Scheduler-based Defenses against Cross-VM Side-channels

Venkatanathan Varadarajan
University of Wisconsin

Thomas Ristenpart
University of Wisconsin

Michael Swift
University of Wisconsin

Abstract

Public infrastructure-as-a-service clouds, such as Amazon EC2 and Microsoft Azure allow arbitrary clients to run virtual machines (VMs) on shared physical infrastructure. This practice of multi-tenancy brings economies of scale, but also introduces the threat of malicious VMs abusing the scheduling of shared resources. Recent works have shown how to mount cross-VM side-channel attacks to steal cryptographic secrets. The straightforward solution is *hard isolation* that dedicates hardware to each VM. However, this comes at the cost of reduced efficiency.

We investigate the principle of *soft isolation*: reduce the risk of sharing through better scheduling. With experimental measurements, we show that a *minimum run time (MRT) guarantee* for VM virtual CPUs that limits the frequency of preemptions can effectively prevent existing Prime+Probe cache-based side-channel attacks. Through experimental measurements, we find that the performance impact of MRT guarantees can be very low, particularly in multi-core settings. Finally, we integrate a simple per-core CPU state cleansing mechanism, a form of hard isolation, into Xen. It provides further protection against side-channel attacks at little cost when used in conjunction with an MRT guarantee.

1 Introduction

Public infrastructure-as-a-service (IaaS) clouds enable the increasingly realistic threat of malicious customers mounting side-channel attacks [35, 46]. An attacker obtains tenancy on the same physical server as a target, and then uses careful timing of shared hardware components to steal confidential data. Damaging attacks enable theft of cryptographic secrets by way of shared per-core CPU state such as L1 data and instruction caches [46], despite customers running within distinct virtual machines (VMs).

A general solution to prevent side-channel attacks is *hard isolation*: completely prevent sharing of particular sensitive resources. Such isolation can be obtained by avoiding multi-tenancy, new hardware that enforces cache isolation [42, 44], cache coloring [34, 36], or software systems such as StealthMem [22]. However, hard isolation reduces efficiency and raises costs because of stranded resources that are allocated to a virtual machine yet left unused.

Another approach has been to prevent attacks by adding noise to the cache. For example, in the Düppel system the guest operating system protects against CPU cache side-channels by making spurious memory requests to obfuscate cache usage [47]. This incurs overheads, and also requires users to identify the particular processes that should be protected.

A final approach has been to interfere with the ability to obtain accurate measurements of shared hardware by removing or obscuring time sources. This can be done by removing hardware timing sources [28], reducing the granularity of clocks exposed to guest VMs [41], allowing only deterministic computations [6], or using replication of VMs to normalize timing [26]. These solutions either have significant overheads, as in the last solution, or severely limit functionality for workloads that need accurate timing.

Taking a step back, we note that in addition to sharing resources and having access to fine-grained clocks, shared-core side-channel attacks also require the ability to measure the state of the cache *frequently*. For example, Zhang et al.'s cross-VM attack on ElGamal preempted the victim every 16 µs on average [46]. With less frequent interruptions, the attacker's view of how hardware state changes in response to a victim becomes obscured. Perhaps surprisingly, then, is the lack of any investigation of the relationship between CPU scheduling policies and side-channel efficacy. In particular, scheduling may enable what we call *soft isolation*: limiting the frequency of potentially dangerous cross-VM interactions. (We use

the adjective soft to indicate allowance of occasional failures, analogous to soft real-time scheduling.)

Contributions. We evaluate the ability of system software to mitigate cache-based side-channel attacks through scheduling. In particular, we focus on the type of mechanism that has schedulers ensure that CPU-bound workloads cannot be preempted before a minimum time quantum, even in the presence of higher priority or interactive workloads. We say that such a scheduler offers a *minimum run time (MRT) guarantee*. Xen version 4.2 features an MRT guarantee mechanism for the stated purpose of improving the performance of batch workloads in the presence of interactive workloads that thrash their cache footprint [11]. A similar mechanism also exists in the Linux CFS scheduler [29].

Cache-based side-channel attacks are an example of such highly interactive workloads that thrash the cache. One might therefore hypothesize that by reducing the frequency of preemptions via an MRT guarantee, one achieves a level of soft isolation suitable for mitigating, or even preventing, a broad class of shared-core side-channel attacks. We investigate this hypothesis, providing the first analysis of MRT guarantees as a defense against cache-based side-channel attacks. With detailed measurements of cache timing, we show that even an MRT below 1 ms can defend against existing attacks.

But an MRT guarantee can have negative affects as well: latency-sensitive workloads may be delayed for the minimum time quantum. To evaluate the performance impact of MRT guarantees, we provide extensive measurements with a corpus of latency-sensitive and batch workloads. We conclude that while worst-case latency can be hindered by large MRTs in some cases, in practice Xen's existing core load-balancing mechanisms mitigate the cost by separating CPU-hungry batch workloads from latency-sensitive interactive workloads. As just one example, memcached, when running alongside batch workloads, suffers only a 7% overhead on 95^{th}-percentile latency for a 5 ms MRT compared to no MRT. Median latency is not affected at all.

The existing MRT mechanism only protects CPU-hungry programs that do not yield the CPU or go idle. While we are aware of no side-channel attacks that exploit such victim workloads, we nevertheless investigate a simple and lightweight use of CPU *state cleansing* to protect programs that quickly yield the CPU by obfuscating predictive state. By implementing this in the hypervisor scheduler, we can exploit knowledge of when a cross-VM preemption occurs and the MRT has not been exceeded. This greatly mitigates the overheads of cleansing, attesting to a further value to soft-isolation style mechanisms. In our performance evaluation of this mechanism, we see only a 10–50 µs worse-case overhead on median latency due to cleansing while providing protection for all guest processes within a VM (and not just select ones, as was the case in Düppel). In contrast, other proposed defenses have similar (or worse) overhead but require new hardware, new guest operating systems, or restrict system functionality.

Outline. In the next section, we provide background on cache-based side-channel attacks and existing defense mechanisms. In Section 3 we describe the Xen hypervisor scheduling system, its MRT mechanism, and the principle of soft isolation. In Section 4 we measure the effectiveness of MRT as a defense. Section 5 shows the performance of Xen's MRT mechanism, and Section 6 describes combining MRT with cache cleansing.

2 Background and Motivation

Our work is motivated by the increasing importance of threats posed by side-channel attacks in multi-tenant clouds, in which VMs from multiple customers run on the same physical hardware. We focus on cache-based side-channels, which are dangerous because they can leak secret information such as encryption keys and have been demonstrated between virtual machines in a cloud environment [46].

Side-channel attacks. We can delineate side-channel attacks into three classes: time-, trace-, and access-driven. Time-driven attacks arise when an attacker can glean useful information via repeated observations of the (total) duration of a victim operation, such as the time to compute an encryption (e.g., [5, 7, 10, 16, 24]). Trace-driven attacks work by having an attacker continuously monitor a cryptographic operation, for example via electromagnetic emanations or power usage leaked to the attacker (e.g., [14, 23, 33].

We focus on access-driven side-channel attacks, in which the attacker is able to run a program on the same physical server as the victim. These abuse stateful components of the system shared between attacker and victim program, and have proved damaging in a wide variety of settings, including [3, 15, 31, 32, 35, 45]. In the cross-process setting, the attacker and victim are two separate processes running within the same operating system. In the cross-VM setting, the attacker and victim are two separate VMs running co-resident (or co-tenant) on the same server. The cross-VM setting is of particular concern for public IaaS clouds, where it has been shown that an attacker can obtain co-residence of a malicious VM on the same server as a target [35]

Zhang, Juels, Reiter, and Ristenpart (ZJRR) [46] demonstrated the first cross-VM attack with sufficient

granularity to extract ElGamal secret keys from the victim. They use a version of the classic Prime+Probe technique [31]: the attacker first *primes* the cache (instruction or data) by accessing a fixed set of addresses that fill the entire cache. He then yields the CPU, causing the hypervisor to run the victim, which begins to evict the attacker's data or instructions from various cache. As quickly as possible, the attacker preempts the victim, and then *probes* the cache by again accessing a set of addresses that cover the entire cache. By measuring the speed of each cache access, the attacker can determine which cache lines were displaced by the victim, and hence learn some information about which addresses the victim accessed.

