained information from across VM or container boundaries (*e.g.,* [46, 50, 60, 59, 37, 32, 43, 35, 57]). The attacks can be divided according to the cache level they target and the attack technique they use. Some of the attacks target the L1 cache, while others take advantage of the shared L3 cache. We leave the details of such attacks to Chapter 3. Instead, here we focus on the high-level attack techniques leveraged by cache-based side-channel attacks, namely, the FLUSH+RELOAD [57] attack and the PRIME+PROBE [46] attack.

Following is a high-level description of these attacks techniques.

2.3.1 FLUSH+RELOAD

The FLUSH+RELOAD attack leverages shared libraries with the victim to launch a cache-based side-channel attack. Libraries are often shared in PaaS clouds between co-located containers. Similarly, if the deduplication feature is enabled in a VMM, libraries can be shared between VMs.

The attacker first identifies a code segment that she is interested in monitoring in a library shared with the victim. The attacker then proceeds by repeating the following three steps to launch the attack.

1) *Flush*: the attacker flushes the code segment from the cache,

2) *Wait*: then waits for a short period of time,

3) *Reload*: the attacker reloads the code segment, measuring access time.

In step 1, to flush the code segment, the attacker can use the “clflush” instruction [5] available on x86 processors. The “clflush” instruction, flushes the cache line passed to it as an argument from all levels of the cache hierarchy. The attacker is only required to pass the virtual address of the code segment as a parameter to the “clflush” instruction. The attacker then waits a short period of time as specified in step 2 and then attempts to reload the code segment in step 3. By measuring the time it takes the attacker to reload the code segment, the attacker can determine if it was already present in the cache or if it was loaded from memory. This determination is made possible by the noticeable difference in memory and cache access times. On the machine we use for our evaluation, Intel(R) Xeon(R) CPU E5-2618L v3 @ 2.30GHz, we observe main memory access times to be around 300 CPU cycles where as L3 access times are close to 100 CPU cycles. Since the attacker flushed the code segment from the cache in step 1, if the attacker observes an access time representative of a cache access in step 3, the attacker can conclude that the victim had loaded the code segment into the cache while the attacker waited in step 2. On the other hand, an access time typical of a memory access in step 3 means that the victim did not load the code segment into the cache while the attacker waited.

2.3.2 Prime+Probe

Unlike the Flush+Reload attack, the Prime+Probe attack does not rely on shared libraries. This places the additional burden on the attacker to determine the target cache line of the victim. The attack follows repeated iterations of a similar three step procedure to that described earlier.

1) *Prime*: the attacker loads its own data into the cache,

2) *Wait*: then waits for short period of time,

3) *Probe*: then reloads the data from step 1, measuring the access time.

In step 1, the attacker evicts all of the victim's lines in the cache by loading a cache-sized chunk of data. The attacker then waits for a short period of time after loading the data. Then, in step 3 the attacks reloads the same data it had previously loaded in step 1. By measuring the time it takes to reload each of the lines, the attacker can determine which of the cache sets were touched by the victim while the attacker waited in step 2. This is because when a line is read by the victim, it would replace a line loaded by the attacker.

After repeated iterations of the aforementioned steps, the attacker can develop an access pattern for each of the cache sets. As an optimization, the attacker can then limit itself to monitoring cache sets that have an access pattern similar to what is expected from the target cache line of the victim. The Prime+Probe attack can be implemented to target the L1 cache, in which case the attacker and victim need to be running on the same core. Furthermore, the attacker is then required to regularly preempt the victim which can make the attack infeasible. The attack can also be implemented to target the shared L3 cache, in which case, the attacker and victim can be running on different cores of the processor and the attacker is no longer required to preempt the victim.

The Prime+Probe attack suffers from greater noise as compared to the flush+reload attack. This is because, in the former, the attacker's cache line(s) can be evicted by any of the victim's lines that map to the same cache set. Therefore, there is no guarantee, when the attacker observes a higher access time, that the victim actually accessed the *target* cache line. Such false positives do not occur when launching the flush+reload attack where, by leveraging shared libraries and the the "clflush" instruction, the attacker is certain that a lower access time in the *reload* step, indicates that the target cache line was accessed by the victim. An advantage of the Prime+Probe attack is that it is more widely applicable as it does not rely on shared binaries.

CHAPTER 3

RELATED WORK

Side-channel and covert-channel attacks are well known in literature. In this chapter, we limit our discussion to cache-based side-channel attacks and defenses against them.

3.1 Cache-based Side-Channel Attacks

Many cache-based side-channel attacks ranging from coarse-grained to fine-grained, from private lab setting to public cloud settings and from attacks on VMs on a single core to cross-core have been discussed in literature (*e.g.,* [46, 50, 60, 59, 37, 32, 43, 35, 57]). PRIME+PROBE and FLUSH+RELOAD are common techniques used to launch such attacks. We discussed both of these techniques in detail in Section 2.3.

