Preventing Cache-Based Side-Channel Attacks in a Cloud Environment

Michael (Misiu) Godfrey and Mohammad Zulkernine, *Member, IEEE*

Abstract—Cloud computing is a unique technique for outsourcing and aggregating computational hardware needs. By abstracting the underlying machines cloud computing is able to share resources among multiple mutually distrusting clients. While there are numerous practical benefits to this system, this kind of resource sharing enables new forms of information leakage such as hardware side-channels. In this paper, we investigate the usage of CPU-cache based side-channels in the cloud and how they compare to traditional side-channel attacks. We go on to demonstrate that new techniques are necessary to mitigate these sorts of attacks in a cloud environment, and specify the requirements for such solutions. Finally, we design and implement two new cache-based side-channel mitigation techniques, implementing them in a state-of-the-art cloud system, and testing them against traditional cloud technology.

Index Terms—Cloud computing, CPU cache, parallel side-channel, performance, side-channel, security, sequential side-channel

1 Introduction

While many of the concepts of the cloud paradigm have been used in other areas, there are some features that are particularly unique to the cloud. One cloud feature of particular interest is the concept of Mutually Distrusting Clients. Unlike most other paradigms, cloud computing allows potentially malicious or antagonistic clients access to the same hardware.

Giving potentially malicious clients access to the same hardware means that they may be able to exploit one another. Specifically, access to the same hardware means that they can exploit the physical properties of the machine for communication or information leakage. Exploiting these properties is called using a hardware-based side-channel [6].

Traditionally, the most difficult task in establishing a side-channel was gaining access to the same hardware as the target. This can be easily accomplished, however, in the cloud. While virtualization and other technologies are heavily used in cloud systems [8], they do not fully prevent information leakage along these channels.

Because each side-channel exploits a specific piece of hardware, each channel is unique and provides unique vulnerabilities. To this extent, the CPU cache is the most commonly exploited channel in the cloud as it yields one of the highest communication speeds the most reliably. Since there has been comparatively little work demonstrating the severity of other channels in the cloud this paper focuses on the prevention of cache-based side-channels as opposed to any other medium. To date the only successful side-channel attacks in the cloud has been performed using the CPU cache [17].

Since the potential for these attacks has been shown, there have been several attempts to mitigate such situations. Unfortunately, these solutions either require the client to modify their software [3], [10], or the underlying hardware [7]. From our studies of cloud systems, we believe that either of these modifications disrupts the relationship between the cloud provider and their users, which we refer to in this paper as the ″cloud model″. We see the lack of an existing, non-intrusive solution to these attacks to be the motivation behind this work.

Our goal in this paper is to provide solutions that both prevent cache-based side-channels and respect the relationship between the cloud provider and the user. To this extent we specify the contributions of the paper as follows:

- A study of cache-based side-channels in a cloud environment. This includes a categorization of existing attacks into the *Sequential* and *Parallel* types.
- Two server-side defences against cache-based side-channels. One focuses on sequential side-channels [2], which includes a technique to prevent the side-channel's occurrence as well as an algorithm designed to implement the technique. The algorithm applies the solution in a minimalistic fashion to help minimize resulting overhead. The second focuses on parallel side-channels, and uses a cache colouring technique to prevent their occurrence and to improve cache efficiency in certain situations.
- An implementation and validation of the above defences. We implement the above defences and demonstrate their ability to prevent cache-based side-channel attacks in an experimental cloud environment. The defences are evaluated by running them against attacks designed to be representative of previously published attacks of the cache-based side-channel genre.

The rest of the paper is organized as follows: Section 2 explains the context in which these attacks occur and are mitigated. Section 3 details other work related to the attacks

- *The authors are with the School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: {godfrey, mzulker}@cs.queensu.ca.*

Manuscript received 14 Jan. 2014; revised 4 June 2014; accepted 23 Aug. 2014. Date of publication 15 Sept. 2014; date of current version 30 Jan. 2015. Recommended for acceptance by D.S.L. Wei, S. Pearson, K. Matsuura, P.P.C. Lee, and K. Naik.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2014.2358236

and how to mitigate them. Section 4 specifies the details of how the attacks work. Section 5 details our solution to sequential side-channels. Section 6 explains our solution to parallel side-channels. Section 7 concludes the paper.

2 Background

The idea of cloud computing revolves around the pooling of multiple, large-scale, computing resources into a single, abstract, entity commonly referred to as the cloud. This construction allows multiple clients concurrent access to these resources. Conceptually similar to the Mainframe paradigm of decades past, cloud computing differs in its exensive use of networking technologies. A cloud system typically focuses on using many machines, composed of lower-grade canonical hardware, rather than fewer, more powerful, machines. Due to its underlying discreteness, complex software technologies need to be used in the cloud to abstract these individual machines into a single dynamically manageable resource.

2.1 The Cloud Model

Cloud providers have a specific relationship with their users that we refer to in this paper as the cloud model. This model refers to two key attributes of cloud functionality that we intend to preserve while making the paradigm more secure. The first is that users tend to run canonical software in the cloud. Often a user will either not have the technical skill necessary to customize their workload for security purposes, or will not have access to the code to make the modifications necessary. The second is that cloud systems are typically built on canonical hardware, and requiring them to do otherwise would be financially, and structurally costly. With these concepts in mind, we specify the conditions for holding with the cloud model:

- It does not require any software modifications by the client or on the client-end of the interface.
- It does not require any modifications to the underlying hardware.

2.2 Side-Channel Attacks

A side-channel in a software program is a means of communication via a medium not intended for information transfer [7]. A side-channel functions by establishing a correlation between the executing software and the underlying hardware phenomena. A reliable correlation can be used to infer what is occurring in one by observing the other.

To infer more specific information about the executing software we typically need more data from the hardware. To this extend the higher rate hardware channels are typically used. As one of the highest rate channels easily accessible in the cloud, the CPU cache is often used for these types of attacks [16].

In the past, side-channels have been used to leak information across virtual barriers in local systems. Cache-based side-channels have been used to break the advanced encryption standard and data encryption standard (DES) protocols. In 2003 Tsunoo et al. were able to crack the DES encryption algorithm by correlating timing measurements in the cache with the encryption of the algorithm [14].

TABLE 1
Solution Types and Modification Requirements for Cache-Based Side-Channels

Solution Type	Source	Hardware	Overhead
Obscure Cache-Data Correlation	Y	N	N
Delay Timing Information	Y	N	N
Normalize Cache State	N	N	Y
Custom Hardware	N	Y	N
Disable Cache	N	N	Y

In order for a cache-based side-channel attack to be possible, the attacker and the victim must share some level of cache. Using modern hardware this is possible in two different ways: One is for the two processes to have sequential access to the cache, and the other is for them to have concurrent access. Sequential access is achieved by having processes context switch on the same processor core, while concurrent access requires a cache to be shared between multiple cores and is a restriction of the hardware.

Research has been done into attacks for both classes of channel [15] with the former typically seen as more portable as only some systems will allow for parallel access to a cache. However, recent trends in hardware technology are seeing more and more CPUs outfitted with larger, shared, victim caches. These caches are designed to be shared among multiple cores and can be accessed in tandem.

While the techniques for attacking these two types of cache are quite similar, the hardware differences require dramatically different solutions. In order to address both types of channels we have devised two solutions. For the sequential channel we apply a technique called Selective Cache Flushing, which can be found in Section 5. For parallel channels we apply a technique called Cache Partitioning, based on cache colouring, which can be found in Section 6. These two solutions can be implemented in conjunction to insulate a hypervisor against both types of side-channels.

3 Related Work

This section details previous work related to side-channels and techniques to mitigate them. It includes a summary of cache-based side-channels and their typical mitigation techniques, then compares them to their cloud-based variations.

3.1 Side-Channels

Side channels have been used in recent work to bypass virtual machine isolation in the cloud. Examples include verifying if two virtual machines are co-resident [9], and more dangerously, the extraction of cryptographic private keys from unwary hosts [17]. Similar research into side-channels yields additional attacks that can be migrated to the cloud [6], [11], [14].

When attention was first drawn to side-channels [4] there were several proposals for how to mitigate their potential [6], [7]. The solutions typically fit into five categories depicted in Table 1. Various techniques to implement these solutions, such as altering how the algorithm uses the cache, or customizing the hardware channel would require either the modification of the source code, the hardware, or cause an unacceptable amount of overhead.

Cache flushing has been considered, but disregarded as a solution to traditional cache-based side-channels because it is expected to generate large amounts of overhead [6].

3.2 Side-Channels in the Cloud

Kim et al. have developed a solution for cache-based side-channels in cloud systems [3]. In their solution, they prevent cache-based side-channels by giving each VM exclusive access to a sectioned portion of the cache they call a stealth page. Using a stealth page, the sharing of cache information is prevented by having each VM restore its context in that page before its time slice execution.

In order to have software applications access these hidden pages, their solution requires the user to make client-side modifications to the software being executed in the guest VM. Due to its canonical nature, however, we believe that most clients will either not have the access, or not have the technical skill, to modify the software they intend to run in the cloud. This restriction demonstrates the need for a solution transparent to the Client.