The ZJRR attack builds off a long line of cross-process attacks (c.f., [3, 4, 15, 31, 32]) all of which target per-core microarchitectural state. When simultaneous multi-threading (SMT) is disabled (as is typical in cloud settings), such per-core attacks require that the attacker time-shares a CPU core with the victim. In order to obtain frequent observations of shared state, attacks abuse scheduler mechanisms that prioritize interactive workloads in order to preempt the victim. For example, ZJRR use inter-processor interrupts to preempt every 16 µs on average. In their cross-process attack, Bangerter et al. abuse the Linux process scheduler [15].

Fewer attacks thus far have abused (what we call) off-core state, such as last-level caches used by multiple cores. Some off-core attacks are coarse-grained, allowing attackers to learn only a few bits of information (e.g., whether the victim is using the cache or not [35]). An example of a fine-grained off-core attack is the recent Flush+Reload attack of Yarom and Falkner [45]. Their attack extends the Bangerter et al. attack to instead target last-level caches on some modern Intel processors and has been shown to enable very efficient theft of cryptographic keys in both cross-process and cross-VM settings. However, like the Bangerter et al. attack, it relies on the attacker and victim having shared memory pages. This is a common situation for cross-process settings, but also arises in cross-VM settings should the hypervisor perform memory page deduplication. While several hypervisors implement deduplication, thus far no IaaS clouds are known to use the feature and so are not vulnerable.

Threat model and goals. Our goal is to mitigate or completely prevent cross-VM attacks relevant to modern public IaaS cloud computing settings. We assume the attacker and victim are separate VMs co-resident on the same server running a Type I hypervisor. The attacker controls the entire VM, including the guest operating system and applications, but the hypervisor is run by the cloud provider and is trusted. SMT and memory deduplication are disabled. In this context, the best known attacks rely on:

(1) *Shared per-core state* that is accessible to the attacker and that has visibly different behavior based on its state, such as caches and branch predictors.

(2) *The ability to preempt the victim VM* at short intervals to allow only a few changes to that hardware state.

(3) *Access to a system clock* with enough resolution to distinguish micro-architectural events (e.g., cache hits and misses).

These conditions are all true in contemporary multi-tenant cloud settings, such as Amazon's EC2. Defenses can target any of these dependencies, and we discuss some existing approaches next.

Prior defenses. Past work on defenses against such side-channel attacks identified the above requirements for successful side-channels and tried to obviate one or more of the above necessary conditions for attacks. We classify and summarize these techniques below.

An obvious solution is to prevent an attacker and victim from sharing hardware, which we call *hard isolation*. Partitioning the cache in hardware or software prevents its contents from being shared [22, 34, 36, 42]. This requires special-purpose hardware or loss of various useful features (e.g., large pages) and thus limits the adoption in a public cloud environment. Assigning VMs to run on different cores avoids sharing of per-core hardware [21, 27, 38], and assigning them to different servers avoids sharing of any system hardware [35]. A key challenge here is identifying an attacker and victim in order to separate them; otherwise this approach reduces to using dedicated hardware for each customer, reducing utilization and thus raising the price of computing.

Another form of hard isolation is to reset hardware state when switching from one VM to another. For example, flushing the caches on every context switch prevents the cache state from being shared between VMs [47]. However, this can decrease performance of cache-sensitive workloads both because of the time taken to do the flush and the loss in cache efficiency.

Beyond hard isolation are approaches that modify hardware to add noise, either in the timing or by obfuscating the specific side-channel information. The former can be accomplished by removing or modifying timers [26, 28, 41] to prevent attackers from accurately distinguishing between microarchitectural events, such as a cache hit and a miss. For example, StopWatch [26] removes all timing side-channels and incurs a worse-case overhead of 2.8x for network intensive workloads. Specialized hardware-support could also be used to obfuscate and randomize processor cache usage [25, 44]. All

of these defenses either result in loss of high-precision timer or require hardware changes.

Similarly, one can allocate exclusive memory resources for a sensitive process [22] or add noise to obfuscate its hardware usage [47]. Similarly, programs can be changed to obfuscate access patterns [8, 9]. These approaches are not general-purpose, as they rely on identifying and fixing all security-relevant programs. Worst-case overheads for these mechanisms vary from 6–7%.

Several past efforts attempt to minimize performance interference between workloads (e.g., Q-clouds [30] and Bubble-Up [27], but do not consider adversarial workloads such as side-channel attacks.

3 MRT Guarantees and Soft Isolation

We investigate a different strategy for mitigating per-core side-channels: adjusting hypervisor core scheduling to limit the rate of preemptions. This targets the second requirement of attacks such as ZJRR. Such a scheduler would realize a security design principle that we call *soft isolation*[1]: limiting the frequency of potentially dangerous interactions between mutually untrustworthy programs. Unlike hard isolation mechanisms, we will allow shared state but attempt to use scheduling to limit the damage. Ideally, the flexibility of soft isolation will ease the road to deployment, while still significantly mitigating or even preventing side-channel attacks. We expect that soft isolation can be incorporated as a design goal in a variety of resource management contexts. That said, we focus in the rest of this work on CPU core scheduling.

Xen scheduling. Hypervisors schedule virtual machines much like an operating system schedules processes or threads. Just as a process may contain multiple threads that can be scheduled on different processors, a virtual machine may consist of multiple virtual CPUs (VCPUs) that can be scheduled on different physical CPUs (PCPUs). The primary difference between hypervisor and OS scheduling is that the set of VCPUs across all VMs is relatively static, as VM and VCPU creation/deletion is a rare event. In contrast, processes and threads are frequently created and deleted.

Hypervisor schedulers provide low-latency response times to interactive tasks by prioritizing VCPUs that need to respond to an outstanding event. The events are typically physical device or virtual interrupts from packet arrivals or completed storage requests. Xen's credit scheduler normally lets a VCPU run for 30ms before preempting it so another VCPU can run. However,

[1]The term "soft" is inherited from soft real-time systems, where one similarly relaxes requirements (in that case, time deadlines, in our case, isolation).

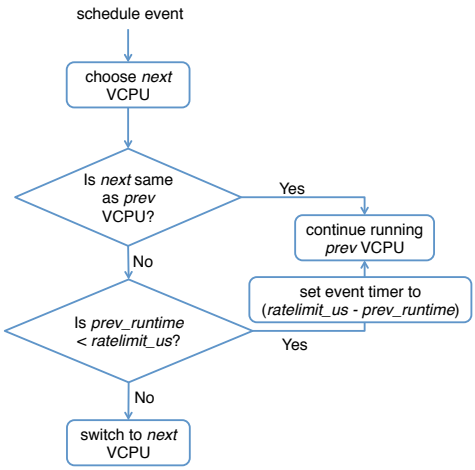


Figure 1: **Logic underlying the Xen MRT mechanism.**

when a VCPU receives an event, it may receive *boost* priority, which allows it to preempt non-boosted VCPUs and run immediately.

VCPUs are characterized by Xen as either *interactive* (or *latency-sensitive*) if they are mostly idle until an interrupt comes in, at which point they execute for a short period and return to idle. Typical interactive workloads are network servers that execute in response to an incoming packet. We refer to VCPUs that are running longer computations as *batch* or *CPU-hungry*, as they typically execute for longer than the scheduler's time slice (30ms for Xen) without idling.

Schedulers can be *work conserving*, meaning that they will never let a PCPU idle if a VCPU is ready to run, or *non-work conserving*, meaning that they enforce strict limits on how much time a VCPU can run. While work-conserving schedulers can provide higher utilization, they also provide worse performance isolation: if one VCPU goes from idle to CPU-hungry, another VCPU on the same PCPU can see its share of the PCPU drop in half. As a result, many cloud environments use non-work conserving schedulers. For example, Amazon EC2's *m1.small* instances are configured to be non-work conserving, allocating roughly 40% of a PCPU (called cap in Xen) to each VCPU of a VM.