Earlier attacks in the literature have focused on non-cloud environments. Osvik *et al.* [46] were the first to introduce the PRIME+PROBE attack and demonstrate its feasibility by launching an efficient side-channel attack across process boundaries targeting AES. Percival [48] describes a similar attack on RSA that leverages hyper-threading. Side-channel attacks in cloud environments have received much attention over the last few years (*e.g.,* [50, 60, 59, 37, 32, 43, 35]). Zhang *et al.* [60] were the first to demonstrate a side-channel attack capable of extracting fine-grained information across containers on a public PaaS cloud. The attack used the FLUSH+RELOAD attack technique. In addition, the authors were able to show that an attacker can achieve co-location with a victim container on public PaaS offerings.

Prior to presenting side-channel attacks across containers, Zhang *et al.* [59] again leverage cache-based side-channels but this time the attack is launched across VMs. Although the attack is launched in a lab setting, the fact that they were able to extract private across VM boundaries, highlights the significant risks posed by cache-based side-channels. They demonstrate the attack on top of the Xen hypervisor. Their approach uses the core-private L1 cache and therefore places the additional burden of frequent preempting of the victim VM on the attacker. The work serves as a follow-up of Risentpart *et al.* [51] where the authors show how to achieve co-location with a victim VM and build on the PRIME+PROBE technique to establish coarse-grained cache-based side-channels across VMs on the public Amazon EC2 cloud [1].

Similar to [59], a number of previous attacks had built timing channels using the core-private L1 caches, this required the attacker to be able to frequently preempt the victim VM. Varadarajan *et al.* [52] showed that such attacks could be thwarted by making minor changes to the VMM scheduler. On the other hand, attacks leveraging the FLUSH+RELOAD technique do not require frequent preemption of the victim VM. Yarom *et al.* [57] and Irazoqui *et al.* [36] demonstrate such attack with the victim and attacker VMs running on different cores on top of the VMware ESX hypervisor. A key limitation of such attacks is that they are dependent on memory deduplication, a feature now disabled by default in all popular hypervisors (*e.g.,* Xen, KVM, VMware ESX).

In an attempt to overcome the aforementioned limitations, recent work by Fangfei *et al.* [43], Irazoqui *et al.* [35] introduced cross-core shared cache side-channel attacks using the PRIME+PROBE attack technique. Both the attacks leverage huge pages to be able to virtually address the complete LLC. The attacks showed that cryptographic keys could be extracted across VMs using the shared LLC without relying on the memory deduplication features.

3.2 Defenses Against Cache-based Side-Channel Attacks

Many defenses against cache-based side-channels have also been proposed. Varadarajan *et al.* [52] introduce a scheduler-based defense against cross-VM side-channel attacks by incorporating a minimum run-time guarantee (MRT) along with a per-core state-cleansing action. Once the MRT of VMs is increased, the attacker is unable to preempt the victim VM frequently enough so as to extract any fine-grained information. The work by Vardarajan *et al.* builds on the Düppel system [61] where tenant VMs clear the L1 cache to protect themselves against cache-based side-channels. However, both of these methods do not address cross-core side-channel attacks.

Software-based cache-partitioning, *viz.,* page-coloring [58], is used to isolate tenants in [39, 49, 31]. Raj *et al.* [49] evaluate the use of page coloring and cache-aware core assignment (gang scheduling) albeit as independent techniques. Godfrey *et al.* [31] employ software-based cache partitioning together with cache flushing to defend against cache-based side-channel attacks. However, most of these techniques are limited by the performance of software-based cache partitioning. In addition, page-coloring based techniques do not support huge pages [9], an important requirement for many big-data cloud applications.

STEALTHMEM [39] overcomes the performance issue by allocating a secure page to each core to protect small amounts of sensitive data belonging to the VM running on the core. CATalyst [42] uses Intel's CAT technology in a similar manner. However, similar to STEALTHMEM, as CATalyst provides a limited number of secure pages to each VM it consequently is limited to protecting small amounts of sensitive data. More importantly, both STEALTHMEM and CATalyst *require modifications to applications* as sensitive variables need to be identified and annotated to avail the protection of secure pages. In contrast, we aim to provide isolation for the *entire container* and our approach *doesn't require any changes to application code.* This makes it much easier for existing applications to use `Cauldron`.

As described in Section 2.3, cache-based side-channel attacks require accurate timing information to differentiate between cache hits and misses. This dependence on timing information has motivated a class of defenses that reduce the accuracy of timing information available to tenants. Vattikonda *et al.* [53] along with the StopWatch [41] and TimeWarp [44] systems are all implementations of this idea. By fuzzing the timing information available to tenants they are able to successfully thwart cache-based side-channel attacks. However, such approaches negatively affect legitimate uses of accurate timing information.

Launching a successful cross-tenant, cache-based side-channel attack requires attackers to co-locate with the victims. Approaches to defend against co-location [26, 33, 28] are complementary to our efforts. New cache architectures to thwart cache-based side-channels [55, 54] face some deployment challenges as they require significant hardware support from chip manufactures. In contrast, `Cauldron` requires only off-the-shelf commodity hardware.