For our solution, we implemented a purely server-side defence for cache-based side-channels in the cloud. To make it fully compatible with the cloud model, we impress the constraints that it prevents cache-based side-channels between co-resident VMs, and also that it requires no modifications of the underlying hardware nor of the software used to run the cloud. From this perspective, the solution should be both secure and invisible to the Client, as well as to the cloud provider. Only the cloud developer would be aware that such a solution is in place.

4 Side-Channel Attacks

Cache-based side-channel attacks can be categorized into two types: Sequential and Parallel. These attacks vary in whether they employ concurrent access to the CPU-cache or not. At time of writing, these two attacks are the only cache-based side-channel attacks known to affect the cloud. The two types are discussed in the following sections.

4.1 Sequential Side-Channel

The first documented form of a cache-based side-channel attack explored in the cloud was by Ristenpart et al. [9] who demonstrated the use of side-channels to verify virtual machine co-residence in Amazon's EC2. As part of their work, they explored the use of a previously identified cache-based side-channel technique, which they refer to as the *Prime+Probe* technique, in a cloud environment. The result of such work is the *prime+trigger+probe* (PTP) technique, illustrated in Fig. 1, a variation designed to work in cloud environments. The purpose of their experiment, using the PTP technique, was to see if a cache-based side-channel could be established between two guest domains in the cloud. The channel was established such that the first VM (referred to as the *probing instance*) could receive a message that the second VM (the *target instance*) encodes in its usage of the cache. The basic version of the PTP technique shown in Fig. 1 is an example of a sequential side-channel.

In the PTP technique, the probing instance first separates the cache lines into two categories: the *Touched* category and the *Untouched* category. Once the channel has been established, cache lines in the touched category will be modified by the target instance. Lines in the untouched category will remain intact.

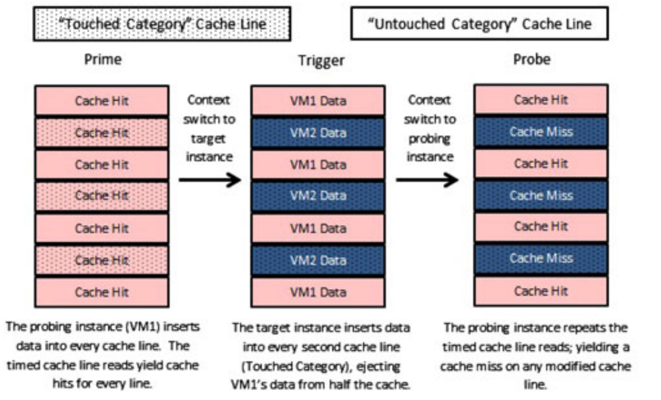

"Touched Category" Cache Line
"Untouched Category" Cache Line
Prime
Trigger
Probe
Cache Hit
Context switch to target instance
VM1 Data
VM2 Data
Context switch to probing instance
Cache Miss
The probing instance (VM1) inserts data into every cache line. The timed cache line reads yield cache hits for every line.
The target instance inserts data into every second cache line (Touched Category), ejecting VM1's data from half the cache.
The probing instance repeats the timed cache line reads; yielding a cache miss on any modified cache line.

Fig. 1. The prime+trigger+probe technique.

Using the timestamp counter in the CPU, the probing instance can measure the number of CPU cycles required to access each cache line. The resulting differences in access times for hits and misses in the cache is observable by the probing instance and will serve as the communication means within the channel.

Once the categories have been established, the probing instance primes the cache by filling as many cache lines as it can (ideally, all of them). It then establishes an access time baseline by reading from each line in both categories. Having just been primed, each cache line should yield a cache hit, regardless of category, thus keeping the baseline access times low. This process is highlighted in the *Prime* step of Fig. 1. Having done this, the probing instance now has a series of values which represent how long it took to access each cache line with a primed cache.

Once the cache has been primed, and the baseline established, the probing instance must *Trigger* (context switch to) the target instance by busy looping, or otherwise giving up its time slice. When the target instance begins its time slice, it heavily accesses the cache lines in one of the pre-defined categories of the cache (the Touched category), but not the other (the Untouched category). The effect of this switch on the data can be seen in the Trigger step of Fig. 1. In this figure, the target instance is accessing every second cache line in an access pattern we have pre-defined. The resulting set of accessed lines we define as the Touched category of the cache. By contrast, cache lines in the untouched category were not accessed by the target instance. When the CPU core context switches back to the probing instance, the instance probes the cache by re-measuring the access times for each line. This is illustrated in the *Probe* step of Fig. 1. If there is a significant increase in the access times for cache lines in the touched category compared to the untouched category, then the probing instance can assume that the target instance was trying to communicate.

The PTP technique was refined in later work by Wu et al. [15], where they establish a high-speed bit-stream by communicating a "1" or a "0" based on whether the difference between category timings is positive or negative (assuming

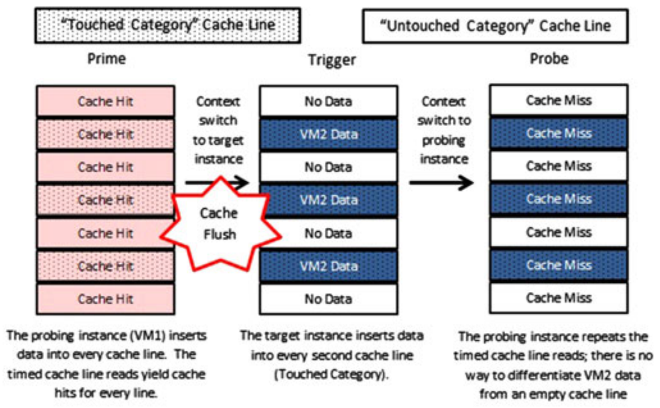

"Touched Category" Cache Line
"Untouched Category" Cache Line
Prime
Trigger
Probe
Cache Hit
Context switch to target instance
Cache Flush
No Data
VM2 Data
Context switch to probing instance
Cache Miss
The probing instance (VM1) inserts data into every cache line. The timed cache line reads yield cache hits for every line.
The target instance inserts data into every second cache line (Touched Category).
The probing instance repeats the timed cache line reads; there is no way to differentiate VM2 data from an empty cache line

Fig. 2. Our solution's effect on the PTP technique (sequential).

the difference is above a certain threshold). Using this technique, they were able to establish reliable side-channel bitstreams of over 190 kbs. This attack technique has so far been the most robust and reliable cache-based side-channel attack demonstrated in a cloud environment. At time of writing, all other cache-based side-channel attacks have been based on this technique, making it a good example of a canonical attack. Since all cache-based side-channel attacks in the cloud rely on this basic technique, a successful inhibition of its principles would mitigate all currently viable attacks, including some of the more dangerous variations [17]. For these reasons, this particular side-channel attack has been implemented and is used as an example attack in our system's evaluation.

4.2 Parallel Side-Channel

While the technique described is for a sequential side-channel, it can be adapted on a shared-cache system to become a parallel side-channel. In this case, the probing and target instances are each running on a separate core but have parallel access to some shared cache. While the cache access is essentially the same, the parallel technique omits the need to trigger between VMs because both the Trigger and the Probe steps are happening concurrently.

Much like the sequential technique, the process begins with the probing instance priming the cache. Once the cache is primed, rather than context switching to the target instance the probing instance starts the "Probe" step. Likewise, once the priming is done, the target instance can start the "Trigger" stage. In this way, the technique functions the same way as the sequential version, except that the "Trigger" and "Probe" steps are occurring at the same time. As both VMs cannot access the same cache lines at the same time, they instead each work on a section of the cache. Therefore, while one set of cache lines is being read, another can be modified by the other VM.

As might be expected, the parallel technique is less reliable as an attack medium, as there tends to be more noise in the system. To date, only a sequential side-channel attack has been demonstrated to be able to do serious damage in the cloud [17]. However, while more difficult to use, parallel channels still hold the potential to be used in such an attack, and they can still be used to gain otherwise inaccessible information about a VM.

5 Selective Cache Flushing

Our solution inhibits the PTP technique by selectively flushing the cache between the Prime and Trigger steps, thereby preventing the probing instance from ever seeing a pattern in the cache hit data. This method is detailed in Fig. 2. Additionally, if the system supports cache warming (the saving of the cache's state on a context switch) the flush can be applied within this subsystem. This would allow the cache to retain useful data for the VM, while still preventing each VM from seeing if another has used the cache. A cache warming technique was not applied in our system as it would not make the system any more secure, it is instead left for future work.

Our solution can be implemented directly into the hypervisor, thereby conforming to the cloud model defined in Section 2.1.

5.1 Expected Issues and Mitigations

The chief drawback of a cache flushing approach lies in the levels of overhead that such a technique may generate. This manifests, principally, as additional cycles required to perform the cache flushes frequently. Starting with a flushed cache should not add much overhead to the system as VMs would usually regain context with a mostly invalid cache. If the system supports cache warming, then this solution can be performed prior to it and the VMs can begin with a warm cache despite the flushes.