Since version 4.2, Xen has included a mechanism for rate limiting preemptions of a VCPU; we call this mechanism a minimum run-time (MRT) guarantee. The logic underlying this mechanism is shown as a flowchart in Figure 1. Xen exposes a hypervisor parameter, `ratelimit_us` (the MRT value) that determines the minimum time any VCPU is guaranteed to run on a PCPU before being available to be context-switched out of the PCPU by another VCPU. One could also rate limit preemptions in other ways, but an MRT guarantee is sim-

ple to implement. Note that the MRT is not applicable to VMs that voluntarily give up the CPU, which happens when the VM goes idle or waits for an event to occur.

As noted previously, the original intent of Xen's MRT was to improve performance for CPU-hungry workloads run in the presence of latency-sensitive workloads: each preemption pollutes the cache and other microarchitectural state, slowing the CPU-intensive workload

Case study. We experimentally evaluate the Xen MRT mechanism as a defense against side-channel leakage by way of soft isolation. Intuitively, the MRT guarantee rate-limits preemptions and provides an attacker less granularity in his observations of the victim's use of per-CPU-core resources. Thus one expects that increased rate-limits decreases vulnerability. To be deployable, however, we must also evaluate the impact of MRT guarantees on benign workloads. In the next two sections we investigate the following questions:

(1) How do per-core side-channel attacks perform under various MRT values? (Section 4)

(2) How does performance vary with different MRT values? (Section 5)

4 Side-channels under MRT Guarantees

We experimentally evaluate the MRT mechanism as a defense against side-channel leakage for per-core state. We focus on cache-based leakage.

Experimental setup. Running on the hardware setup shown in Figure 2, we configure Xen to use two VMs, a victim and attacker. Each has two VCPUs, and we pin one attacker VCPU and one victim VCPU to each of two PCPUs (or cores). We use a non-work-conserving scheduler whose configuration is shown in Figure 9. This is a conservative version of the ZJRR attack setting, where instead the VCPUs were allowed to float — pinning the victims to the same core only makes it easier for the attacker. The hardware and Xen configurations are similar to the configuration used in EC2 m1.small instances [12]. (Although Amazon does not make their precise hardware configurations public, we can still gain some insight into the hardware on which an instance is running by looking at `sysfs` and the `CPUID` instruction.)

Cache-set timing profile. We start by fixing a simple victim to measure the effects of increasing MRT guarantees. We have two functions that each access a (distinct) quarter of the instruction cache (I-cache)[2]. The victim

[2]Our test machine has a 32 KB, 4-way set associative cache with 64-byte lines. There are 128 sets.

Machine Configuration	Intel Xeon E5645, 2.40GHz clock, 6 cores in one package
Memory Hierarchy	Private 32 KB L1 (I- and D-cache), 256 KB unified L2, 12 MB shared L3 and 16 GB main memory.
Xen Version	4.2.1
Xen Scheduler	Credit Scheduler 1
Dom0 OS	Fedora 18, 3.8.8-202.fc18.x86_64
Guest OS	Ubuntu 12.04.3, Linux 3.7.5

Figure 2: **Hardware configuration in local test bed.**

alternates between these two functions, accessing each quarter 500 times. This experiment models a simple I-cache side-channel where switching from one quarter to another leaks some secret information (we call any such leaky function a *sensitive* operation). Executing the 500 access to a quarter of the I-cache requires approximately 100 µs when run in isolation.

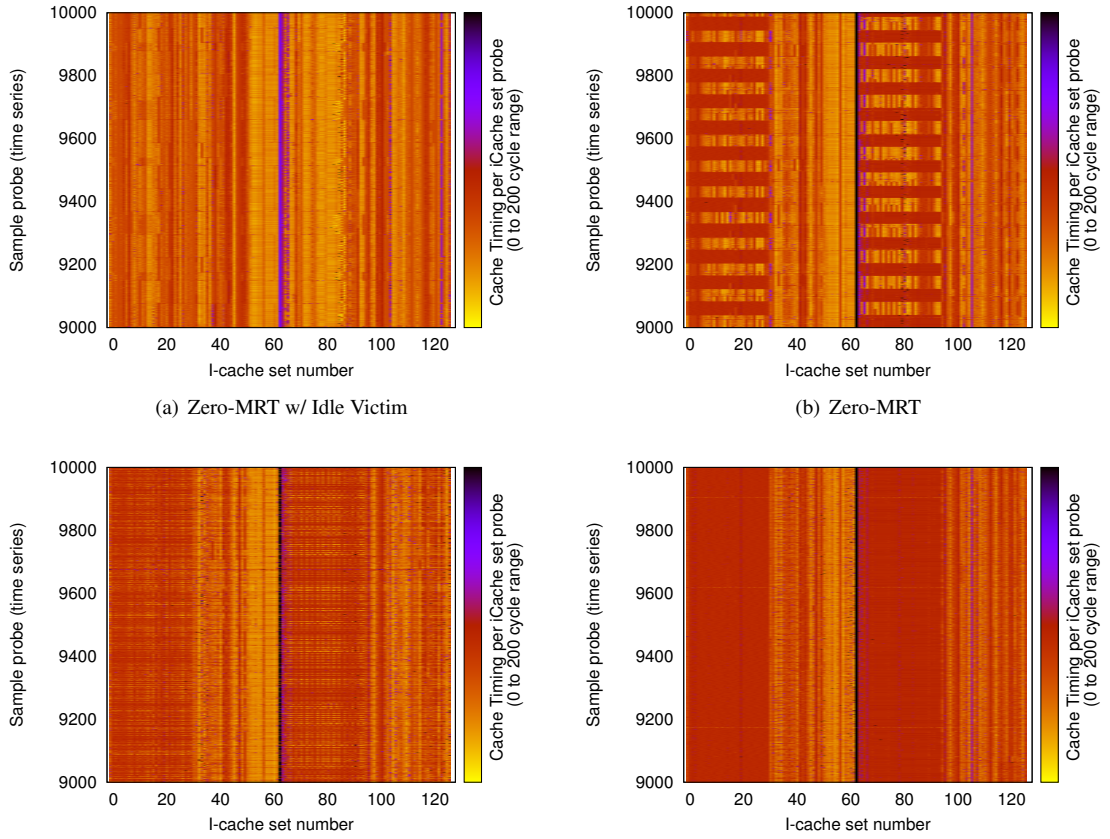
We run this victim workload pinned to a victim VCPU that is pinned to the same PCPU as the attacker VCPU. The attacker uses the IPI-based Prime+Probe technique[3] and measures the time taken to access each I-cache set, similar to ZJRR [46].

Figure 3 shows heat maps of the timings of the various I-cache sets as taken by the Prime+Probe attacker, for various MRT values between 0 (no MRT) and 5 ms. Darker colors are longer access times, indicating conflicting access to the cache set by the victim. One can easily see the simple alternating pattern of the victim as we move up the y-axis of time in Figure 3(b). Also note that this is different from an idle victim under zero-MRT shown in Figure 3(a). With no MRT, the attacker makes approximately 40 observations of each cache set, allowing a relatively detailed view of victim behavior.

As the MRT value increases we see the loss of resolution by the attacker as its observations become less frequent than the alternations of the victim. At an MRT of 100 µs the pattern is still visible, but noisier. Although the victim functions run for 100 µs, the prime+probe attacker slows downs the victim by approximately a factor of two, allowing the pattern to be visible with a 100 µs MRT. When the MRT value is set to 1 ms the attacker obtains no discernible information on when the switching between each I-cache set happens.

In general, an attacker can observe victim behavior that occurs at a lower frequency than the attacker's preemptions. We modify the victim program to be 10x

[3]Note that the attacker requires two VCPUs, one measuring the I-cache set timing whenever interrupted and the other issuing the IPIs to wake up the other VCPU. The VCPU issuing IPIs is pinned to a different PCPU.



