TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks

Robert Martin John Demme Simha Sethumadhavan
Computer Architecture and Security Technologies Lab
Department of Computer Science, Columbia University, New York, NY, USA
rdm2128@columbia.edu, jdd@cs.columbia.edu, simha@cs.columbia.edu

Abstract

Over the past two decades, several microarchitectural side channels have been exploited to create sophisticated security attacks. Solutions to this problem have mainly focused on fixing the source of leaks either by limiting the flow of information through the side channel by modifying hardware, or by refactoring vulnerable software to protect sensitive data from leaking. These solutions are reactive and not preventative: while the modifications may protect against a single attack, they do nothing to prevent future side channel attacks that exploit other microarchitectural side channels or exploit the same side channel in a novel way.

In this paper we present a general mitigation strategy that focuses on the infrastructure used to measure side channel leaks rather than the source of leaks, and thus applies to all known and unknown microarchitectural side channel leaks. Our approach is to limit the fidelity of fine grain timekeeping and performance counters, making it difficult for an attacker to distinguish between different microarchitectural events, thus thwarting attacks. We demonstrate the strength of our proposed security modifications, and validate that our changes do not break existing software. Our proposed changes require minor – or in some cases, no – hardware modifications and do not result in any substantial performance degradation, yet offer the most comprehensive protection against microarchitectural side channels to date.

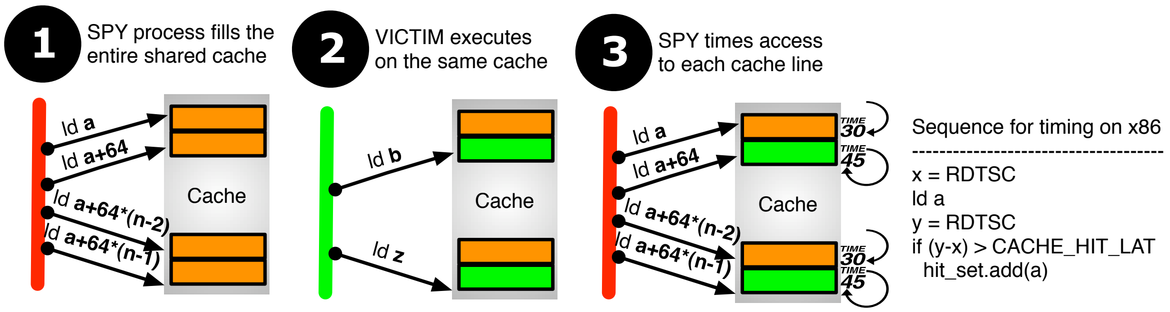
1 Introduction

Any computation has an impact on the environment in which it runs. This impact can be measured through physical effects such as heat or power signatures, or through how the computation consumes system resources such as memory, cache, network or disk footprints. In a side channel attack, an attacker collects these unintentional leakages to compromise the confidentiality of the computation.

A particular class of side channel attacks that rely on microarchitectural leaks has gained notoriety in the last decade. In these attacks, shared on-chip resources like caches or branch predictors have been used to compromise software implementations of cryptographic algorithms [4, 8, 9, 12, 13, 14, 15]. A particularly dangerous attack demonstrated in 2009 revealed that an attacker could record keystrokes typed in a console from another co-resident virtual machine in a cloud setting by measuring cache utilization [17]. But microarchitectural side channel dangers are not limited to cryptographic software or cloud installations: as system-on-chip designs become popular, the tight integration of components may make physical side channels more difficult to exploit. Attackers will likely turn to microarchitectural leaks to learn sensitive information.

In this paper, we present a general mitigation strategy to protect against microarchitectural side channels. Instead of fixing the source of leaks, which requires identification and modification of all on-chip leaky structures, we propose simple changes to the on-chip measurement infrastructure making it difficult to accurately measure leakage.