The overhead generated from flushing the cache is dependent on two main factors: The ratio of flushes to context switches and the frequency of context switching in the system. These factors compound one another to generate the main source of overhead for the system. The more context switches the more flushes required, the more flushes required the more overhead generated.

When cache-based side-channels were being investigated in a non-virtualized environment, flushing the cache was deemed too expensive for general use [6]. This was because of the high rate at which cache flushes would need to occur and the overhead that would come with each flush. However, a cloud system does not require side-channel security on the process-level, but on VM-level granularity, as this is how they are allocated resources.

By comparison, a hypervisor will be running fewer VMs simultaneously than a regular OS would be running processes, and the rate of switching between them would be much lower. This means that the context switch rate between VMs in a cloud system should be much lower than that between processes in a regular OS. Since the cache-flushing code can be inserted into the existing context-switching functionality the overhead generated by flushing the cache should be directly proportional to the context switch rate. With this in mind the reduced granularity of cloud systems makes this technique more viable in a cloud environment.

In addition to the reduced rate, some mitigating factors as to the frequency of flushing can be postulated. The frequency of flushing depends on the exact circumstances during which the sharing of the cache could occur. For instance, flushing the cache would not be necessary if a guest VM switches to a VM that does not intend to access the cache (such as the idle domain). By tightening restrictions on when a context switch between virtual machines is considered vulnerable to a side-channel attack, we can reduce the frequency of context switches that require a cache flush.

In total, the issue of cache flushing overhead can be addressed in two ways simultaneously. The frequency of cache flushing is naturally reduced by focusing on VM-level granularity, rather than process level granularity; and the ratio of cache flushes to context switches can be reduced by tightening restrictions on when a context switch is considered vulnerable to a side-channel. By addressing each factor, the net overhead decreases to a point where it is manageable in a standard cloud system.

5.2 Cache Flushing Technique

Our solution to sequential side-channels involves the incorporation of our cache flushing technique into a canonical cloud system. The technique includes two new functions within the hypervisor: One is a server run process to revert the cache to a blank slate (flush the cache), and the other is a tainting algorithm in the scheduler for deciding when flushing the cache is necessary. The sequential side-channel solution was implemented using the code base for the open source Xen Hypervisor, Version 4.2.

5.2.1 *Decision Algorithm*

Flushing a high level cache on a modern machine can be a time consuming process. As mentioned in the above section, it is better to flush the cache only when necessary. Because the PTP technique requires that two VMs have consecutive access to a CPU core, flushing the cache is only required when this situation is able to occur. For instance, Xen maps VMs to *domains* so that it can compare guest VMs, the host VM, and additional states (such as an idle CPU) using the same structures. If a CPU goes from executing Domain1 (VM1) to the Idle domain (Xen's interpretation of an idle CPU) then back to Domain1 there is no need to flush the cache, as no side-channel could have been implemented over such a context. A tainting algorithm, Algorithm 1, was implemented to record which VMs own data currently in the cache and to determine when it is necessary to flush those data.

Algorithm 1. Prevent Same-VM or Idle-VM Flushes

```
Function contextSwitch(DomX, DomY) {
if DomX.taint == idle then return; end if
if DomX.taint == DomY.id then return; end if
if DomY.id == idle then
    DomY.taint := DomX.taint; return;
end if
flushCache(); return;
} EndFunction
```

The Xen scheduler operates on scheduling units called VCPUs. Each VCPU represents a virtual CPU and is associated with a domain. According to our algorithm, each VCPU is given a new data field *taint* which indicates the origin of the data that currently reside in the CPU cache. Upon initialization, each VCPU's taint value is assigned the identifier for the domain it represents, with the exception of the Idle VCPU, which does not have an associated domain and is assigned an idle value.

The outcome of Algorithm 1 is that a cache flush occurs only when a context switch changes from one domain to another where the second domain has the ability to establish a side-channel with the first. Switching to the Idle domain, or to the same domain, will not invoke a cache flush. Avoiding a flush during these situations can be critical, as they will arise often in an elastic cloud environment.

At present Algorithm 1 is quite conservative and does not distinguish between the Prime and Probe steps of a side-channel attack. In practice, a flush would only be necessary before the Probe step, which could be recorded in the scheduler by further tainting. Theoretically, this would further reduce the overhead of this algorithm by at least 50 percent but is left by the authors for future work.

5.2.2 *Flushing Function*

The cache flushing functionality was implemented in two versions. The first version is a hardware dependent (referred to as the x86-secure for its use of the instruction set) implementation. The second version is an independent (referred to as the portable-secure) implementation.

If the hardware has built-in functionality for flushing the cache, such as with the x86 instruction set, this task can be accomplished more efficiently, but sometimes less precisely. Both versions are compared with the existing version of the Xen 4.2 hypervisor (referred to as the insecure hypervisor) in Section 5.5. In this case, we use the *wbinvd* (write-back invalidate) function built into the x86 instruction set to flush the cache. This instruction invalidates every cache line by toggling the validity bit associated with that line after writing any relevant information back to memory.

The hardware independent (referred to as the portable-secure) cache flushing function is implemented by allocating a chunk of memory equal to or larger than the size of the hardware's L2 cache (or the largest CPU cache available on the machine). Next, the chunk is divided into cache line sized blocks. When the flushing function is invoked, it iterates over these blocks, altering the data stored in each. The effect is that the cache line associated with each of the blocks is modified and will need to be written back to memory on the next access of that line. This will result in a cache miss as the cache has been overwritten.

The main difference between these two techniques is that the hardware-specific flush will typically invalidate the cache lines, toggling their validity flags and indicate to the CPU that they contain useless data while leaving the data themselves unchanged. By contrast the hardware independent solution will overwrite each cache line (fill them with useless data) but leave the validity flags untouched. Either technique will cause following attempts to access the cache lines to miss, but the implementation is very different.

5.3 Experimental Evaluation

5.3.1 *Objectives*

The evaluation of our secure hypervisors focuses on obtaining the answers to two research questions: "Does this hypervisor prevent sequential cache-based side-channels", and "What is the performance difference between this and the insecure hypervisor". To answer these questions, we have simulated a single-server cloud environment. In this environment, we subject each hypervisor to a conventional sequential cache-based side-channel attack, using the PTP technique, designed for use in cloud environments. In addition, we subject each hypervisor to a series of workloads under different configurations, designed to test different behaviours of the system. The resulting completion times are used to determine a likely increase in overhead.

5.3.2 Environment

The evaluation was conducted using the x86-secure, portable-secure, and insecure hypervisors with a Dom0 running Ubuntu 12.04 and each guest running a sparse installation of Ubuntu 11.04.

Each hypervisor was installed on a Sun Ultra 40 model machine, with 16 GB of RAM and two Opteron 2218 2.6 Ghz processors. Each processor contains two cores, each of which uses its own 1 MB CPU L2 cache. Since this solution is specific to side-channels that do not have concurrent cache access, this machine was selected because it does not have a shared L3 cache. This machine is referred to in subsequent sections as the Sun machine.

For the Apache and 7zip benchmarks, the hypervisors were also run on an IBM X3200 M3 model machine, with 32 GB of RAM and a single Xeon X3430 processor. This processor contains four cores, each of which uses its own 256 MB CPU L2 cache and a shared 8 MB victim L3 cache. In the subsequent sections, this machine is referred to as the IBM machine.

From our experience with cloud systems and examples in the related work [3], we believe these machines are comparable to servers that might be used in a cloud environment.

Unless otherwise stated, all environmental parameters and configurations were left on their default values. It should be noted that in the following section, the term context switch refers to the hypervisor-level process of switching attention between virtual machines, as opposed to any pre-existing definitions.

5.3.3 Side-Channel Prevention

The hypervisors were evaluated using a side-channel attack based on the experiments done by [16] which uses the PTP technique. The side-channel Receiver and Sender programs, performing the functions of the probing instance and the target instance, respectively, were each installed onto separate guest instances. Both programs were executed simultaneously by co-resident guests on the test-bed machine. The attack was designed to send an identifiable string of 160 bits from the target instance to the probing instance. The attack was run ten times on each hypervisor to verify the consistency of the results.

In order to verify each hypervisor's ability to block the side-channel under any conditions, the side-channel was given ideal conditions to work in. Specifically, the probing instance, the target instance, and Dom0 were the only virtual machines running on the hypervisor and the first two were pinned to a single CPU core separate from the core on which Dom0 was pinned. This configuration represents the best possible conditions for a cache-based side-channel attack; any variations would make it more difficult for the attack to succeed. Our experiments assume that if an attack can be prevented under these conditions, then the same prevention would work for environments more hostile to the attack's success. The viability of the defence mechanisms implemented should not be affected by these configurations.

These ideal conditions were applied only to test side-channel prevention. For the performance experiments the scheduler was given free reign over where it moved the VMs except where each experiment specifies.