CHAPTER 4

APPROACH

4.1 System Model

We consider a public Platform-as-a-Service (PaaS) or Infrastructure-as-a-cloud (IaaS) cloud environment. Such a system allows the co-location of containers belonging to different clients/organizations on the same physical hardware. The mechanisms and ideas presented here are independent of the actual container runtime framework but we base our implementation on Docker [23]. Customers of the cloud framework need to specifically label the subset of containers that require increased protection from attacks that `Cauldron` enables (explained further below).

We also assume that the cloud computing infrastructure is built on commodity off-the-shelf (COTS) components that have multiple levels of caches, some of which are shared. For instance, we carry out our experiments on the Intel Haswell series of processors that have a three-level cache hierarchy: private level 1 (L1) and level 2 (L2) caches for each core and a last level (L3) cache that is shared among all the cores. This is the model of the system that we use for the remainder of this thesis. However the proposed methodologies are generally applicable as long as there exists a method to partition the caches at runtime. For this purpose, we turn to the Intel *Cache Allocation Technology* (CAT) [22, 24] that allows us to control the partitioning of the shared L3 cache. The CAT mechanism is configured using model-specific registers (MSRs). This can be carried out at runtime in a dynamic fashion using software mechanisms. On the Haswell series of processors the maximum number of partitions is limited to *four*. We provide a more detailed discussion on the Intel CAT in Section 2.2.2. Like in [40], we also assume that hyper-threading is disabled on machines hosting secure containers.

4.2 Attack Model

As mentioned earlier, we assume that containers from different sources (*e.g.,* different organizations) can be co-located on the same underlying machine. Hence, we assume that an adversary's container(s) can execute, in parallel, with those of the 'victim'. This allows the attacker to launch side-channel attacks using caches [59, 43, 35]. 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). In this thesis, we specifically focus on Prime+Probe and Flush+Reload attacks, both of which can be carried out in a public cloud computing infrastructure. We discuss the attacks in detail in Section 2.3.

We assume that the cloud service providers are, in themselves, not malicious since they have a vested interest in protecting their reputations. For this thesis, we do not consider attacks that aim to compromise the cloud computing infrastructure or the underlying operating system(s) – this will be taken up as part of future work. Containers can also be compromised via the communication network, *e.g.,* due to a vulnerability in a (web) service running in a container. These threats are not in the scope of this work as they are present even while executing in private IT infrastructures owned by the user. Furthermore, we assume that the list containing the subset of containers that require the higher resiliency/security is also trusted. Or rather, a front-end mechanism that accepts the jobs in the first place includes a vetting methodology that is separate from the actual physical hardware running the containers.

4.3 Cauldron Design

The design goals for the `Cauldron` framework are as follows: *(i)* protect containers from both same-core and cross-core cache-based side-channel attacks; *(ii)* not require changes to user applications and libraries; *(iii)* be easy to adopt and deploy and *(iv)* incur low performance overheads. Figure 4.1 presents a high-level overview of the framework.

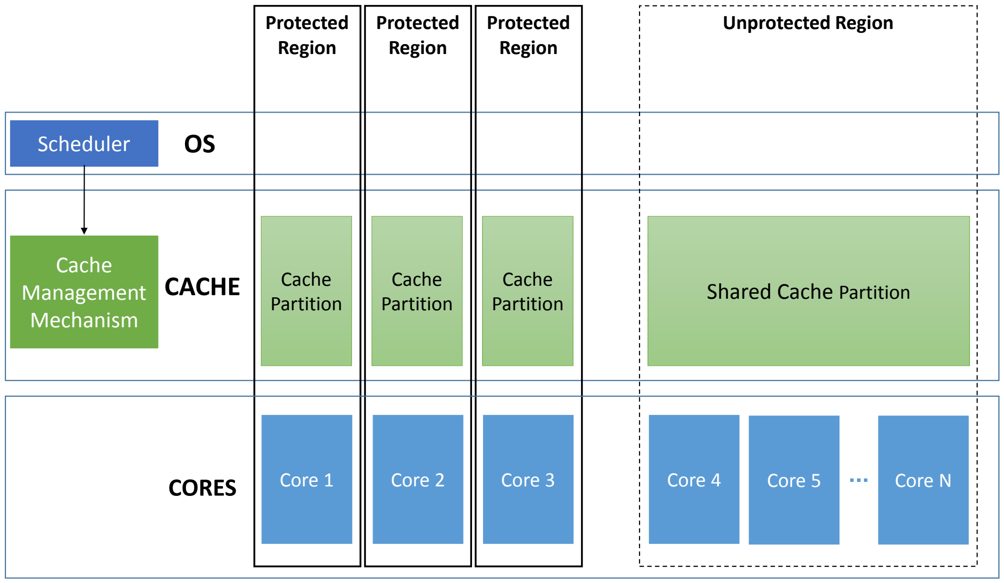


Figure 4.1: System design overview

4.3.1 Protection Model

In the `Cauldron` framework, clients/tenants are required to tag containers that need extra care for security – *i.e.,* the clients indicate which containers are carrying out critical operations or are dealing with sensitive data. This, along with the information about container organizational ownership, is the only additional input required from the user(s)[1]. If a tenant marks a particular (subset set of) containers as requiring increased security then the `Cauldron` framework will ensure that these containers are protected from side-channel attacks – both, from *all* containers belonging to other organizations as well as well as the non-secure containers of the same tenant. In fact, it is easy to extend `Cauldron` to support multi-level security (MLS) policies such as Bell-LaPadula confidentiality policy. This will extend the utility of this framework to private MLS cloud environments where the tenant containers all belong to the same organization but may have different security labels.