We explain our solution in the context of a popular microarchitectural side channel attack called the prime-probe attack (Figure 1). Consider two processes, a victim and a spy. The spy process first "primes" all cache sets with its data (represented by red boxes). The victim executes and knocks off some cache lines (green boxes). Following the execution of the victim (or in parallel), in the "probe" phase, the spy process queries each cache line to see if its data items have been knocked out of the cache. If the spy is successful at detecting a cache miss, it knows that the victim must have used that cache line. Then, using additional information about the software algorithms used by the victim or how the OS manages the address space, the spy can deduce the exact datum used by the victim. In this attack and in other microarchitectural attacks, a key step is to determine if a certain microarchitectural event (such as a hit/miss on the primed line, in this case) happened. The spy can determine this information directly using microarchitectural performance event counters or indirectly by measuring the



```
Sequence for timing on x86
------------------------------------
x = RDTSC
ld a
y = RDTSC
if (y-x) > CACHE_HIT_LAT
  hit_set.add(a)
```

Figure 1: A typical microarchitectural side channel attack. A key step is measuring timing (step 3). Our solution is to fuzz timing.

time for the load to return; longer return times indicates a miss in the cache line.

While it is easy to come up with ad hoc fixes for this particular case using software, e.g. flushing cache before process swaps, or hardware changes, e.g. randomizing data placement in the caches, these solutions are reactive to known attacks: they may not generalize to other side channels (which may be unpublished), or even attacks that use the same side channel in a different way. Our solution, in contrast, mitigates all microarchitectural side channel attacks by controlling access to timing and microarchitectural event counters. Without any way to meaningfully measure differences between microarchitectural events, the attacker cannot extract the signal from the noisy data, and microarchitects can design without worry of side channel leaks. Continuing with the example, our solution will provide arbitrary time values to the spy process on each measurement making it difficult to differentiate between hits and misses.

However, it turns out that controlling timing and microarchitectural event counters is nuanced. While most programs are oblivious to on-chip performance counters (from a functional perspective), we find that many system and application programs routinely use timing information for their correct, functional operation. For instance, without a timing source, Linux will not boot, multimedia games will not work correctly, and some cryptographic libraries that use microarchitectural entropy lose strength.

To control timekeeping without breaking the software, we first identify all possible sources of timing information, and then individually discuss how and to what degree each timing source can be fuzzed. We observe that there are only three ways to receive timing information: internal, hardware time sources, such as the time stamp counter on x86; external time sources, such as external interrupts or network packets that deliver time from another computer or device; and through software only clocks that use knowledge of microarchitecture events (*e.g.,* using 1 cycle ADDs in tight loop) to create a virtual clock. This taxonomy is complete because it covers every meaningful combination of internal and external sources, and hardware and software for deducing time. We outline how we handle each of these sources:

- To obscure internal sources, we propose changing the granularity of the internal clock counter. If the minimum resolution of the clock is greater than the longest microarchitectural event on a chip, then it cannot be used to distinguish microarchitectural events. We describe and demonstrate that our algorithm for providing this functionality provides strong security guarantees. It is also trivially implementable in hardware.

- We argue that external events cannot be delivered fast enough to distinguish between microarchitectural events. This is because external events have to cross the chip boundary to get into the chip, and the longest microarchitectural events are also those that cross the boundary.

- We show that software clocks use a very distinct form of inter-thread communication that intentionally races. These races are easily detected using previously proposed race detectors, or our lightweight hardware detection unit that uses existing on-chip infrastructure with minor modifications. Once detected these events can be obscured.

Finally, we recognize that some performance monitoring circumstances will warrant exact timing information and show how obfuscation scheme can be configured to provide such support. This only requires adding one new instruction to the ISA that operates in privileged mode.

We validate our changes by implementing our solution in a virtual machine environment and confirmed that our obscuration schemes does not break software. We show that average performance impact of obscuring clocks is less than 4% (a loose upper bound), and only requires few bits of storage. The combination of these features provide an efficient method for preventing microarchitectural timing attacks.

The rest of the paper is organized as follows: we describe related work in Sec. 2 and the threat model in Sec. 3. We discuss backwards compatibility requirements in Sec. 4. Following that we present our solutions to thwart hardware and software clocks in Sec. 5 and Sec. 6 respectively, and microarchitectural and architectural design in Sec. 7. We provide a security analysis in Sec. 8 and experimental results in Sec. 9. Finally, we discuss how our method for obscuring timing information could be extended to on-chip performance counters in Sec. 10 and conclude in Sec. 11.

2 Related Work

Instruction caches, data caches and branch predictors have all been shown to leak enough information to execute successful attacks on cryptographic systems. These structures have no knowledge of context switches, so when a new process gains control of the CPU it will also inherit the state of the previous process's caches. Although an unprivileged process cannot directly query the state of the caches, it can use clever timing tricks to infer their state, and from that learn sensitive information about the previous process.