(a) Zero-MRT w/ Idle Victim

(b) Zero-MRT

(c) 100 µs-MRT

(d) 1 ms-MRT

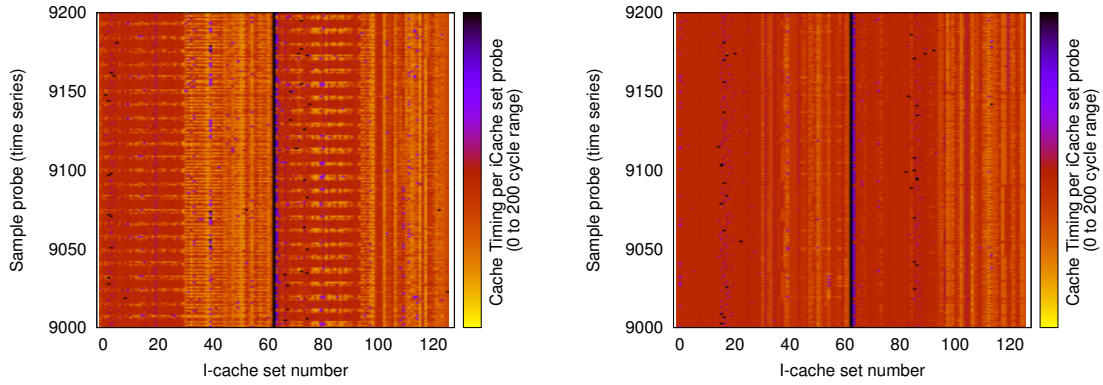
Figure 3: **Heatmaps of I-cache set timing as observed by a prime-probe attacker.** Displayed values are from a larger trace of 10,000 timings. (a) Timings for idle victim and no MRT. (b)–(d) Timings for varying MRT values with the victim running.

slower (where each function takes approximately 1 ms standalone). Figure 4 shows the result for this experiment. With a 1 ms MRT, we observe the alternating pattern. When the MRT is raised to 5 ms, which is longer than the victim's computation (≈ 2 ms), no pattern is apparent. Thus, when the MRT is longer than the execution time of a security-critical function this side-channel fails.

While none of this proves lack of side-channels, it serves to illustrate the dynamics between side-channels, duration of sensitive victim operations, and the MRT: as the MRT increases, the frequency with which an attacker can observe the victim's behavior decreases, and the signal and hence leaked information decreases. All this exposes the relationship between the speed of a sensitive operation, the MRT, and side-channel availability for an attacker. In particular, very long operations (e.g., longer than the MRT) may still be spied upon by side-channel attackers. Also, infrequently accessed but sensitive memory accesses may leak to the attacker. We hypothesis that at least for cryptographic victims, even moderate MRT values on the order of a handful of milliseconds are sufficient to prevent per-core side-channel attacks. We next look, therefore, at how this relationship plays out for cryptographic victims.

ElGamal victim. We fix a victim similar to that targeted by ZJRR. The victim executes the modular exponentiation implementation from libgcrypt 1.5.0 using a 2048-bit exponent, base and modulus, in a loop. Pseudocode of the exponentiation algorithm appears in Figure 5. One can see that learning the sequence of operations leaks the secret key values: if the code in lines 7 and 8 is executed, the bit is a 1; otherwise it is a zero. We instrument libgcrypt to write the current bit being operated upon to a memory page shared with the attacker, allowing us to determine when preemptions occur relative to operations within the modular exponentiation.

For no MRT guarantee, we observe that the attacker can preempt the victim many times per individual square, multiply, or reduce operation (as was also reported by ZJRR). With MRT guarantees, the rate of preemptions drops so much that the attacker only can interrupt once



(a) 1 ms-MRT (b) 5 ms-MRT

Figure 4: **Heatmaps of I-cache set timings as observed by a prime-probe attacker for 10x slower victim computations.** Displayed values are from a larger trace of 9,200 timings.

```
SQUAREMULT(x, e, N):
 1: Let e_n, ..., e_1 be the bits of e
 2: y ← 1
 3: for i = n down to 1 do
 4:     y ← SQUARE(y)
 5:     y ← MODREDUCE(y, N)
 6:     if e_i = 1 then
 7:         y ← MULT(y, x)
 8:         y ← MODREDUCE(y, N)
 9:     end if
10: end for
11: return y
```

Figure 5: **Modular exponentiation algorithm used in libgcrypt version 1.5.0.** Note that the control flow followed when $e_i = 1$ is lines $4 \to 5 \to 6 \to 7 \to 8$ and when $e_i = 0$ is lines $4 \to 5$; denoted by the symbols `1` and `0`, respectively.

Xen MRT (ms)	Avg. ops/run	Min. ops/run
0	0.096	0
0.1	14.1	4
0.5	49.0	32
1.0	92.6	68
2.0	180.7	155
5.0	441.2	386
10.0	873.1	728

Figure 6: **The average and minimal number of ElGamal secret key bits operated upon between two attacker preemptions for a range of MRT values.** Over runs with 40K preemptions.

Xen MRT (ms)	Preemptions per function call		
	Min	Median	Max
0	3247	19940	20606
0.1	74	155	166
0.5	22	42	47
1.0	16	22	25
2.0	10	11	13
5.0	0	4	6
10.0	1	2	3

Figure 7: **Rate of preemption with various MRT.** Here the function called is the Modular-Exponentiation implementation in libgcrypt with a 2048 bit exponent. Note that for zero MRT the rate of preemption is very high that victim computation involving a single bit was preempted multiple times.

every several iterations of the inner loop. Figure 6 gives the number of bits operated on between attacker preemptions for various MRT values. Figure 7 gives the number of preemptions per entire modular exponentiation computation. We see that for higher MRT values, the rate of preemption per call to the full modular exponentiation reduces to just a handful. The ZJRR attack depends on multiple observations *per operation* to filter out noise, so even at the lowest MRT value of 100 µs, with 4–14 operations per observation, the ZJRR attack fails. In the full version [40], we discuss how one might model this leakage scenario formally and evidence a lack of any of a large class of side-channel attacks.

AES victim. We evaluate another commonly exploited access-driven side-channel victim, AES, which leaks secret information via key-dependent indexing into tables stored in the L1 data cache [15, 31]. The previous attacks, all in the cross-process setting, depend on observing a very small number of cache accesses to obtain a clear signal of what portion of the table was accessed by the victim. Although there has been no known AES attack in the cross-VM setting (at least when deduplication is turned off, otherwise see [19]), we evaluate effectiveness of MRT against the best known IPI Prime+Probe

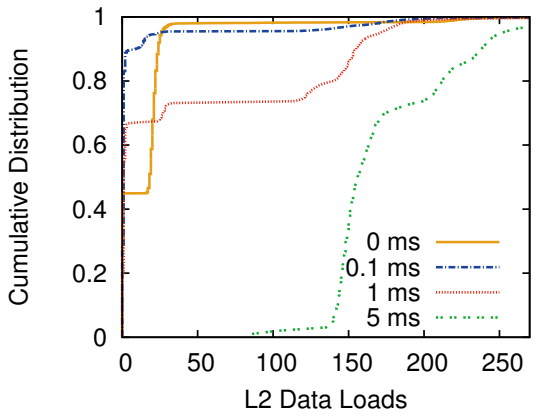


Figure 8: **CDF of L2 data-loads per time slice experienced by OpenSSL-AES victim.** L2 data loads are performance counter events that happen when a program requests for a memory word that is not in both L1 and L2 private caches (effectively a miss). When running along-side a Prime+Probe attacker, these data-cache accesses can be observed by the attacker.

spy process due to ZJRR. In particular, we measured the number of private data-cache misses possibly observable by this Prime+Probe attacker when the victim is running AES encryption in a loop.

To do so, we modified the Xen scheduler to log the count of private-cache misses (in our local testbed both L1 and L2 caches are private) experienced by any VCPU during a scheduled time slice. This corresponds to the number of data-cache misses an attacker could ideally observe. Figure 8 shows the cumulative distribution of the number of L2-data cache misses (equivalently, private data-cache loads) during a time slice of the victim running OpenSSL-AES. We can see that under no or lower MRTs the bulk of time slices suffer only a few tens of D-cache misses that happen between two back-to-back preemptions of the attacker. (We note that this is already insufficient to perform prior attacks.) The number of misses increases to close to 200 for an MRT value of 5 ms. This means that the AES process is evicting its own data, further obscuring information from a would-be attacker. Underlying this is the fact that the number of AES encryptions completed between two back-to-back preemptions increases drastically with the MRT: found that thousands to ten thousands AES block-encryptions were completed between two preemptions when MRT was varied from 100 µs to 5 ms, respectively.