[1]We assume that the cloud service provider has a separate mechanism to verify whether the containers that claim to be from a certain organization are actually so. This is an orthogonal problem to the one being presented in this paper.

4.3.2 Protected Regions

Every host that is part of the `Cauldron` framework will have two regions – a *protected region* and an *unprotected region*. We leverage the Intel Cache Allocation Technology (CAT) [22, 24] to partition the host processor into the above regions. CAT allows us to partition the shared L3 cache among individual cores. Such partitions are created along cache ways; a minimum partition size being 2 MB. CAT can be configured at runtime using model-specific registers (MSRs). This allows us to *add, remove* or even *scale* partitions as needed. Hence, each "region" (protected/unprotected) in *Cauldron* is defined by: *(a)* a subset of processor cores and *(b)* a cache partition that is private only to those cores. One such set (or maybe more than one depending on how the system is configured) is marked as the "protected region". Hence, containers that are marked as secure/sensitive will be run on the protected core(s). CAT provides useful partitioning semantics and prevents cross-core PRIME+PROBE attacks across partitions. In particular, attackers cannot prime the L3 cache of the victim that is executing on a different partition since CAT prevents evictions across partitions. However, a trivial design entirely reliant on CAT will not prevent all side-channel attacks (*e.g.,* same-core attacks). Also, FLUSH+RELOAD attacks will still succeed since a process on one core can get hits from content loaded onto the cache from another partition. `Cauldron` addresses all of these problems while still taking advantage of the hardware-supported cache partitioning.

4.3.3 Cache Flushing

Even with hardware cache partitioning, L1 and L2 based side-channel attacks can succeed when the attacker and the victim share the same core [59]. Hence, we use *cache flushing mechanisms* to prevent such attacks. We use a software flushing mechanism (details in Section 4.4.3) but many processors include hardware mechanisms. One way to implement the flush mechanism would be invoke it within each container just before it relinquishes control. This would require modifications to the end user application. Moreover, it also increases inefficiency since, as we shall explain soon, the flushing mechanism *does not need to run after every protected container.*

4.3.4 Smart Scheduling

We need to ensure that *(a)* the sensitive containers are scheduled on to the protected cores and *(b)* the number of flushes is minimized. One way to to improve the isolation and security for the protected region is to assign only one core for the protected region. However, this may not be practical as *(i)* many cloud workloads are multi-threaded and will do better with multiple cores and *(ii)* Intel CAT currently only supports a small number of L3 partitions (maximum of four for this processor). One fix could be to create *multiple protected regions* with one core (and associated LLC partition) each. To ensure cloud applications get access to enough protected cores threads belonging to a container could be allowed to execute across multiple protected regions. As for the issue of number of flushes (in the protected regions), one trivial method could be to initiate a flush mechanism after *every* sensitive container executes. Of course, this could result in a lot of wasted resources since we *do not need to flush the cache if two sensitive containers from the same organization/tenant execute in a back-to-back fashion*. Also, we only flush the cache partition belonging to the protected region(s). This ensures that other containers do not pay the performance penalty associated with having their caches flushed. Reducing the size of the LLC that needs to be flushed is important as LLC sizes have grown to sizes of over 45MB. Flushing such a large cache securely would be detrimental to system performance.

4.3.5 Preventing Page Sharing

As previously discussed, the Intel CAT technology alone cannot thwart all cross-container L3 based attacks. To prevent an adversary container from getting a hit due to page(s) being in the protected L3 partition of the victim (CAT allows cross-partition hits but not evictions), we should *prevent page sharing*. In fact, only containers from the same organization may be allowed to share pages and even then only if such containers are all marked for protection. However, it is common for container frameworks to use layered file systems that share page caches (*e.g.,* AUFS in Docker). While it would be ideal to control this at a finer granularity, in our current design we employ filesystems that do not use shared page caches (*e.g.,* btrfs, DeviceMapper).

4.4 Cauldron Implementation

Below we outline the details of our implementation.

4.4.1 Protected Regions

In our current implementation, we associate only one core with a protected region. As shown in Figure 4.1, a machine with an 8-core processor is partitioned into 3 protected regions each with one core and their own LLC allocated with CAT, and one unprotected region with the remaining 5 cores and a larger partition of LLC. Each protected region is allocated a cache partition of 2MB. We believe this to be a reasonable choice for cloud applications [30], and our performance evaluation suggests as much. We will explore dynamically varying the number and size of protected regions and the cores associated with them to improve performance of multi-threaded applications in future work.