Though many microarchitectural side channel attacks [2, 3, 8, 4, 9, 12, 13, 15] differ in their exact implementation and efficacy, most rely on fine grain timing information. The RDTSC instruction, named from an abbreviation of *ReaD TimeStamp Counter*, is an x86 instruction that returns the current value of the timestamp counter (TSC) register [1]. Since the TSC register is incremented every cycle, measurements taken using RDTSC can be accurate to the nanosecond scale. Percival's attack [15] relied on distinguishing between 120-cycle and 170-cycle events. Bernstein's attack [4] relied on distinguishing between timings 45 to 130 cycles apart. The attack of Osvik et al. [13] relied on distinguishing between 65-cycle and 120-cycle events. The branch prediction attack of Acıiçmez et al. [3], in the best case, relied on distinguishing 5250-cycle events from 5500-cycle events.

Numerous countermeasures to microarchitectural side channel attacks have been previously suggested and implemented, though most focus on modifying the implementations of specific cryptographic algorithms rather than fixing the underlying hardware [8, 4, 9, 12, 13, 14, 15]. The downside of this effort is that the countermeasures are not just application specific, but also platform and side channel specific. After a program is modified to prevent one attack, it may be vulnerable to alternative side channels or other sections of code may still be vulnerable. Additionally, relying on programmers to detect and fix side channel vulnerabilities in their code is impossible — not just because programmers are inherently fallible, but also because code may predate the hardware on which it runs.

Previous work has also considered countermeasures that involve hardware modifications to defeat microarchitectural attacks more generally. These fall into two categories: modification to leaky hardware structure, or the disabling of measurement infrastructure.

Fixing Leaky Hardware: Bernstein [4] proposed a new CPU instruction that would allow a process to load specific data into the L1 cache in a constant number of cycles irrespective of hits or misses. Multiple sources [4, 12, 13, 14, 15] considered a fully partitioned cache in SMT or clearing the cache on every context switch, though all admit that the resulting performance degradations would be unacceptably high. Page [14] suggested intentionally randomizing the order of non-dependent loads and stores to confuse spy processes, or inserting random delays into the overall execution time of a cryptographic process, both of which could be done through hardware modifications, though he noted that such a solution increases overall execution time and can be defeated by averaging the results of execution over many runs to average out the perturbations. Percival [15] suggested modifying the eviction policy of SMT to prevent a process from exerting undue control over the L1 cache of its sibling process, though does not go on to describe what might constitute a safe eviction policy short of disabling cache sharing entirely. Wang and Lee [20, 19] proposed two new data cache designs, PLcache and RPcache, and showed how both were effective at preventing specific microarchitectural side channel attacks based on data cache sharing. Their techniques also improved performance but it is unclear how those ideas can be extended to other shared structures.

Disabling Timing Infrastructure Percival also considered limiting or removing access to the RDTSC instruction, but dismissed the idea because he contended that attackers on multi-processor systems could effectively instantiate a virtual timestamp counter, *i.e.,* a software clock (see Section 6). Bangerter et al. [8] suggested completely disabling fine grain timers like RDTSC. They reported that disabling RDTSC would make cache misses impossible to detect without arousing suspicion, though they noted that much existing software requires RDTSC access and thus considered disabling RDTSC to be largely impractical. We provide a method of obscuring RDTSC that allows most software to operate smoothly in Section 5.

3 Threat Model

In this paper, we specifically attempt to mitigate software-based, microarchitectural side-channel attacks.

A microarchitectural attack is one where the attacker exploits the characteristics of the microarchitectural implementation to reveal the victim's secret information. As the attacker shares the processor with the victim, she can make observations on her own microarchitectural state and use those to make inferences about the victim. Alternatively, or simultaneously, she can indirectly modify the microarchitectural state of the victim to slightly affect the victim's execution rate. In all cases, we define microarchitectural attacks to be those that are carried out by exploiting timing

[1] These fine grain timing instructions are not unique to x86: most architectures have an equivalent register to the TSC on x86 such as the TB register on PowerPC and the TICK register on SPARC. Throughout this paper we will use the x86 terminology TSC and RDTSC for simplicity.

differences on the nano- or micro-second timescale.