5.4 Performance Experiments

The overhead generated by each of the secure hypervisors is expected to be directly proportional to the amount of "vulnerable" context switches. Because of the variable nature of cloud workloads, it was necessary to evaluate the hypervisors under a variety of configurations. The hypervisor configurations selected for experimentation address three categories of variation: variations on the type of workload; variations on the Xen Credit scheduler [12] configurations; and variations of the workload's magnitude.

A variation in the type of workload undertaken by the guests in the system can significantly affect the execution of the system. Because of this, the workloads used in the experiments are designed to emulate different types of applications one might find in the cloud. To better accomplish this goal, we evaluate our hypervisors with a combination of established benchmarks and customized benchmarks designed to generate high levels of overhead in the system.

System configurations consist of modifications to the internal variables which control the scheduling of guests' access to the CPUs. These values are used in cloud environments to customize performance to the expected workloads the cloud will process. Different amounts of work imposed concurrently on the system can also significantly affect performance. Cloud systems are designed to manage a dynamic number of clients and their workloads. Variation in the workload magnitude will simulate this attribute.

The performances of the secure hypervisors were compared given a standard benchmark and a customized workload with two different configurations. While multiple benchmarks and customized workloads were initially used [2], the results were similar from one workload to another and therefore only examples are described in this paper. The results of these comparisons are listed at the end of the section.

5.4.1 Measurement of Cache Flushing Overhead

The first round of experimentation was to simply measure the amount of CPU cycles required for the context switch code to execute in each of the three hypervisors. This was done because almost all of the overhead generated by our solution manifests in this functionality, and it serves to establish a good baseline for how much overhead we can expect due to context switching.

The Flush Timing experiments were performed by encompassing the context-switching functionality of the hypervisor in code that records the time-stamp counter in the CPU. We refer to the difference between the readings at the beginning of the function and at the end as the amount of time (in CPU cycles) that was required to perform the context switch, as this is the only section of code that should be affected by our modifications.

TABLE 2
Context Switch Timings (Sun / IBM)

Hypervisor	Thousand Cycles per Switch	% of CPU time
Unmodified	(3 / 3)	(0.02 / 0.03)
Portable-Secure	(1,500 / 100)	(11.54 / 0.83)
x86-Secure	(400 / 1,300)	(3.08 / 10.83)

To run the experiment, we run two guest VMs on each hypervisor, pin them to the same CPU, and proceed to run CPU intensive code on them. The resulting times for the context switches are recorded in the hypervisor and averaged to generate the values shown in Table 2.

It should be noted that these readings were taken using the selective cache flushing algorithm (Algorithm 1) and as a result the only timings considered are those that would be affected by our solution i.e., context switches between VMs that could potentially establish a channel. Depending on the workload, these timings could be frequent, or infrequent, but should always take more time than the default context switching code.

5.4.2 Apache Benchmark with Varying Number of VMs

The Apache benchmark program (AB) is a standard http webserver benchmarking tool [1]. Apache was chosen because it is open source, frequently available, commonly used as a benchmarking service, and represents the type of webservice one would expect to see running in a cloud environment. In our experiments we run the benchmark simultaneously on 1, 2, 4, 8, and 16 co-resident VMs for each system. Each of the VMs have been distributed evenly amongst the CPUs. The benchmark yields an average number of *requests per second* that the system can handle.

5.4.3 Latency Workload with Varying Timeslice Value

The custom Latency workload forks two processes that alternate their execution 500,000 times by use of semaphores. This implementation implies that both processes wake and block 500,000 times throughout one execution. The Latency workload was designed to simulate a high-latency scenario with many system calls, as processes wake and release their use of the CPU.

The Timeslice parameter is a value, in milliseconds, which represents the default amount of consecutive time allocated to a VCPU for execution on a CPU core. Adjusting the Timeslice value allows the user to customize the expected number of context switches to the expected latency of the system. The default value is 30 ms.

5.4.4 Latency Workload with Varying VM Number

The Number of Virtual Machines is the number of guest domains running concurrently on the system, each executing the same load as all others. From the hardware and VM sizes available in canonical clouds (such as Amazon's EC2), we estimate the average number of VMs running on a moderately loaded machine to be somewhere between 6 and 16. For the sake of our experiments, we have selected a VM number of 10, since it allows us to run VMs of canonical size (modelled after EC2) while still maximizing the number of machines within this range. Other researchers [3] have also considered the range of 6-16 to be an acceptable number of VMs with which to model cloud server activity. Despite having this value established as a default, cloud systems are designed to be elastic, dynamically increasing and reducing the number of VMs running on a machine as demand needs. To that extent, we have added values as low as 1 and as high as 50 to our test-bed to use as extreme cases.

Unless otherwise specified, all experiments were conducted under the default configuration of 10 guests, with a Timeslice value of 30 ms.

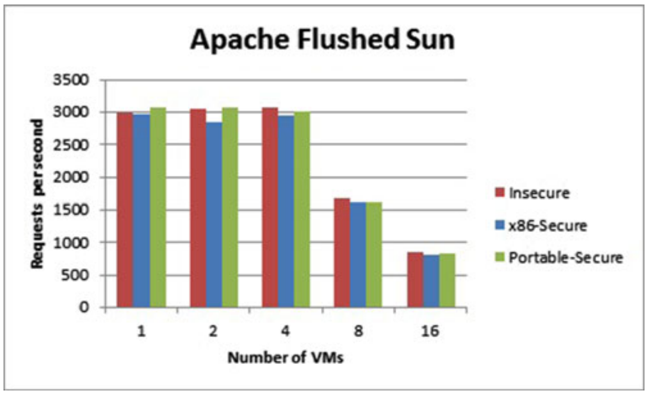


Fig. 3. The Apache benchmark sun with varying # of VMs.

5.5 Results

The attack performed on the insecure hypervisor was able to successfully communicate the entire 160 bit message between instances every time (10/10). This result demonstrates the vulnerability of the unmodified system. By contrast, both of the secure hypervisors yielded 0 bits of successful communication over all twenty attempts. This result suggests that the side-channel could not be successfully established with the countermeasures in place.

Figs. 3 and 4 present the results of each performance benchmark being running simultaneously on a varying number of VMs, as described in the beginning of Section 5.4. Each experiment observed a mean number of Requests per Second over three instances of the benchmark as performed by the Phoronix Test Suite [5]. The results from each VM running in tandem were averaged for the final result shown.

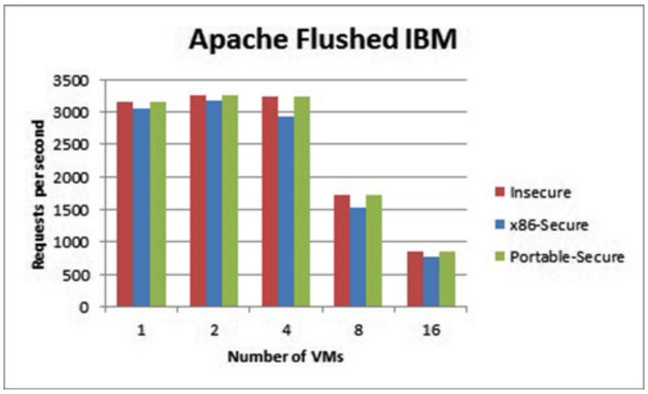
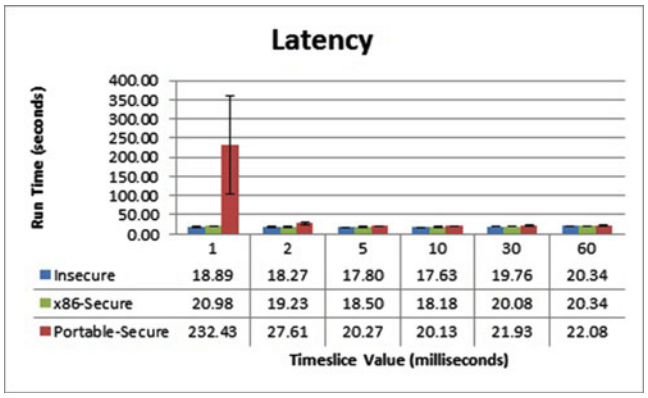
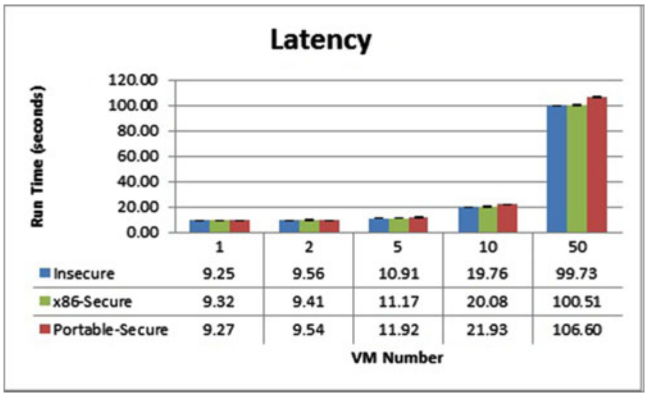


Fig. 4. The Apache benchmark IBM with varying # of VMs.