4.4.2 Smart Scheduler

In order to ensure `Cauldron` is easily deployable, we implement all kernel level scheduler code as a loadable kernel module requiring no changes to the host kernel. We leverage kernel return probes [12] to hook into the kernel schedule routine that is called on every context-switch. Once control is passed to our module, we determine if the container to be scheduled next belongs to a different organization than the container previously scheduled in this protected region. If the containers belong to different organizations we flush the cache, otherwise we immediately return control to the kernel schedule routine.

4.4.3 Cache Flushing

`Cauldron` employs a software-based mechanism to flush the cache allocated to a secure partition. For each protected region, we allocate a physically contiguous array equal to the size of the L3 partition allocated to that region. This is done at the time our kernel module is loaded into the kernel. During a context-switch involving containers belonging to different organizations, we read the elements in this array into the cache. This operation flushes every cache set in the specific protected region. In order to minimize the overhead of cache flushing we traverse the array in cache line size steps. This insures that each cache line allocated to the specific protected partition is touched only once.

4.4.4 Integration with Docker

We use Docker to manage containers in `Cauldron` framework. The organization of a given container is configured using the "–parent-cgroup" option of the Docker *run* command. If an organization wants a particular container to run with the protection of `Cauldron`, then the parent cgroup of that container will be used to differentiate it from containers of other organizations scheduled on protected cores. Implementing the `Cauldron` framework in a popular container runtime like Docker, ensures compatibility with existing software and allows for easier deployment and integration into the larger container management ecosystem.

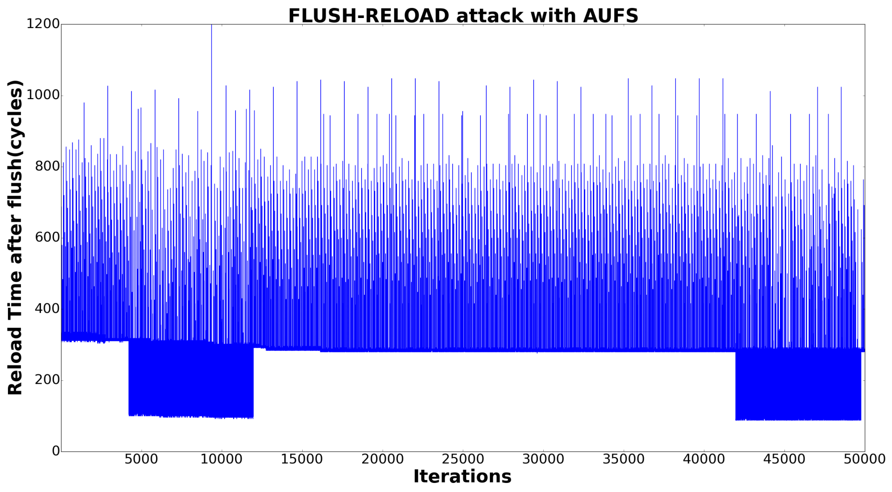
CHAPTER 5

EVALUATION

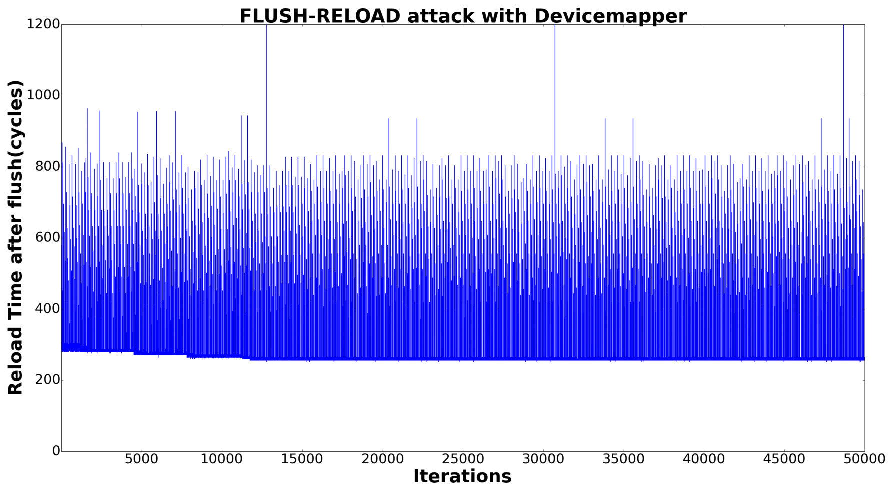
We evaluate our design and implementation in terms of both effectiveness (the ability of the system to stop attacks) and in terms of the performance overhead of running containers on a system using our secure design. We ran our experiments on a Ubuntu 15.04 operating system on top of a CAT-enabled 8 core Intel(R) Xeon(R) CPU E5-2618L v3 @ 2.30GHz machine with 16 GB memory and a 20MB (shared) L3 cache. We use Docker v1.91 as the container runtime.