A software-based attack is one that is executed entirely in software. This means that the attacker can only learn information from machine instructions she executes on the system used for the attack. This excludes, for instance, any power monitoring attacks, where the attacker monitors the power consumption of the system using external probes. It also excludes passive signal monitoring attacks, where the attacker listens to signal leaking from the system such as electromagnetic radiation from the monitor or the sound of typing on the keyboard.

For an attacker to carry out a microarchitectural attack, the attacker needs only user-level access to this system but should be able execute arbitrary user-level code. The attacker also has some means of inducing the victim to run the sensitive code, or else the victim runs the sensitive code often enough that the attacker can simply wait and listen. Finally for a microarchitectural attack both the victim and the attacker share the same processor or a set of processors.

Though we define microarchitectural side-channel attacks to be those that happen on the nano- or micro-second timescale, it is conceivable that in a software attack a set of microarchitectural events could be aggregated to produce a sufficiently large change in program execution time that can be detected with coarse timing information. For instance, an attacker can repeatedly measure few thousands of sensitive unknown latency loads using coarse grained timers to draw probabilistic conclusions about the fraction of loads that hit or miss in the cache. We do not know of such attacks, and it is unclear how effective these attacks can be. Further, such a side-channel leakage blurs the line between "microarchitectural" and "throughput" related side channels, and programs vulnerable to these aggregation attacks fall outside the scope of this paper.

4 Timekeeping Requirements

As this paper recommends changes to the behavior of hardware timekeeping instructions, we must take great care not to disrupt the programs that already use these instructions. In this section we outline some properties of timekeeping infrastructure that any modifications must respect.

Strictly Increasing The RDTSC instruction returns the number of cycles since processor start-up, and is expected to always increase. Any modification to RDTSC behavior must still maintain that RDTSC is a strictly increasing function. Violating this constraint can cause serious problems. For instance, when multicores were first introduced, each core had its own independent TSC register that ran freely on its own without any synchronization. But, when a process would context switch from one core to another while it was making multiple RDTSC calls, it would occasionally see the TSC register as having decreased [1]. To correct this, operating systems now make efforts to synchronize the TSC registers on every core when they first start and the TSC register increments even when the core is asleep.

Entropy The RDTSC instruction is sometimes used to gather entropy by observing the final few bits of the returned value. The entropy gathered can be used for cryptographic purposes or other algorithms that require random bits[2]. The least significant bits of RDTSC represent randomness from OS scheduling, pipeline state, and cache state of the processor. To maintain this functionality, modifications to RDTSC must still enforce that the less significant bits of the return value become unpredictable over time.

Relative Accuracy RDTSC can be used to gather fine-grained performance statistics. Multimedia applications and games use RDTSC to effect a short term delay by spinning until RDTSC reaches a certain value in the near future. For the RDTSC results to make sense in these cases, successive calls to RDTSC must have a difference that accurately reflects the number of cycles between the two calls. We call this the relative accuracy of RDTSC, meaning that a return value is accurate relative to a previous call. Any modification to RDTSC must maintain a degree of relative accuracy. We note that relative accuracy is not a correctness constraint. Software should be resilient to a variety of return values returned by RDTSC, because even without our changes, the RDTSC instruction itself can take a variable number of cycles to execute due to the state of caches, DVFS, scheduling, etc.

Absolute Accuracy The timestamp counter value tells the program for how many cycles the processor has been running, and it is possible that some programs use RDTSC to acquire that information. Though we cannot find explicit examples of this, it is conceivable some program may rely on this property. For instance, a program might use RDTSC to send an email to the system administrator after a computer has been running for more than 10 days. Any timekeeping modifications should enable systems to acquire accurate information when necessary.

5 Securing Hardware Timekeeping

Our goal is to adjust the timekeeping infrastructure sufficiently to thwart side channel measurements but at the same time not break existing software. Towards this we propose that the RDTSC instruction be obfuscated by small randomized offsets.

The extent of the obfuscation will be controlled by the current obfuscation level, denoted as e, which is set by the OS. When e is set to zero, or when in privileged mode, RDTSC instructions will execute with full speed and accuracy. This allows any timing-related OS components like the scheduler to operate as normal, and allows the OS to

[2] OpenSSL 1.0.0, `crypto/rand/rand_nw.c`, ll 157-165

disable the obfuscation on a per-process basis for any programs that are trusted and require high fidelity timing to work properly.

We propose obfuscating RDTSC in two ways: by inserting a real-time delay that stalls RDTSC execution, called the real offset; and by modifying the return value of the instruction by a small amount, called the apparent offset.