Summary. While side channels pose a significant threat to the security of cloud computing, our measurements in this section show that, fortunately, the hypervisor scheduler can help. Current attacks depend on frequent preemptions to make detailed measurements of cache contents. Our measurements show that even delaying preemption for a fraction of millisecond prevents known attacks. While this is not proof that future attacks won't be found that circumvent the MRT guarantee, it does strongly suggest that deploying such a soft-isolation mechanism will raise the bar for attackers. This leaves the question of whether this mechanism is cheap to deploy, which we answer in the next section.

Note that we have focused on using the MRT mechanism for CPU and, indirectly, per-core hardware resources that are shared between multiple VMs. But rate-limiting-type mechanisms may be useful for other shared devices like memory, disk/SSD, network, and any system-level shared devices which suffer from a similar access-driven side-channels. For instance, a timed disk read could reveal user's disk usage statistics like relative disk head positions [20]. Fine-grained sharing of the disk across multiple users could leak sensitive information via such a timing side-channel. Reducing the granularity of sharing by using MRT-like guarantees in the disk scheduler (e.g., servicing requests from user for at least T_{min}, minimum service time, before serving requests from another user) would result in a system with similar security guarantees as above, eventually making such side-channels harder to exploit. Further research is required to analyze the end-to-end performance impact of such a mechanism for various shared devices and schedulers that manage them.

5 Performance of MRT Mechanism

The analysis in the preceding section demonstrates that MRT guarantees can meaningfully mitigate a large class of cache-based side-channel attacks. The mitigation becomes better as MRT increases. We therefore turn to determining the maximal MRT guarantee one can fix while not hindering performance.

5.1 Methodology

We designed experiments to quantify the negative and positive effects of MRT guarantees as compared to a baseline configuration with no MRT (or zero MRT). Our testbed configuration uses the same hardware as in the last section and the Xen configurations are summarized in Figure 9. We run two DomU VMs each with a single VCPU. The two VCPUs are pinned to the same PCPU. Pinning to the same PCPU serves to isolate the effect of the MRT mechanism. The management VM, Dom0, has 6 VCPUs, one for each PCPU (a standard configuration option). The remaining PCPUs in the system are otherwise left idle.

Work-conserving configuration

Dom0	6 VCPU / no cap / weight 256
DomU	1 VCPU / 2 GB memory / no cap / weight 256

Non-work-conserving configuration

Dom0	6 VCPU / no cap / weight 512
DomU	1 VCPU / 2 GB memory / 40% cap / weight 256

Figure 9: **Xen configurations used for performance experiments.**

CPU-hungry Workloads

Workload	Description
SPECjbb	Java-based application server [37]
graph500	Graph analytics workload [1] with scale of 18 and edge factor of 20.
mcf, *sphinx*, *bzip2*	SpecCPU2006 cache sensitive benchmarks [17]
Nqueens	Microbenchmark solving n-queens problem
CProbe	Microbenchmark that continuously trashes L2 private cache.

Latency-sensitive Workloads

Workload	Description
Data-Caching	Memcached from Cloud Suite-2 with twitter data set scaled by factor of 5 run for 3 minutes with rate of 500 requests per second [13].
Data-Serving	Cassandra KV-store from Cloud Suite-2 with total of 100K records[4] [13]
Apache	Apache webserver, HTTPing client [18], single 4 KB file at 1 ms interval.
Ping	Ping command at 1 ms interval.
Chatty-CProbe	One iteration of *CProbe* every 10 µs.

Figure 10: **Workloads used in performance experiments.**

We use a mix of real-world applications and microbenchmarks in our experiments (shown in Figure 10). The microbenchmark *CProbe* simulates a perfectly cache-sensitive workload that continuously overwrites data to the (unified) L2 private cache, and *Chatty-CProbe* is its interactive counterpart that overwrites the cache every 10 µs and then sleeps. We also run the benchmarks with an idle VCPU (labeled *Idle* below).

5.2 Latency Sensitivity

The most obvious potential performance downside of a MRT guarantee is increased latency: interactive workloads may have to wait before gaining access to a PCPU. We measure the negative effects of MRT guarantees by running latency-sensitive workloads against *Nqueens* (a CPU-bound program with little memory access). Figure 11 shows the 95^{th} percentile latency for the interactive workloads. The baseline results are shown as a MRT of 0 on the X-axis. As expected, the latency is approximately equal to the MRT for almost all workloads (*Apache* has higher latency because it requires multiple packets to respond, so it must run multiple times to complete a request). Thus, in the presence of a CPU-intensive workload and when pinned to the same PCPU, the MRT can have a large negative impact on interactive latency.

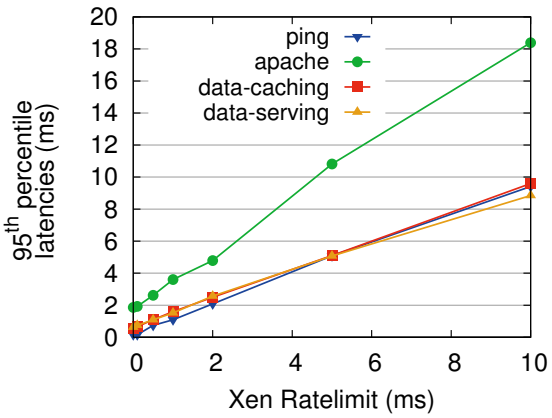


Figure 11: **95^{th} Percentile Latency of Various Latency Sensitive Workloads.** Under non-work-conserving scheduling.

As the workloads behave essentially similarly, we now focus on just the *Data-Caching* workload. Figure 12 shows the response latency when run against other workloads. For the two CPU-intensive workloads, *CProbe* and *Nqueens*, latency increases linearly with the MRT. However, when run against either an idle VCPU or *Chatty-CProbe*, which runs for only a short period, latency is identical across all MRT values. Thus, the MRT has little impact when an interactive workload runs alone or it shares the PCPU with another interactive workload.

We next evaluate the extent of latency increase. Figure 13 shows the 25^{th}, 50^{th}, 75^{th}, 90^{th}, 95^{th} and 99^{th} percentile latency for *Data-Caching*. At the 50^{th} percentile and below, latency is the same as with an idle VCPU. However, at the 75^{th} latency rises to half the MRT, indicating that a substantial fraction of requests are delayed.

We repeated the above experiments for the work-conserving setting, and the results were essentially the same. We omit them for brevity. Overall, we find that enforcing an MRT guarantee can severely increase latency when interactive VCPUs share a PCPU with CPU-intensive workloads. However, they have limited impact when multiple interactive VCPUs share a PCPU.

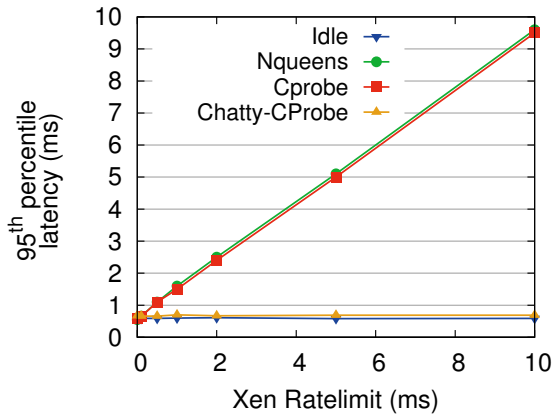


Figure 12: **95th Percentile Request Latency of *Data-Caching* Workload with Various Competing Micro-benchmarks.** Under non-work-conserving scheduling.

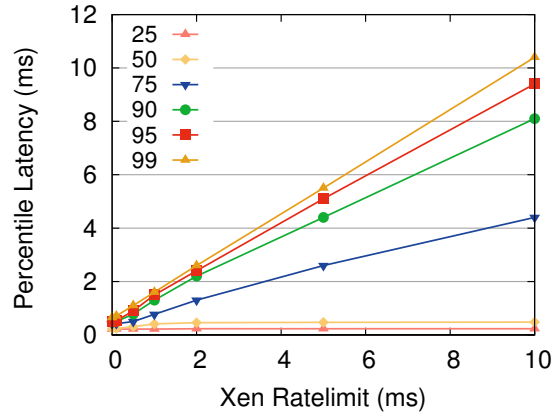


Figure 13: **25th, 50th, 75th, 90th, 95th and 99th Percentile Latency of *Data-Caching* Workload when run alongside *Nqueens*.** Under non-work-conserving scheduling.