	1	2	5	10	30	60
🟦 Insecure	18.89	18.27	17.80	17.63	19.76	20.34
🟩 x86-Secure	20.98	19.23	18.50	18.18	20.08	20.34
🟥 Portable-Secure	232.43	27.61	20.27	20.13	21.93	22.08

Fig. 5. Latency workloads on Xen with varying Timeslice.



	1	2	5	10	50
🟦 Insecure	9.25	9.56	10.91	19.76	99.73
🟩 x86-Secure	9.32	9.41	11.17	20.08	100.51
🟥 Portable-Secure	9.27	9.54	11.92	21.93	106.60

Fig. 6. Latency workloads on Xen with varying # of VMs.

Figs. 5 and 6 present the results of each performance experiment with the customized workload described at the end of Section 5.4. Each experiment observed a mean execution time of a customized workload with varying Latency values. The results are as reported by the VMs involved over twenty executions under identical conditions. The means observed for each VM were then averaged to represent the values shown. The error bars represent standard deviations between the means of each VM.

5.5.1 Side-Channel Prevention

As mentioned in Section 5.5, there was no successful side-channel established between two given VMs using either secure hypervisor. This leads us to conclude, that for the attack we have tested against, our modified systems are capable of mitigating sequential cache-based side-channel attacks.

As we were using a representative attack, it is important to determine to exactly what extent we can extrapolate this result. Our experiment includes the most common type of attack (sequential cache-based side-channel) which uses the PTP technique, and is currently the only type known to have done serious damage in the cloud [17]. This particular solution was not designed to prevent an attack exploiting the parallel access of a higher level cache by multiple cores. While the solution may very well serve to inhibit such an attack, any mitigation would be due to cache flushing as interference from third party VMs otherwise not involved in the attack. In such a case, we speculate that our solution would act as some sort of noise amplifier. Likely, it would inhibit the cache channel to a degree proportional to the frequency of context switches on the targeted CPU core. While more secure than the canonical hypervisor, we believe such a defence would be unreliable at best.

Our experiments are restricted to this one representative attack and two test systems. However, the theory holds that our solution should prevent any communication between VMs across a sequential cache-channel. As this attack represents the most basic form of communication, we believe that this experiment stands as a proof of concept that our solution can prevent sequential cache-based side-channels in a cloud environment without interfering with the cloud model.

5.5.2 Measurement of Cache Flushing Overhead

Prior to subjecting the systems to any benchmarking workloads, experiments were run to determine the average number of CPU cycles required for the modified context switch code. The average number of CPU cycles (c) for the hypervisors run on the Sun and IBM machines are reported in Table 2. This table shows the average number of cycles required to run through the __context_switch() code in the hypervisor for each configuration. In addition, it uses the average number of context switches per second (gained empirically) and extrapolates how many cycles per second would be dedicated to context switching using this rate. The final column in the table shows an estimation of how much of the CPU's time is dedicated to context switching using these parameters.

The main purpose of this experiment was to compare the cache flushing techniques on machines with significantly different cache hardware. To this extent, we compare each hypervisor's performance on both machines.

For the Unmodified hypervisor, both machines show a similar overhead per context switch, marked at 3,000 c (CPU cycles). Despite the clock speed difference on these machines, (2.6 Ghz versus 2.4 Ghz) these numbers show a very low amount of overhead in the system (less than 0.05 percent of clock cycles).

For the Portable-Secure hypervisor, the Sun shows 1,500,000 c per switch and the IBM shows 100,000 c. This difference can be attributed to the size of the cache needing to be flushed. Since the portable-secure solution only focuses on flushing the L2 cache, the size difference between the IBM's L2 and the Sun's L2 can be felt. The Sun machine has a 1 MB L2 cache per core, whereas the IBM machine has only a 256 KB L2 per core. Since the IBM machine would need to iterate through one quarter of the memory size, we can expect a significant reduction in the overhead it produces compared to the Sun for this solution. We can further hypothesize that if the experiments were conducted on machines with other sized caches that the overhead would increase or decrease proportionally to the size of the cache. This assumes, of course, that the majority of the overhead is being caused by having to flush more cache, and that it will not be done more efficiently on larger systems. Techniques like pre-fetching or fetching multiple cache lines that are likely to be present in larger-cache systems would likely reduce this predicted overhead.

For the x86-Secure hypervisor, the Sun shows 400,000 c per switch and the IBM is marked at 1,300,000 c. This large difference can also be attributed to the size of the caches needing to be flushed. Unlike the portable-secure solution, the hardware instructions used to flush the cache for the x86-secure solution flush all levels of the cache, this includes the IBM's 8 MB L3 victim cache. Because the Sun's last level cache is the L2, the wbinvd instruction requires less communication and has fewer caches, of smaller size, to flush. The overhead difference between these two systems should demonstrate the effect of flushing different layers of cache.

Expected context switch rates were measured for the insecure system as 200 switches per second (per core) for the Latency workload, performed under the default configurations. Performed on cores with a clock rate of 2.6 Ghz (Sun), this would imply an expected overhead of 11.5 percent of CPU cycles for the Latency workload on the portable-secure system. The same calculations suggest expected overhead of 1.3 percent for the x86-secure system, and comparatively negligible overhead on the insecure system (approximately 1/500th to that of the modified system). The following experiments compare the observed results to this expected overhead.

5.5.3 Apache Benchmark with Varying Number of VMs

The results of running the Apache benchmark on the Sun and IBM machines can be seen in Figs. 3 and 4, respectfully. As can be seen in these figures, there is relatively little overhead shown in this benchmark, with no secure hypervisor yielding less than 90 percent efficiency of the original for any configuration.

In the Sun's performance in Fig. 3 we can see that, with few exceptions, both the x86-secure and the portable-secure hypervisors perform slightly worse, but quite comparably to the unmodified hypervisor. Despite the larger disparity of overhead anticipated from the Cache Flush Overhead experiments, they seem to perform quite similarly. The IBM machine, by comparison, shows more overhead for the x86-secure hypervisor than for the portable-secure. The reasons for this, as explained in Section 5.5.2, is that the portable secure technique needs to flush less cache on the IBM machine than the Sun, whereas the x86 secure techniques need to flush the entire L3 cache.

Overall, the performance for all three hypervisors were relatively close, with comparatively little overhead demonstrated. As a result, we run the Latency experiments in order to determine workloads that may have a more serious negative impact on the hypervisors.

5.5.4 Latency Workloads with Default Values

A comparison of the three hypervisors, using the Latency workload, was run under the aforementioned default values (Timeslice = 30 ms, Number of VMs = 10). The results for the Latency custom benchmark completion times, run on the Sun machine, shows a 11.01 percent increase for the portable-secure system, and a 1.61 percent increase for the x86-modified system. These values can be viewed in Figs. 5 and 6. These results are similar to those predicted.

The results of modifying the Timeslice value can be seen in Fig. 5. As might be expected, modifying the Timeslice value appears to have the largest impact on the completion time of each workload. Most notably, there is a dramatic increase in the completion time on the portable-secure system (1130.47 percent) when Timeslice is set to its minimum value, 1 ms. The 1 ms Timeslice value also generates a surprisingly high (more than 27 times any other Timeslice value) standard deviation. This suggests that some sort of variable condition is arising during the application execution; generating a large amount of overhead. Very likely, this is the result of a CPU thrashing situation, with the overhead for each context switch being magnified 500 fold by the increased context switch overhead of the portable-secure hypervisor. This overhead is reflected in the x86-secure hypervisor as well, but to a much lesser extent (11.06 percent). This is likely due to the same cause, but magnified significantly less due to the reduced context switch time by comparison to the time spent in application execution.

Due to the completely unreasonable level of overhead generated in this configuration, we conclude that the portable-secure hypervisor is not a practical implementation for a cloud system requiring this high a level of latency sensitivity; at least not on a machine with such a large L2 cache. However, less than 15 percent overhead could be considered acceptable for the x86-secure system. This would imply that any implementation of our solution for such latency-sensitive configurations would need to be implemented using hardware specific instructions, or on a system with a smaller cache.

Despite the above conclusion, very few cloud systems would require such a high level of latency optimization (which is the absolute minimum value Timeslice can be set to). Such a latency-sensitive system would also lose much of the benefit of a cache due to its frequent context switching. Considering this, the system would probably be better defended by the complete removal or disabling of the cache, rather than its frequent flushing.

While not encountering the same problems as with a value of 1, the overhead generated by having Timeslice set to 2 is still substantial. For the portable-secure system, generated overhead sits near 50 percent under these configurations. Setting Timeslice to 2 seems to be the turning point at which the additional overhead generated by the portable-secure hypervisor ceases to be considered an acceptable loss. For values greater than 2, the overhead tends to remain below 15 percent, which we deem an acceptable overhead for the additional security provided. The x86-secure system, however, manages to keep the overhead at less than 6 percent for Timeslice values equal to or more than 2. We deem this overhead level acceptable for the increased levels of security.