5.1 Security Evaluation

As discussed previously, FLUSH+RELOAD attacks across `Cauldron` partitions may still succeed if the attacker and victim share pages as using the `clflush` instruction the attacker can flush any of the lines in the shared pages from the cache hierarchy. Since `clflush` is not a privileged instruction, trapping it would incur significant overhead; instead `Cauldron` focuses on the sources of shared memory the attack leverages. To defend against FLUSH+RELOAD attacks across `Cauldron` partitions we used a layered filesystem that does not support page caching of shared layers. As discussed further in Chapter 6, we intend to expand `Cauldron` to also partition the page cache along organization boundaries so that containers belonging to the same organization can benefit from memory sharing of base layers. To validate that this approach prevents FLUSH+RELOAD attacks from deducing meaningful information we launched the FLUSH+RELOAD attack from [57] on a victim running GPG decryption across a `Cauldron` partition. Figure 5.1a shows access times observed by the attacker when using AUFS, a layered filesystem using a page cache, and when using DeviceMapper (Figure 5.1b), a layered filesystem driver that does not support a page cache. As can be seen in it-



(a) Using AUFS



(b) Using DeviceMapper

Figure 5.1: Cache access latency over time for the FLUSH+RELOAD attack

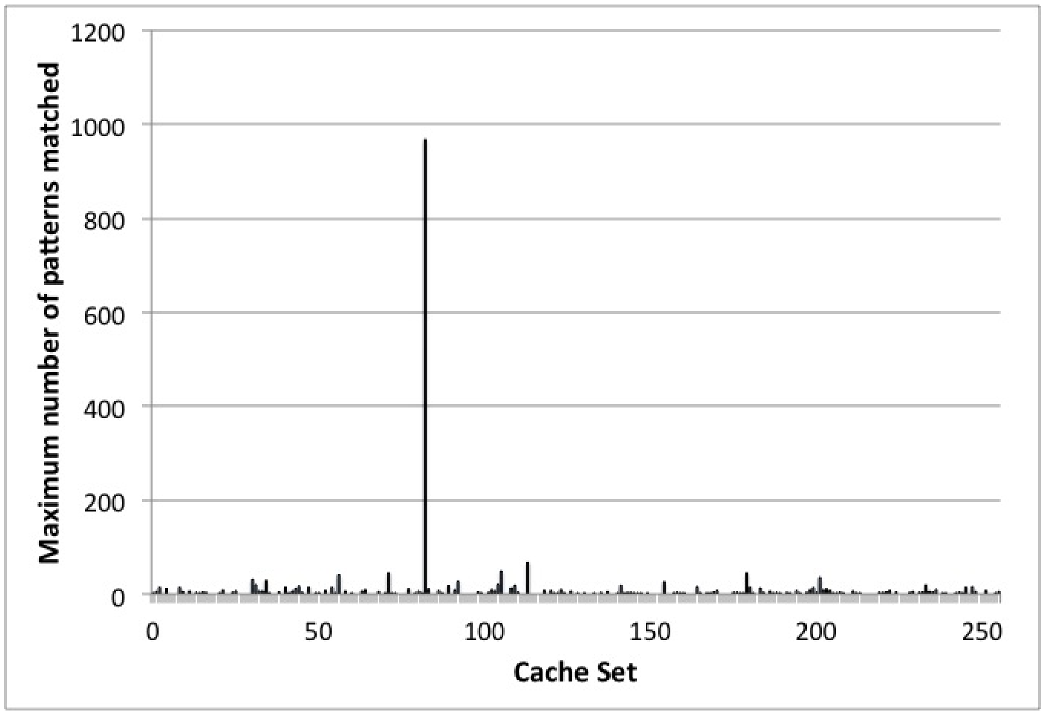
erations 5000 through 12500 and 42500 through 50000 in Figure 5.1a when using AUFS the attacker observes low access times due to cache hits from the victim's LLC partition and will be able to deduce meaningful information about victim's computation patterns. However, the attacker will be unable to derive information when the DeviceMapper storage backend is used as can be seen by the lack of access time variations in Figure 5.1b.

To validate that `Cauldron` is able to defend against cross-core Prime-Probe attacks, we launch the attack from [43] with the attacker and victim running on different cores in the unprotected partition and then with the victim running in a separate `Cauldron` partition. In the attack implementation, the attacker uses the Prime+Probe technique to monitor the victim's access patterns of different cache sets. The attacker then searches the cache trace for temporal access patterns indicative of the target application. A high pattern match count on a cache set indicates that the attacker is able to identify the cache set being used by the target application, in this case GnuPG version 1.4.13. The attacker can then continue to extract the private key.