5.3 Batch Efficiency

In addition to measuring the impact on latency-sensitive workloads, we also measure the impact of MRT guarantees on CPU-hungry workloads. The original goal of the MRT mechanism was to reduce frequent VCPU context-switches and improve performance of batch workloads. We pin a CPU-hungry workload to a PCPU against competing microbenchmarks.

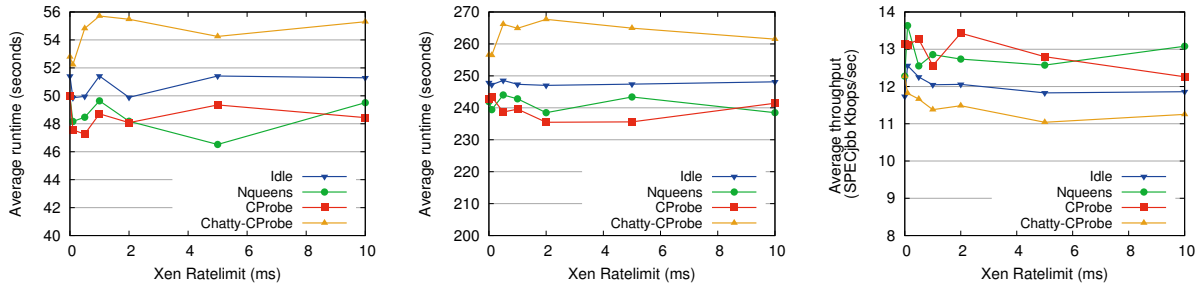
Figure 14 shows the effect of MRT values on the *graph500* workload when run alongside various competing workloads. Because this is work-conserving scheduling, the runtime of *graph500* workload increases by roughly a factor of two when run alongside *Nqueens* and *CProbe* as compared to *Idle*, because the share of the PCPU given to the VCPU running *graph500* drops by one half. The affect of MRT is more pronounced when looking running alongside *Chatty-CProbe*, the workload which tries to frequently interrupt *graph500* and trash its cache. With no MRT guarantee, this can double the runtime of a program. But with a limit of only 0.5 ms, performance is virtually the same as with an idle VCPU, both because *Chatty-CProbe* uses much less CPU and because it trashes the cache less often.

With a non-work-conserving scheduler, the picture is significantly different. Figure 15 shows the performance of three batch workloads when run alongside a variety of other workloads, for various MRT values. First, we observe that competing CPU-bound workloads such as *Nqueens* and *CProbe* do not significantly affect the performance of CPU-bound applications, even in the case of *CProbe* that trashes the cache. This occurs because the workloads share the PCPU at coarse intervals (30 ms), so the cache is only trashed once per 30 ms period. In contrast, when run with the interactive workload *Chatty-CProbe*, applications suffer up to 4% performance loss, which *increases* with longer MRT guarantees. Investigating the scheduler traces showed that under zero MRT the batch workload enjoyed longer scheduler time slices of 30 ms compared to the non-zero MRT cases. This was because under zero MRT highly interactive *Chatty-CProbe* quickly exhausted Xen's boost priority. After this, *Chatty-CProbe* could not preempt and waited until the running VCPU's 30 ms time slice expires. With longer MRT values, though, *Chatty-CProbe* continues to preempt and degrade performance more consistently.

Another interesting observation in Figure 15 is that when the batch workloads share a PCPU with an idle VCPU, they perform worse than when paired with *Nqueens* or *CProbe*. Further investigation revealed that an idle VCPU is not completely idle but wakes up at regular intervals for guest timekeeping reasons. Overall, under non-work-conserving settings, running a batch VCPU with any interactive VCPU (even an idle one) is worse than running with another batch VCPU (even one like *CProbe* that trashes the cache).

5.4 System Performance

The preceding sections showed the impact of MRT guarantees when both applications are pinned to a single core. We next analyze the impact of the Xen scheduler's VCPU placement policies, which choose the PCPU on which to schedule a runnable VCPU. We configure the system with 4 VMs each with 2 VCPUs to run on 4 PCPUs under a non-work conserving scheduler. We run three different sets of workload mixes, which together capture a broad spectrum of competing workload combinations. Together with a target workload running on both VCPUs of a single VM, we run: (1) *All-Batch* — consisting of worst-case competing CPU-hungry work-



(a) *mcf* (b) *graph500* (c) *SPECjbb*

Figure 15: **Average runtime of various batch workloads under non-work conserving setting.** Note that for *SPECjbb* higher is better (since the graph plots the throughput instead of runtime). All data points are averaged across 5 runs.

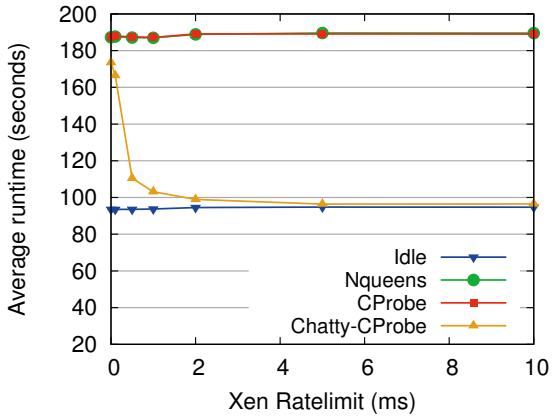


Figure 14: **Average runtime of *graph500* workload when run alongside various competing workloads and under work-conserving scheduling.** Averaged over 5 runs.

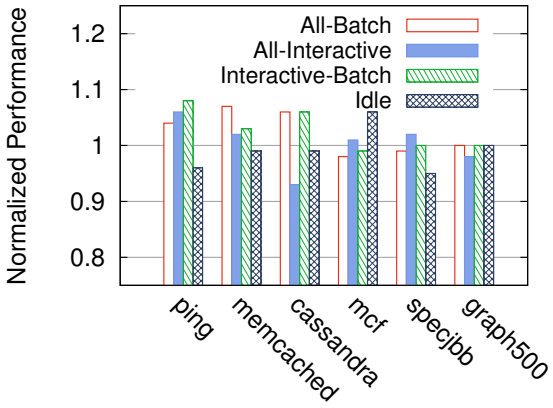


Figure 16: **Normalized Performance in a non-work-conserving configuration with 5 ms MRT.** Normalized to performance under zero-MRT case. The left three workloads report 95^{th} percentile latency and the right three report runtime, averaged across 5 runs. In both cases lower is better.

load (*CProbe*); (2) *All-Interactive* — consisting of worst-case competing interactive workload (*Chatty-CProbe*); and (3) *Batch & Interactive* — where half of other VCPUs run *Chatty-CProbe* and half *CProbe*. We compare the performance of Xen without MRT to running with the default 5ms limit. The result of the experiment is shown in Figure 16. For interactive workloads, the figure shows the relative 95^{th} percentile latency, while for CPU-hungry workloads it shows relative execution time.

On average across the three programs and three competing workloads, latency-sensitive workloads suffered on average of only 4% increase in latency with the MRT guarantee enabled. This contrasts sharply with the 5-fold latency increase in the pinned experiment discussed earlier. CPU-hungry workloads saw their performance improve by 0.3%. This makes sense given the results in the preceding section, which showed that an MRT guarantee offers little value to batch jobs in a non-work-conserving setting.

To understand why the latency performance is so much better than our earlier results would suggest, we analyzed a trace of the scheduler's decisions. With the non-work-conserving setting, Xen naturally segregates batch and interactive workloads. When an interactive VCPU receives a request, it will migrate to an idle PCPU rather than preempt a PCPU running a batch VCPU. As the PCPU running interactive VCPUs is often idle, this leads to coalescing the interactive VCPUs on one or more PCPUs while the batch VCPUs share the remaining PCPUs.