5.5.5 Latency Workloads with Varying VM Number

The results of modifying the number of guests executing concurrently within the system can be seen in Fig. 6. The most surprising result from experimenting with this variable was the observation that, on the portable-secure system, there was less of an overhead increase with 50 guests than with 10. From our understanding of the Credit Scheduler (the scheduling algorithm used by Xen), this is due to the algorithm's tendency to reduce the number of context switches between VCPUs as there are more VCPUs waiting for execution time. Scheduling this way leads to each VCPU being given less frequent, but longer lasting sequential runs on the CPU core, and therefore reduces the amount of overhead due to context switching. The x86-secure system did not show particularly significant overhead over the course of this experiment.

For smaller numbers of guests, the variation in the VM numbers showed no significant overhead increase from the insecure to secure versions. This is likely due to the fact that no context switches are necessary when there are a smaller number of domains running than CPU cores available.

6 Cache Partitioning

While selectively flushing the cache is an effective way to prevent sequential side-channels, it is not a reliable method of preventing parallel side-channels. Flushing the cache to prevent a side-channel relies on the cache being flushed in between the sender and the receiver attempting to communicate. For a sequential channel, there is a clear ordering to the sending and receiving of messages, i.e., the sender writes a message and, once it is done, the receiver reads the message. Because of the ordering, we have a clear window in which to flush the cache when the context switch between the sender and the receiver occurs.

For a parallel channel, no such obvious window exists. Both the sender and receiver are attempting to communicate over a shared cache by accessing it at the same time. Implementing a flushing function in this scenario would require a flush every time either VM modified the cache. Not only would this make the cache useless, but it would add massive overhead to the system. Rather than flush certain cache lines when they are a threat, we have opted to solve the problem of parallel cache-based side-channels by preventing co-resident VMs from being able to evict one another's data from the cache.

6.1 Solution Overview

For this solution, we partition the shared cache into a number of smaller sections, called slices or partitions, and allow each VM access to a subset of these partitions (usually one). In this format, each cache line belongs to one, and only one, partition. Therefore, if two VMs are given different partitions of the cache they are unable to evict, or otherwise touch, one another's cache lines (or any cache lines outside of their partitions for that matter). The effect of this partitioning on the PTP Technique can be seen in Fig. 7.

As we can see in Fig. 7, there are two cache partitions enforced on the cache data. The one in the red dotted line (Partition 1) is reserved for VM1 (the Probing instance), and the one in the green dotted line (Partition 2) is reserved for VM2 (the Target instance). These partitions map to the first three and last four cache lines out of the seven shown.

When the probing instance (VM1) tries to prime the cache, it is only able to prime the cache lines in its partition, i.e., the first three lines, and has to leave the other four empty. When the triggering instance (VM2) then attempts to modify every other cache line, it can only do so in its own partition and therefore ends up evicting none of VM1's data. When the probing instance once again reads from the cache lines that it is able, it can see no difference from when it left, and therefore no communication was able to occur.

This solution intends to prevent a parallel cache-channel from ever occurring by preventing the VMs from ever sharing the same cache lines. This is markedly different from our sequential solution which prevents sequential channels by actively stopping channels when they have a possibility of forming.

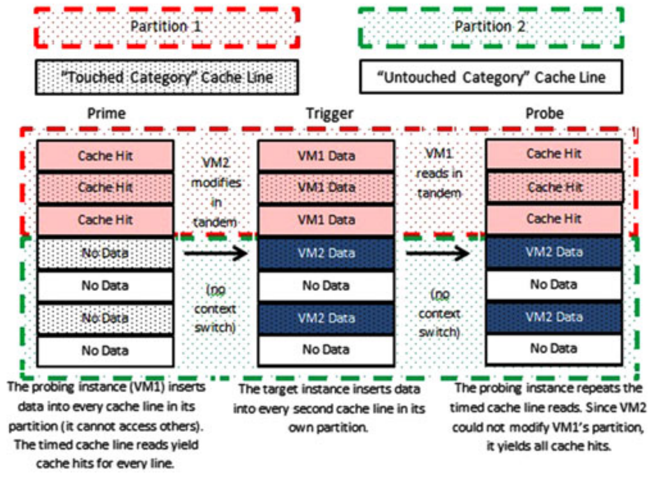

Partition 1
Partition 2
"Touched Category" Cache Line
"Untouched Category" Cache Line
Prime
Trigger
Probe
Cache Hit
Cache Hit
Cache Hit
No Data
No Data
No Data
No Data
VM2 modifies in tandem
(no context switch)
VM1 Data
VM1 Data
VM1 Data
VM2 Data
No Data
VM2 Data
No Data
VM1 reads in tandem
(no context switch)
Cache Hit
Cache Hit
Cache Hit
VM2 Data
No Data
VM2 Data
No Data
The probing instance (VM1) inserts data into every cache line in its partition (it cannot access others). The timed cache line reads yield cache hits for every line.
The target instance inserts data into every second cache line in its own partition.
The probing instance repeats the timed cache line reads. Since VM2 could not modify VM1's partition, it yields all cache hits.

Fig. 7. The effect of partitioning the cache on the PTP technique (parallel).

6.2 Expected Issues and Mitigations

The biggest expected issue in partitioning the cache is the reduced efficiency of the cache's usage. If the solution restricts each VM's usage of the cache to too small a section, it could negatively impact performance. Since the side-channel could be prevented by removing the cache entirely, the goal of this solution is to prevent such channels while still retaining useful efficiency from the cache. Effectively, this makes the task to prevent side-channels from occurring, while incurring as little overhead as possible.

Previous work on cache partitioning [13], however, has shown that intelligent partitioning of the cache can be used to reduce the eviction rate of cache data and actually improve performance. Ultimately, the tradeoff in this situation is that by partitioning the cache, the size of the usable portion of cache for each VM is reduced, but it will also have less useful data evicted by other VMs. As a cache portion gets smaller, there are fewer VMs that can compete for its usage, making it a more efficient portion of cache to work with. If a VM is given a cache of half the size, but does not have to worry about data being evicted from it by foreign VMs, then it may end up yielding a greater cache hit/miss ratio.

The ideal ratio of cache efficiency to size is difficult to determine. Therefore, one of the goals of the experiments in this section is to demonstrate which partitioning schemes may yield more efficient cache usage, while still ensuring side-channel security. To this end we have repeated our experiments with multiple partitioning configurations to determine how they might affect different workloads.

6.3 Cache Partitioning Technique

The implementation of our parallel side-channel solution involves the incorporation of our Cache Partitioning technique into a canonical cloud system. The number of partitions can be configured statically when building the hypervisor. In this section, we will typically refer to the unmodified hypervisor as Default (1) or as Partitioned (1), as the unmodified system considers the cache to be one large entity. The modified hypervisor is referred to as Partitioned (x) where x is the number of partitions into which the cache has been split. Our solution was implemented using the code base for the open source Xen Hypervisor, Version 4.2. The outcome is a new hypervisor which selectively assigns VMs memory based on the partition to which they are assigned.

Our solution partitions the entire memory pool at boot time and as a result the number of partitions are fixed. It does this by assigning all memory pages to partitions based on the least significant bits of their page frame numbers. At boot time, VMs that belong to one partition are only allowed to allocate memory that belongs to the same partition. Ultimately, this may lead to wasted resources, as a single VM running in a four-way partitioned system can only allocate one quarter of the total memory. Even-balancing of loads, however, can make sure that memory resources are maximized. Having access to multiple machines, each with different partition configurations, can guarantee that the loads can be appropriately distributed so as not to encounter a scenario wherein such resources are wasted.

6.3.1 Cache Colouring

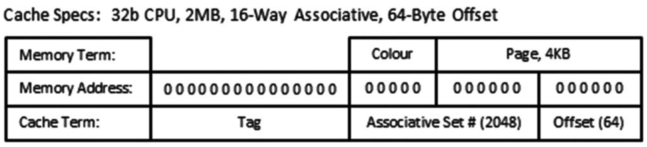
Cache colouring is a process by which memory pages are mapped to cache lines via groupings called *colours*. In reality, the term "colours" just refers to specific aspects of the memory address (usually the left-most bits corresponding to a page frame number). The mapping of memory addresses to cache locations, referred to as *cache lines*) is implemented in the hardware, but is typically universal over systems. The location of a particular memory address in the cache can therefore be known in advance and an operating system, or hypervisor, can select memory addresses to correspond to particular sets of cache lines.

Usually cache colouring is done for cache hit optimization purposes. For instance, much like with virtual page colouring, it makes no sense for two instructions that are consecutive in memory to evict one another in the cache. Cache colouring solves this problem by mapping consecutive memory addresses to separate sections of the cache. Only sufficiently distant memory addresses will be able to evict one another.

Despite its primary use as an optimization agent, the specific mapping of memory addresses to cache lines can be exploited for other purposes. In this case, we intend it to bolster the security of the system by mimicking VM isolation across the cache.