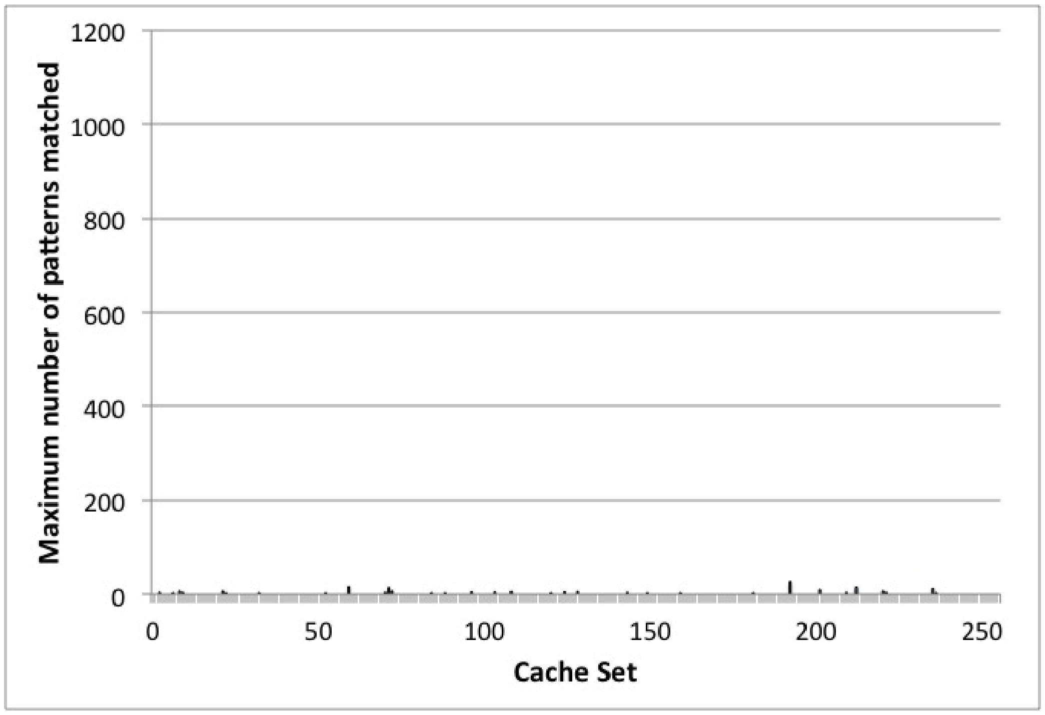
Figure 5.2a shows the pattern match count for the different cache sets when the attacker and victim run on different cores in the unprotected region. A high pattern match count for one cache set shows that the attacker is able to identify the cache set being used by the victim application. Therefore the attack is successful. On the other hand, minimal pattern matches, shown in Figure 5.2b indicates that the attack is unsuccessful once we move the victim to a protected region. This is because it is not possible to prime the LLC cache of the victim running in a different partition as CAT doesn't allow evictions across partitions.

5.2 Performance Evaluation

We have observed promising initial results by running the OpenSSL and Redis Benchmarks using configurations packaged by Phoronix Test Suite [15]. The OpenSSL benchmark tests RSA 4096-bit performance throughput while the Redis Benchmark tests a variety of operations including getting and setting values, list operations, and a set operation. We ran each benchmark with 1, 2, 4, and 8 containers assigned to different organizations on the pro-



(a) With the victim and attacker running in the unprotected region



(b) With the victim and attacker running in different partitions

Figure 5.2: Pattern match count for the Prime+Probe attack

tected partitions of *Cauldron* and saw less than 1% overhead for the OpenSSL benchmark and no more than 16% overhead for the worst case Redis Benchmark (8 containers running on only 3 cores, at full load). In future work we will evaluate this further while investigating features such as dynamic sizing of the secure region (right now `Cauldron` statically dedicates 3 cores to running protected containers).

CHAPTER 6

FUTURE WORK

Preliminary evaluation results indicate that `Cauldron` meets its design goals reasonably well but that there is room for improvement. In particular, `Cauldron` is effective in thwarting both Flush+Reload and Prime+Probe attack types, and in protecting against both same-core and cross-core attacks. `Cauldron` doesn't require changes to user applications and libraries and is also easy to deploy requiring changes only to the host environment and support for Intel CAT technology which is readily available. However, the current implementation of `Cauldron` requires the use of a storage backend that doesn't allow libraries to be shared between containers, in order to defend against cross-partition Flush+Reload attacks. This may increase the memory footprint of containers in high density environments. A future direction is to explore mitigation of this overhead by selectively enabling shared pages.

`Cauldron` currently only supports protected partitions with single cores. This is to ensure that Prime+Probe attacks using the LLC cannot be launched across cores within a protected region. However, an alternative approach to dealing with such attacks is to employ constrained- or gang-scheduling to ensure that only containers belonging to the same tenant can be running on the cores in a given protected region at any given time. Each approach has its own advantages and disadvantages. The current approach of limiting each protected region to one core but allowing cloud workloads to execute across multiple protected regions will lead to smaller LLC availability and frequent context switches leading to increased cache-flushing overhead. The alternative approach on the other hand may lead to underutilization of processor cores due to gang-scheduling. We will explore the trade-offs between these two approaches in future work.

Another direction for future work is to attempt further reduce cache-flushes, say, by increasing the minimum runtime (MRT) of processes and thereby reducing the total number context switches albeit at the expense of container response time. Previous works [34, 45, 47] have shown that it is also possible to optimize the number of cache-flushes by minimizing context switches involving processes belonging to different security levels. Other interesting directions for future work include dynamically adjusting the L3 cache size and cores associated with `Cauldron` partitions.