5.5 Summary

Overall, our performance evaluation shows that the strong security benefits described the in Section 4 can be achieved at low cost in virtualized settings. Prior research suggests more complex defense mechanisms [22, 26, 28, 43, 47] that achieve similar low performance overheads but at a higher cost of adoption, such as sub-

stantial hardware changes or modifications to security-critical programs. In comparison, the MRT guarantee mechanism is simple and monotonically improves the security against many existing side-channel attacks with zero cost of adoption and low overhead.

We note that differences between hypervisor scheduling and OS scheduling mean that the MRT mechanism cannot be as easily applied by an operating system to defend against malicious processes. As mentioned above, a hypervisor schedules a small and relatively static number of VCPUS onto PCPUs. Thus, it is feasible to coalesce VCPUs with interactive behavior onto PCPUs separate from those running batch VCPUs. Furthermore, virtualized settings generally run with *share-based* scheduling, where each VM or VCPU is assigned a fixed share of CPU resources. In contrast, the OS scheduler must schedule an unbounded number of threads, often without assigned shares. Thus, there may be more oversubscription of PCPUs, which removes the idle time that allows interactive VCPUs to coalesce separately from batch VCPUs. As a result, other proposed defenses may still be applicable for non-virtualized systems, such as PaaS platforms that multiplex code from several customers within a single VM [2].

6 Integrating Core-State Cleansing

While the MRT mechanism was shown to be a cheap mitigation for protecting CPU-hungry workloads, it may not be effective at protecting interactive ones. If a (victim) VCPU yields the PCPU quickly, the MRT guarantee does not apply and an attacker may observe its residual state in the cache, branch predictor, or other hardware structures. We are unaware of any attacks targeting such interactive workloads, but that is no guarantee future attacks won't.

We investigate incorporating per-core state-cleansing into hypervisor scheduling. Here we are inspired in large part by the Düppel system [47], which was proposed as a method for guest operating systems to protect themselves by periodically cleansing a fraction of the L1 caches. We will see that by integrating a selective state-cleansing (SC) mechanism for I-cache, D-cache and branch predictor states into a scheduler that already enforces an MRT guarantee incurs much less overhead than one might expect. When used, our cleansing approach provides protection for all processes within a guest VM (unlike Düppel, which targeted particular processes).

6.1 Design and Implementation

We first discuss the cleansing process, and below discuss when to apply it. The cleanser works by executing a specially crafted sequence of instructions that together overwrite the I-cache, D-cache, and branch predictor states of a CPU core. A sample of these instructions is shown in Figure 17; these instructions are 27 bytes long and fit in a single I-cache line.

In order to overwrite the branch predictor or the Branch Target Buffer (BTB) state, a branch instruction conditioned over a random predicate in memory is used. There are memory move instructions that add noise to the D-cache state as well. The last instruction in the set jumps to an address that corresponds to the next way in the same I-cache set. This jump sequence is repeated until the last way in the I-cache set is accessed, at which point it is terminated with a `ret` instruction. These instructions and the random predicates are laid out in memory buffers that are equal to the size of the I-cache and D-cache, respectively. Each invocation of the cleansing mechanism randomly walks through these instructions to touch all I-cache sets, D-cache sets, and flush the BTB.

We now turn to how we have the scheduler decide when to schedule cleansing. There are several possibilities. The simplest strategy would be to check, when a VCPU wakes up, if the prior running VCPU was from another VM and did not use up its MRT. If so, then run the cleansing procedure before the incoming VCPU. We refer to this strategy as *Delayed-SC* because we defer cleansing until a VCPU wants to execute. This strategy guarantees to cleanse only when needed, but has the downside of potentially hurting latency-sensitive applications (since the cleanse has to run between receiving an interrupt and executing the VCPU). Another strategy is to check, when a VCPU relinquishes the PCPU before its MRT guarantee expires, whether the next VCPU to run is from another domain or if the PCPU will go idle. In either case, a cleansing occurs before the next VCPU or idle task runs. Note that we may do unnecessary cleansing here, because the VCPU that runs after idle may be from the same domain. We therefore refer to this strategy as *Optimistic-SC*, given its optimism that a cross-VM switch will occur after idle. This optimism may pay off because idle time can be used for cleansing.

Note that the CPU time spent in cleansing in Delayed-SC is accounted to the incoming VCPU but it is often free with Optimistic-SC as it uses idle time for cleansing when possible.

6.2 Evaluation

We focus our evaluation on latency-sensitive tasks: because we only cleanse when an MRT guarantee is not hit, CPU-hungry workloads will only be affected minimally by cleansing. Quantitatively the impact is similar to the results of Section 5 that show only slight degradation due to *Chatty-CProbe* on CPU-hungry workloads.

We use the hardware configuration shown in Figure 2.

```
000 <L13-0xd>:
  0: 8b 08           mov    (%rax),%ecx
  2: 85 c9           test   %ecx,%ecx
  4: 74 07           je     d <L13>
  6: 8b 08           mov    (%rax),%ecx
  8: 88 4d ff        mov    %cl,-0x1(%rbp)
  b: eb 05           jmp    12 <L14>

00d <L13>:
  d: 8b 08           mov    (%rax),%ecx
  f: 88 4d ff        mov    %cl,-0x1(%rbp)

012 <L14>:
 12: 48 8b 40 08     mov    0x8(%rax),%rax
 17: e9 e5 1f 00 00  jmpq   <next way in set>
```

Figure 17: **Instructions used to add noise.** The assembly code is shown using X86 GAS Syntax. `%rax` holds the address of the random predicate used in the `test` instruction at the relative address `0x2`. The moves in the basic blocks `<L13>` and `<L14>` reads the data in the buffer, which uses up the corresponding D-cache set.

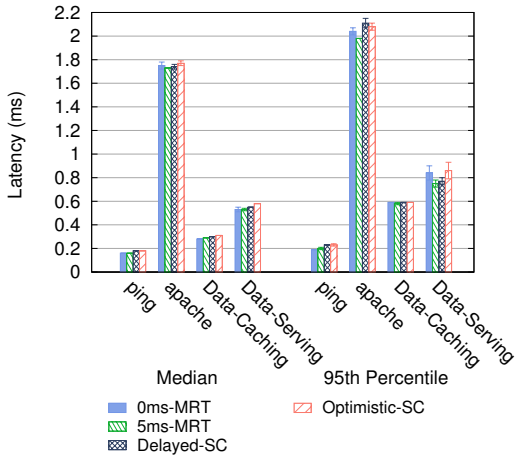


Figure 18: **Median and 95^{th} percentile latency impact of the cleansing scheduler under *worst-case* scenario.** Here all the measured workloads are feed by a client at 500 requests per second. The error bars show the standard deviation across 3 runs.

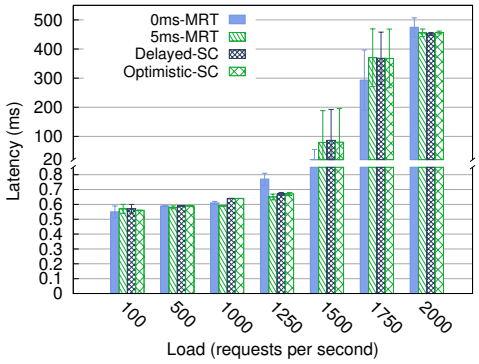
We measured the standalone, steady state execution time of the cleansing routine as 8.4 µs; all overhead beyond that is either due to additional cache misses that the workload experiences or slow down of the execution of the cleansing routine which might itself experience additional cache misses. To measure the overhead of the cleansing scheduler, we pinned two VCPUs of two different VMs to a single PCPU. We measured the performance of one of several latency-sensitive workloads running within one of these VMs, while the other VM ran a competing workload similar to *Chatty-CProbe* (but it did not access memory buffers when awoken). This ensured frequent cross-VM VCPU-switches simulating a worst case scenario for the cleansing scheduler.

We ran this experiment in four settings: no MRT guarantee (0ms-MRT), a 5 ms MRT guarantee (5ms-MRT), a 5 ms MRT with Delayed-SC, and finally a 5 ms MRT with Optimistic-SC. Figure 18 shows the median and 95^{th} percentile latencies under this experiment. The median latency increases between 10–50 µs compared to the 5ms-MRT baseline, while the 95^{th} percentile results are more variable, and show at worst a 100 µs increase in tail latency. For very fast workloads, like *Ping*, this results in a 17% latency increase despite the absolute overhead being small. Most of the overhead comes from reloading data into the cache, as only 1/3rd of the overhead is from executing the cleansing code.