Modern caches have a strict mapping from physical memory to cache lines. Using this we can predict which cache lines any given page will have access to. Because of this mapping, we can therefore assign a VM to a cache partition by restricting its assignable pages to those which correspond with that partition. The manipulation of the mapping of memory pages to cache lines is known as cache colouring.

The cache line/partition a memory address maps to can be determined by looking at its least significant bits. The standard cache map shows a memory address separated into three sections. Starting from the least significant bits these sections are the Cache Block (or Offset), the Index (or Associative Set #), and the Tag.



Cache Specs: 32b CPU, 2MB, 16-Way Associative, 64-Byte Offset

Memory Term:		Colour	Page, 4KB	
Memory Address:	000000000000000	00000	000000	000000
Cache Term:	Tag	Associative Set # (2048)		Offset (64)

Fig. 8. Mapping of a memory address in the cache.

As is visible in Fig. 8, a large section of the Associative Set # (the Index) is pre-defined by the address' page number and cannot be changed using page-level granularity. However, in our given example there remains several bits in the address that overlap with the Associative Set # and do not overlap with the page definition. These bits are what we refer to as the page colour, as only pages that share these 5 bits in common will be able to compete for the same cache lines. For instance, the memory addresses 0x00001000 and 0x11001000 can evict one another as they share the same colour (the same bits for bit positions 15-20), whereas addresses 0x00001000 and 0x00000100 would not.

Using this pattern, the hypervisor can assign pages to guest VMs based on colour, therefore restricting a VM's cache access to a subset that follows that bit combination. In the above example, the hypervisor has up to 32 colours to work with, which means it can effectively partition the cache into a maximum of 32 equal parts of 64 KB. Alternatively, it could choose to restrict the use of a subset of those 5 bits. For instance, if it were to use 4 bits (out of 5) we would have 16 partitions; 3 would yield 8, etc.

Our implementation uses this form of cache colouring to enforce that each VM be restricted to a unique partition, or group of partitions, of the shared cache. For instance, if the system is set to have 16 partitions, then each of, up to, 16 running VMs will be assigned memory from the set that maps to their cache partition. If there are additional VMs, they will need to be assigned within a used partition, or the number of partitions will need to be increased.

Our studies of cloud technologies suggest that the typical cloud machine may run between 6 and 16 VMs simultaneously. Therefore, to maximize security, we suggest partitioning the cache either 8 or 16 times depending on the expected load. At the moment, the number of partitions are declared statically so as to reduce the overhead of having to reassign partitions. Further work may focus on enabling dynamic partitioning to maximize memory/cache usage at a tradeoff to reassignment overhead.

6.4 Experimental Evaluation

6.4.1 Objectives

The evaluation of our secure hypervisors is based on the same objectives and structure as the evaluation done in Section 5.3, with the difference that we focus on parallel instead of sequential side-channels.

6.4.2 Environment

The evaluation was conducted using the default Xen 4.2 hypervisor and our modified version where we have divided the cache into 2, 4, 8, or 16 partitions. The experiments were run on the IBM contexts described in Section 5.3.

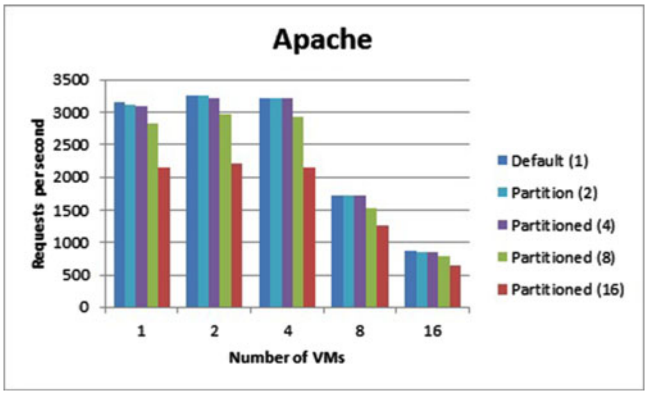


Fig. 9. Apache benchmark.

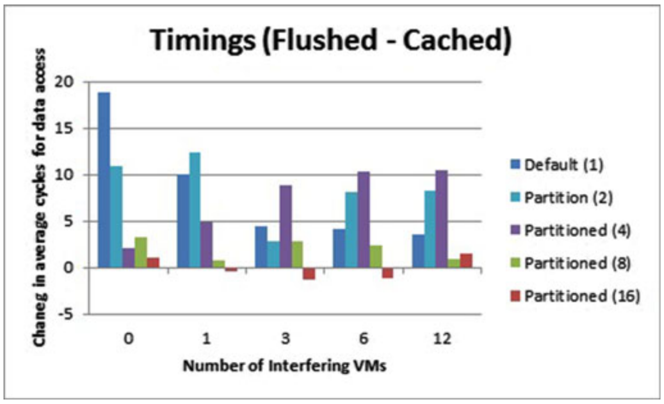


Fig. 10. Cache timing benchmark (flushed-cached).

6.4.3 Side-Channel Prevention

The hypervisors were evaluated using the same side-channel attack used against the Selective Cache Flushing solution in Section 5 but adapted for a parallel environment.

The attack was executed in the same ideal conditions as the sequential attack and makes the same extrapolations about the viability of the defense.

6.5 Performance Experiments

Statically partitioning the cache effectively reduces the size of the cache that each VM has access too. However, it also should prevent any VM from having its cache data evicted by another VM. Due to these conflicting factors, the amount of overhead should vary based on how well the workload lines up with the partitions. Ideally, if the workload is such that each VM is on exactly one partition and needs the cache at the same time then the reduced cache size should not have a negative impact on performance. On the contrary, given these ideal situations we should expect the partitioned hypervisor to provide an increase in performance since the VMs would be prevented from interfering with one another's cache usage. To this extent, we once again run both a standardized and a customized workload to explore their effects on the system.

It should also be noted that the overhead due to the implementation itself will all be executed during the booting process of the VMs (as they are assigned memory) as opposed to any point during their workload execution.

Each of the tests in the following section was conducted using the default hypervisor and the partitioned hypervisor with the amount of partitions set to 2, 4, 8, and 16. The results are discussed in Section 6.6.

6.5.1 Apache Benchmark with Varying Number of VMs and Partitions

The Apache benchmark used in these experiments is the same used in the Flushing tests in the Section 5.5. This benchmark was chosen for both experiments because we believe it represents a believable cloud workload and has been developed as a robust benchmark. Using this same benchmark in both sections also gives us a base on which to compare the overhead generated by both solutions. Based on the design of the benchmark, it is expected that both solutions should have an impact on the performance.

6.5.2 Memory Access Rates with Varying Number of Partitions

The Timings benchmarks are programs of our own design, specifically tuned to observe cache hit-frequency based within a section of memory. In the Cached version, the program assigns a portion of memory larger than the L2, but no larger than the L3 cache (in this case 2 MB). The program then iterates over this memory, storing it in the cache. Next, the program does timed accesses of each of these memory locations and takes the average. The idea is to bias the results in terms of cache hits, determining how quickly the memory can be run through if almost all the cache accesses are hits.

In the Flushed version of this program, everything functions the same except that the cache lines are flushed in between the first and second iterations over the memory (in between the cache priming and the timings). This is to bias the results in favour of cache misses so that we have a comparable baseline.

By comparing the flushed and non-flushed values for the same hypervisor configurations, we should be able to determine how much use of the cache each hypervisor is making.

For the Timing benchmark, the Timing program was run on a single VM which was given uncontested access to a single CPU. For different stages of the experiment we then started additional VMs running a cache contention program on different CPUs. The cache contention programs ran a process that simply attempted to modify as many cache lines as possible as fast as possible, in a similar fashion to the program used to send and receive side-channel messages.

6.6 Results

The attack performed on the insecure hypervisor was able to successfully communicate the entire 20 bit message between instances every time (10/10). This result demonstrates the vulnerability of the unmodified system. By contrast, the partitioned hypervisor yielded 0 bits of successful communication over all twenty attempts. This result suggests that the side-channel could not be successfully established with these countermeasures in place.

Figs. 9 and 10 present the results of each performance experiment given in Section 6.5. Fig. 9 shows the Apache Benchmark with varying numbers of partitions and VMs running the benchmark, and Fig. 10 shows the net difference between runs of the Timings benchmark with varying partitions and VMs. Each experiment observed a mean execution time reported by the VMs involved over twenty workload executions under identical conditions. The means observed for each VM were then averaged to represent the values shown. The error bars represent standard deviations between the means of each VM.

6.6.1 Side-Channel Prevention

As mentioned in Section 6.6, there was no successful side-channel established between two given VMs using the secure hypervisor. This leads us to conclude, that for the attack we have tested against, our modified systems are capable of mitigating sequential cache-based side-channel attacks.