CHAPTER 7

CONCLUSION

The proliferation of lightweight commodity computing and the slowing down of Moore's law could mean that cloud providers may no longer be able to scale their hardware resources to match the increased demand from their clients. Hence, there will be an increased likelihood that computing jobs from multiple organizations could be co-located on the same physical hardware. This raises serious security and privacy concerns that we hope to mitigate by use of the `Cauldron` framework. `Cauldron` intends to provide isolation guarantees to application developers; and it does this without requiring any changes to the applications themselves.

REFERENCES

[1] Amazon ec2. https://aws.amazon.com/ec2.

[2] Apparmor. http://wiki.apparmor.net/index.php/Main_Page.

[3] Azure virtual machines. https://azure.microsoft.com/en-us/services/virtual-machines/.

[4] Chroot. http://man7.org/linux/man-pages/man2/chroot.2.html.

[5] Clfush instruction reference. http://x86.renejeschke.de/html/file_module_x86_id_30.html.

[6] Control groups introduced in mainstream linux. http://kernelnewbies.org/Linux_2_6_24.

[7] Docker hub. https://hub.docker.com/.

[8] Google compute engine. https://cloud.google.com/compute/.

[9] Huge pages. https://lwn.net/Articles/374424/.

[10] Kernel-based virtual machine. http://www.linux-kvm.org/.

[11] Kernel namespaces. https://lwn.net/Articles/531114/.

[12] Kernel probes (kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt.

[13] Look-aside look-through caches. http://download.intel.com/design/intarch/papers/cache6.pdf.

[14] Namespaces in operation. https://lwn.net/Articles/531114/.

[15] Phoronix test suite. http://www.phoronix-test-suite.com/.

[16] Quay. https://www.quay.io/.

[17] Redhat introduction to cgroups. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html.

[18] rkt. https://www.coreos.com/rkt.

[19] Selinux. http://selinuxproject.org/page/Main_Page.

[20] User namespaces. http://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[21] Federal Cloud Computing Strategy. http://www.cio.gov/documents/Federal-Cloud-Computing-Strategy.pdf, February 2011.

[22] Cache monitoring technology and cache allocation technology. http://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html, 2015.

[23] docker. https://www.docker.com/, 2015.

[24] Improving real-time performance by utilizing cache allocation technology. http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html, 2015.

[25] Linux containers. https://linuxcontainers.org/, 2015.

[26] Azar, Y., Kamara, S., Menache, I., Raykova, M., and Shepard, B. Co-location-resistant clouds. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2014), CCSW '14, ACM, pp. 9–20.

[27] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[28] Bijon, K., Krishnan, R., and Sandhu, R. Mitigating multi-tenancy risks in iaas cloud through constraints-driven virtual resource scheduling. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2015), SACMAT '15, ACM, pp. 63–74.

[29] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers.

[30] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 37–48.

[31] Godfrey, M., and Zulkernine, M. Preventing cache-based side-channel attacks in a cloud environment. *Cloud Computing, IEEE Transactions on 2*, 4 (2014), 395–408.

[32] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 897–912.

[33] Han, Y., Chan, J., Alpcan, T., and Leckie, C. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *Dependable and Secure Computing, IEEE Transactions on PP*, 99 (2015), 1–1.

[34] Hu, W.-M. Lattice scheduling and covert channels. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on* (May 1992), pp. 52–61.

[35] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on* (May 2015), pp. 591–604.

[36] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 299–319.

[37] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Know thy neighbor: Crypto library detection in the cloud. In *Proceedings on Privacy Enhancing Technologies 2015* (2015).

[38] Jackson, P., and Lameter, C. *CGROUPS*.

[39] Kim, T., Peinado, M., and Mainar-Ruiz, G. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 11–11.

[40] Kim, T., Peinado, M., and Mainar-Ruiz, G. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 189–204.

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

[42] Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing.

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

[44] Martin, R., Demme, J., and Sethumadhavan, S. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 118–129.

[45] Mohan, S., Yoon, M.-K., Pellizzoni, R., and Bobba, R. Real-time systems security through scheduler constraints. In *Euromicro Conference on Real-Time Systems* (July 2014), pp. 129–140.

[46] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'06, Springer-Verlag, pp. 1–20.

[47] Pellizzoni, R., Paryab, N., Yoon, M.-K., Bak, S., Mohan, S., and Bobba, R. A generalized model for preventing information leakage in hard real-time systems. In *IEEE Real-Time Embedded Technology and Applications Symposium* (April 2015 (accepted)).

[48] Percival, C. Cache missing for fun and profit, 2005.

[49] Raj, H., Nathuji, R., Singh, A., and England, P. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2009), CCSW '09, ACM, pp. 77–84.

[50] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[51] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[52] Varadarajan, V., Ristenpart, T., and Swift, M. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 687–702.

[53] Vattikonda, B. C., Das, S., and Shacham, H. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2011), CCSW '11, ACM, pp. 41–46.

[54] Wang, Z., and Lee, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 494–505.

[55] Wang, Z., and Lee, R. B. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2008), MICRO 41, IEEE Computer Society, pp. 83–93.

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

[57] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 719–732.

[58] Ye, Y., West, R., Cheng, Z., and Li, Y. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 381–392.

[59] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.

[60] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

[61] Zhang, Y., and Reiter, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 827–838.