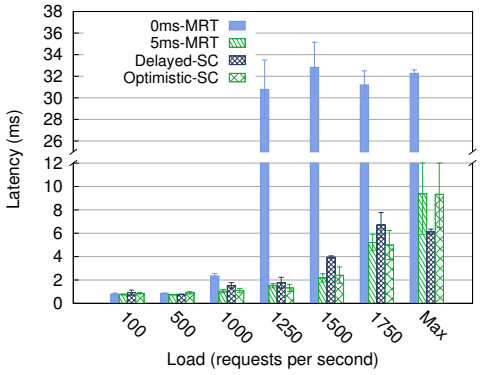
To measure overhead for non-adversarial workloads, we replaced the synthetic worst-case interactive workload with a moderately loaded Apache webserver (at 500 requests per second). The result of this experiment is not shown here as it looks almost identical to Figure 18, suggesting the choice of competing workload has relatively little impact on overheads. In this average-case scenario, we observed an overhead of 20–30 µs across all workloads for the *Delayed-SC* and 10–20 µs for *Optimistic-SC*, which is 10 µs faster. Note that in all the above cases, the cleansing mechanism perform better than the baseline of no MRT guarantee with no cleansing.

To further understand the trade-off between the two variations of state-cleansing, we repeated the first (worst-case) experiment above with varying load on the two latency-sensitive workloads, *Data-Caching* and *Data-Serving*. The 95^{th} percentile and median latencies of these workloads under varying loads are shown in Figure 19 and Figure 20, respectively. The offered load shown on the x-axis is equivalent to the load perceived at the server in all cases except for *Data-Serving* workload whose server throughput saturates at 1870rps (this is denoted as *Max* in the graph).

The results show that the two strategies perform similarly in most situations, with optimization benefiting in a few cases. In particular, we see that the 95% latency for heavier loads on *Data-Serving* (1250, 1500, and 1750) is significantly reduced for Optimistic-SC over Delayed-SC. It turned out that the use of idle-time for cleansing in Optimistic-SC was crucial for *Data-Serving* workload as the tens to hundreds of microsecond overhead of cleansing mechanism under the Delayed-SC scheme was enough to exhaust boost priority at higher loads. From scheduler traces of the runs with *Data-Serving* at 1500rps, we found that the VM running the *Data-Serving* workload spent 1.9s without boost priority under Delayed-SC compared to 0.8s and 1.1s spent under

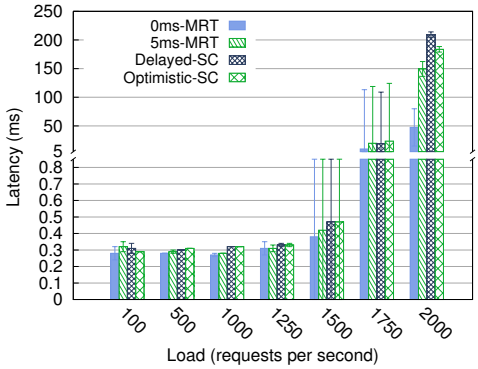


(a) *Data-Caching*

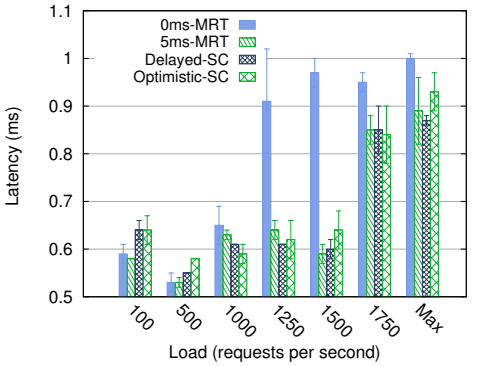


(b) *Data-Serving*

Figure 19: 95^{th} **percentile latency impact of the cleansing scheduler with varying load on the server.** (a) *Data-Caching* and (b) *Data-Serving*. The error bars show the standard deviation across 3 runs.



(a) *Data-Caching*



(b) *Data-Serving*

Figure 20: **Median latency impact of the cleansing scheduler with varying load on the servers.** (a) *Data-Caching* and (b) *Data-Serving*. The error bars show the standard deviation across 3 runs.

5ms-MRT and Optimistic-SC, respectively (over a 120 long second run). The *Data-Serving* VM also experienced 37% fewer wakeups under Delayed-SC relative to 5ms-MRT baseline, implying less interactivity.

We conclude that both strategies provide a high-performance mechanism for selectively cleansing, but that Optimistic-SC handles certain cases slightly better due to taking advantage of idle time.

7 Conclusions

Cloud computing promises improved efficiency, but opens up new threats due to the sharing of hardware across mutually distrustful customers. While virtual machine managers effectively prevent *direct access* to the data of other customers, current hardware platforms inherently leak information when that data is accessed through predictive structures such as caches and branch predictors.

We propose that the first line of defense against these attacks should be the software responsible for determining access: the hypervisor scheduler. For cache-based side channels, we showed that the simple mechanism of MRT guarantees can prevent useful information from being obtained via side-channel attacks that abuse per-core state. This suggests a high performance way of achieving soft isolation, which limits the frequency of potentially dangerous cross-VM interactions.

We also investigate how the classic defense technique of CPU state cleansing can interoperate productively with MRT guarantees. This provides added protection for interactive workloads at low cost, and takes advantage of the fact that the use of MRT makes rescheduling (each of which may require cleansing) rarer.

Finally we note that while the focus of our work was on side-channel attacks, the soft-isolation approach, and the mechanisms we consider in this paper in particular, should also be effective at mitigating other classes of shared-resource attacks. For example, resource-freeing [39] and on-system degradation-of-service attacks.

Acknowledgments

We thank Yinqian Zhang for sharing his thoughts on defenses against side-channel attacks and the code for the IPI-attacker that was used in the ZJRR attack. This work was supported in part by NSF grants 1253870, 1065134, and 1330308 and a generous gift from Microsoft. Swift has a significant financial interest in Microsoft.

References

[1] Graph 500 benchmark 1. `http://www.graph500.org/`.

[2] Heroku PaaS system. `https://www.heroku.com/`.

[3] O. Aciiçmez. Yet another microarchitectural attack:: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[4] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

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

[6] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 103–108. ACM, 2010.

[7] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[8] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of aes. In *Selected Areas in Cryptography*, pages 69–83. Springer, 2005.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[11] G. Dunlap. Xen 4.2: New scheduler parameters. http://blog.xen.org/index.php/2012/04/10/xen-4-2-new-scheduler-parameters-2/.

[12] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. ACM, 2012.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12. ACM, 2012.

[14] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 251–261, May 2001.

[15] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011.

[16] M. W. B. Heinz and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, Feb. 2012.

[17] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[18] HTTPing. Httping client. `http://www.vanheusden.com/httping/`.

[19] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. Cryptology ePrint Archive, Report 2014/435, 2014. `http://eprint.iacr.org/`.

[20] P. A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *2012 IEEE Symposium on Security and Privacy*, pages 52–52. IEEE Computer Society, 1991.

[21] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

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

[23] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397, Aug. 1999.

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

[25] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, pages 393–404. IEEE Computer Society, 2009.

[26] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using stopwatch. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:1–12, 2013.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11. ACM, 2011.

[28] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12. IEEE Computer Society, 2012.

[29] I. Molnar. Linux Kernel Documentation: CFS Scheduler Design. `http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[30] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, 2010.

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

[32] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[33] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *LNCS*, pages 200–210, Sept. 2001.

[34] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[35] 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 *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

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

[37] SPEC. SPECjbb2005 - Industry-standard server-side Java benchmark (J2SE 5.0). Standard Performance Evaluation Corporation, June 2005.

[38] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[39] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[40] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based Defenses against Cross-VM Side-channels, 2014. Full version, available from authors' web pages.

[41] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen (short paper). In T. Ristenpart and C. Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, Oct. 2011.

[42] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[43] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07. ACM, 2007.

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

[45] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.

[46] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[47] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13. ACM, 2013.