As we were using a representative attack, it is important to determine exactly to what extent we can extrapolate this result. Our experiment uses a modified version of the PTP technique designed to work in a parallel environment. This particular solution was not designed to prevent sequential side-channel attacks and is expected to have no effect on such attacks should they occur. Our empirical analysis only tests the solution on one representative attack, and on a single hardware set. However, the theory states that, if properly implemented, this solution should prevent any shared cache communication between VMs regardless of attack or system. In this case, we believe that our experiment stands as a proof of concept that such a solution can work to prevent parallel side-channels in a cloud environment without interfering with the cloud model as we have defined it.

6.6.2 Apache Benchmark with Varying Number of VMs and Partitions

The first thing to note about the Apache benchmark is that the runs for one, two, and four VMs all involve four or less instances of the benchmark being run on four cores. Once we increase the number of VMs to 8, or 16, we start having to assign multiple benchmarking instances to the same core at the same time. Due to this fact, we would expect to see a massive drop in effectiveness from all instances as we increase to 8 or 16 active VMs. This drop can be seen in Fig. 9 as the requests per second drop by almost a factor of 2 as the amount of VMs increases from 4 to 8 and again from 8 to 16.

Consider Fig. 9. In this figure, we can see that the amount of overhead generated (the reduction in requests per second) as the amount of partitions increases does not increase significantly when the cache is split into 1-4 partitions. However, we do see the overhead increasing for 8, and especially 16, partitions. Considering that the L3 cache on this machine is of 8 MB, when split into four partitions each VM will have 2 MB at its disposal. It seems that this might be a pivotal amount of cache memory for the benchmark because when given 1 MB (partition (8)) or 512 KB (partition (16)) partitions the program slows down. We would expect to see this trend reflected in systems with different amounts of memory as well, with the pivotal point being hit sooner or later based on the size of the shared cache available on the machine. This assumes that the benchmark will require the same amount of memory to perform despite the cache size of the machine it is running on.

Interestingly, as the number of VMs increase, the percentage of overhead generated for the larger number of partitions decreases. This may be because, as there are more benchmarking instances vying for the same cache memory they begin evicting one another's data. Effectively, this means they are not able to make use of the full 2 MB+ cache that would be available to them. This implies that as the amount of VMs increases, there will be less overhead generated by increasing the number of partitions.

6.6.3 Memory Access Rates with Varying Number of Partitions

The first thing to note is that, despite attempts to reduce all non-CPU-cache interference, there are still some bottlenecks in the system that can affect the overhead as the number of partitions change. Factors that can affect these timings can include page contiguity, access to translation look-aside buffer, resource scheduling, and many other shared resources detailed in the software or hardware architecture. For this reason, we have run all of the Timings experiments with a modified program that flushes the expected cache line before each line is accessed for a guaranteed miss. For comparison sake we have included Fig. 10 which directly compares the cached results to the flushed results.

Interestingly, when partitioning the cache two or four ways the partitioned hypervisor out-performs the default hypervisor while there are a large amount of interfering VMs. This is reminiscent of work done by Tam et al. [13] where they demonstrated cache colouring as a performance optimization technique.

When the Timings program is running without interference, the non-partitioned hypervisor can let it have access to the entirety of the cache. This would, as expected, cause it to access data faster because it can generate more cache hits. When run under the partitioned hypervisor the program is only given access to a fraction of the cache. This should increase its overhead accordingly.

When there are VMs running interfering programs on different CPUs, however, they interfere with the Timings program (unless prevented) by evicting data from the shared cache. In the non-partitioned hypervisor this can be seen as the average access time goes up threefold during the transition from 0 to 1 interfering machine. For the partitioned hypervisor, this overhead cannot occur until there are more VMs than partitions. While there can be more VMs than partitions in this system, it should be noted that a side-channel could occur between any two VMs that share a partition, so for maximum security this type of situation should be avoided.

Settings with a larger number of partitions tend to generate more overhead even at their ideal loads. This is likely due to the fact that, as the memory gets excessively partitioned there are less and less contiguous memory pages available causing the memory assigned to a VM becomes fragmented. From Fig. 10, we can expect that the best balance for high loads is somewhere around 4 partitions. However, this solution will only prevent side channels between VMs that are located in different partitions. This performance/security tradeoff is something that the provider would need to consider when configuring their system.

7 Conclusions and Future Work

Two unique security attributes of the cloud motivated this research. First, the cloud's architecture is particularly susceptible to cache-based side-channel attacks. Second, such attacks in the cloud cannot be solved by conventional means without interfering with the cloud model. To address these problems, we have developed, implemented, and evaluated two new techniques designed to prevent cache-based side-channels. One is for dealing with sequential side-channels, and the other for parallel side-channels.

These techniques are implemented entirely within the server (hypervisor) of a cloud system, so as not to interfere with the cloud's methods of operation. Our solutions are unique in that they both address cache-based side-channels in the cloud and do not interfere with the cloud model (require no changes to the client-side code, nor to the underlying hardware).

As shown in our evaluation, our secure hypervisors can effectively prevent sequential cache-based side-channels while generating less than 15 percent overhead, even with the systems configured to our worst case scenario. We show the tradeoffs between hardware specific solutions and those more abstracted and which should be considered based on the hardware being used. We also demonstrate the efficiency of our system on a large L2 cache; as the workload becomes more latency-sensitive the portable system becomes less efficient - generating high levels of overhead for high-latency workloads. As they are designed to deal with hardware directly, Xen and other hypervisors already provide hardware specific installations that can easily be adapted to include efficient side-channel solutions.

In terms of parallel side-channels, our secure hypervisor can effectively prevent parallel cache-based side-channels while generating overhead that is highly dependent on the number of partitions needed. If fewer partitions are needed, the solution can run as fast, or possibly faster, than the insecure hypervisor. If more partitions are needed, then the solution will run with overhead up to 20-30 percent, though highly depending on workload.

The two techniques presented here are able to prevent cache-based side-channels in a cloud environment without interfering with the cloud model. In future work, we believe that the overhead imposed by these solutions can be reduced and a practical solution deployed in an enterprise cloud Environment.

Acknowledgments

The preliminary version of this paper was published in [2].

References

[1] Apache Software Foundation. (2013). Apache http server benchmarking tool [Online]. Available: httpd.apache.org/docs/2.2/programs/ab.html

[2] M. Godfrey and M. Zulkernine, "A server-side solution to cache-based side-channels in the cloud," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 163–170.

[3] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEAL THMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. 21st USENIX Conf. Security Symp.*, Berkeley, CA, USA, 2012, p. 11.

[4] B. W. Lampson. (1973, Oct.). A note on the confinement problem. *Commun. ACM* [Online]. *16(10)*, pp. 613–615. Available: http://doi.acm.org/10.1145/362375.362389

[5] Phoronix Media. (2013). Phoronix test suite [Online]. Available: http://www.phoronix-test-suite.com/

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers' Track RSA Conf. Topics Cryptol.*, 2006, pp. 1–20.

[7] D. Page. (2003). Defending against cache-based side-channel attacks. *Inf. Security Tech. Rep.* [Online]. *8(1)*, pp. 30–44. Available: http://www.sciencedirect.com/science/article/pii/S1363412703001043

[8] S. Pearson, "Privacy, security and trust in cloud computing," in *Privacy and Security for Cloud Computing*, S. Pearson and G. Yee, Eds. London, U.K.: Springer, 2013, pp. 3–42.

[9] 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. 16th ACM Conf. Comput. Commun. Security*, 2009, pp. 199–212.

[10] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw. Workshops*, 2011, pp. 194–199.

[11] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on SSH," in *Proc. 10th Conf. USENIX Security Symp.*, 2001, vol. 10, p. 25.

[12] Citrix Systems. (2012). Xen credit scheduler [Online]. Available: http://wiki.xensource.com/xenwiki/CreditScheduler

[13] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *Proc. Workshop Interact. Oper. Syst. Comput. Archit.*, 2007.

[14] Y. Tsunoo, T. Saito, T. Suzaki, and M. Shigeri, "Cryptanalysis of DES implemented on computers with cache," in *Proc. 5th Int. Workshop Cryptograph. Hardware Embedded Syst.*, 2003, pp. 62–76.

[15] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proc. 21st USENIX Conf. Security Symp.*, 2012, p. 9.

[16] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proc. 3rd ACM Workshop Cloud Comput. Security Workshop*, 2011, pp. 29–40.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 305–316.

Michael (Misiu) Godfrey received the BS and MSc degrees from Queen's University, Kingston, Canada. His research interests include cloud systems with a specialty in security and storage. He also does work with other large-scale systems, and currently works as a compiler developer at IBM Canada.

Mohammad Zulkernine received the BSc degree from the Bangladesh University of Engineering and Technology (BUET), Bangladesh, the MEng degree from Japan, and the PhD degree from Waterloo, Canada. He is a Canada research chair in software dependability and an associate professor at the School of Computing, Queen's University, Canada. He leads the Queen's Reliable Software Technology (QRST) research group. His current research interests include software reliability and security that are sponsored by a number of provincial and federal research funding agencies and industry. He was one of the program cochairs of SSIRI 11, COMPSAC 12, and HASE 14. He is a senior member of the IEEE and the ACM, and a licensed professional engineer in the province of Ontario, Canada.