To calculate these offsets, we conceptually divide time into *epochs*. Epochs vary in length randomly from 2^{e-1} to $2^e - 1$ cycles. Within any given epoch, we denote the current epoch length as E.

The real offset delays execution of each RDTSC until a random time in the subsequent epoch, and requires that the TSC register always be read on an epoch boundary (Figure 2). When a RDTSC instruction is encountered during execution, its execution will be stalled until the end of the current epoch. On the epoch boundary, the TSC register will be read. The instruction will then continue to stall for a random number of cycles in the range $[0, E)$ of the subsequent epoch. The sum of these two stalls is the real offset, denoted D_R.

In addition to a real offset, we use an apparent offset to fuzz the cycle count returned by RDTSC. After the TSC register has been read at the epoch boundary, a random value in the range $[0, E)$ will be added. This apparent offset is generated independently of the real offset, making RDTSC instructions appear to have executed at a time completely uncorrelated to the time where execution actually resumed.

As a result of these modifications, malicious processes in user-space will no longer be able to make fine grain timing measurements to a granularity smaller than 2^{e-1}, making microarchitectural events undetectable as long as 2^{e-1} is more than the largest difference between on-chip microarchitectural latencies.

Our proposal also maintains all desiderata previously outlined for RDTSC. The obscured RDTSC will be strictly increasing, and relative accuracy will be at most inaccurate by 2^{e+1} cycles. Further, while each RDTSC instruction will be slowed down considerably because of the real delays, the throughput of the system will be largely unaffected except for unusual programs that load the RDTSC frequently. The overwhelming majority of existing programs do not call RDTSC enough to exhibit any slowdown.

5.1 Previously proposed modifications and their flaws

Previous work proposed alternative modifications to RDTSC. We now outline why previous proposals are insufficient.

Disallow any user-space RDTSC instructions Percival [15] and more recently Bangerter et al. [8] both considered disabling RDTSC entirely as a simple and effective way to obscure timing information. As they conclude, however, this change is too disruptive for most systems to tolerate. We solve this by allowing access to the coarse values of RDTSC. Note that our approach is no more dangerous than disabling RDTSC entirely, as an attacker could find other sources for coarse timing information (see VTSC, Section 6).

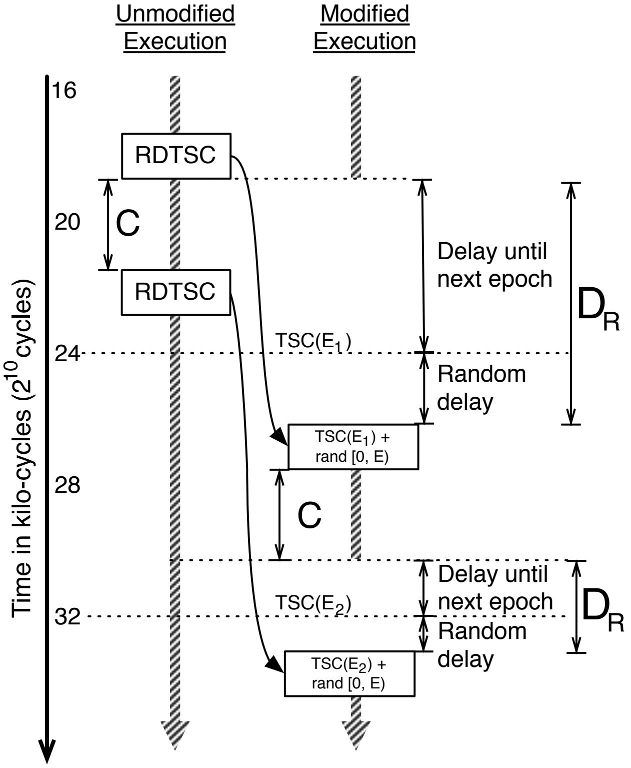


Figure 2: LHS shows normal execution of RDTSC instructions timing code section marked C. We propose adding two delays to RDTSC instructions (RHS). The first delay stalls execution of RDTSC instruction in real-time until the next and the second delays into the next epoch by some random amount. Additionally, the return value of RDTSC is modified to be some random time within the new epoch. This second stop adds randomness to measurement C, defeating statistical reconstruction.

Mask the least significant bits of RDTSC Osvik et al. [13] suggested an obscured RDTSC that masked the least significant bits of RDTSC, creating a step function. They regard such a solution impractical because statistical analysis would defeat it. Vattikonda et al. [18] also noted that such an implementation is not sufficient to prevent fine grain timing by a sufficiently motivated attacker. By calling RDTSC until the epoch bits changed, an attacker would know he was at the beginning of an epoch. He could then execute the code he wished to time, and then continuously call RDTSC and increment a register in a tight loop until the end of the epoch. We solve this problem by both inserting a real offset rather than merely masking the returned result and using variably-sized epochs.

Random offset to RDTSC Jayasinghe et al. [9] suggested adding random offsets to the return value of RDTSC. In isolation, such solutions are impractical. Without any protection mechanism, random offsets may allow for time to appear to go backward: for instance, if RDTSC is called twice in rapid succession, and the random offset for the first call is much larger than the second, the second call will appear to finish earlier. No simple solution to this problem exists.

One possibility is to keep track of the most recently returned value and to disallow a small random offset to be generated immediately after a large one. This repairs RDTSC so that it will only move forward, but allows an attacker to defeat the randomness. By calling RDTSC many times in a row, the attacker will significantly narrow the available range of the offset so that, in the extreme case, it will only be able to return the maximum offset.

If instead the offset is unbounded, repeated calls to RDTSC will result in time appearing to move forward much faster than it actually is. Eventually the offset will be large enough to affect the system. For instance, a legitimate program may call RDTSC and incorrectly calculate the time of day as being hours ahead of its true value.

We therefore conclude that the only way to solve this problem is to prevent a process from calling RDTSC too frequently using a physical delay, as we do in our solution with our real offset.

6 Securing Software Timekeeping

Even with the RDTSC instruction obscured, an attacker on a multicore system can execute a virtual clock to obtain timing information. Previously, Percival [15] recognized this was possible and referred to it as a virtual Time Stamp Counter. In this paper, we abbreviate it as VTSC.

VTSC Formulations Percival imagined a situation where a process has two threads, A and B, each running on separate cores and sharing a memory address X. Thread A initializes a local register to zero, and then runs an infinite loop of incrementing the local register and storing it to X. Assuming no other thread is attempting to read X, this implementation would increment X at a rate of about once per cycle. To use this information, thread B would read X immediately before and directly after the instructions it wishes to time and calculate the difference, mimicking the RDTSC instruction. Thus, Percival's implementation creates write/read shared memory communication, and we refer to his suggestion as a $W \rightarrow R$ VTSC.

But this is not the only way VTSC can be implemented. We present an alternative VTSC implementation that can be implemented with write/write shared memory communication, which we refer to as a $W \rightarrow W$ VTSC. In this implementation, an attacker would use two timer threads T_1 and T_2 and one probe thread P, as well as two shared memory addresses X and Y. T_1 will run in a tight loop of loading X, incrementing it, and storing it back to X. T_2 will do the same for Y. To make a timing measurement, P will set X to 0, perform the timed set of instructions, and then set Y to 0. Afterward, T_1 and T_2 can be terminated and $Y - X$ will reveal the time.

6.1 Countermeasures

Our countermeasure to prevent VTSC timekeeping requires two parts: a detector (a method for detecting VTSC shared memory communications) and a delay producer. The detector must catch all instances $W \rightarrow R$ or $W \rightarrow W$ communications, ideally with as few false positives as possible. The delay producer must, when activated, insert delays that obscure any timing information a VTSC could obtain through such communication.

Several hardware race detectors have been proposed [7, 10, 11, 16] that can detect all VTSC communications with varying degrees of false positives. Of these, the most recent uses existing (Nehalem) performance counters to detect $W \rightarrow R$ communication [7]. While the performance counter trick thwarts $W \rightarrow R$ VTSC, current Nehalem performance counters do not allow us to detect $W \rightarrow W$ communication. If these counters are added in future microarchitectures, the same technique could be used to protect against our $W \rightarrow W$ implementation.

While the performance counter based implementation is fairly simple, one drawback is that most of the events that are caught will be false positives, as many of them can come from shared memory communication that happens far apart in time. However, by construction we know that VTSC needs to quickly share data with another thread, and we care only about very close races that happened within a few tens to hundred cycles. So traditional hardware race detectors or even performance counters that track these dependencies across cache evicts for thousands of cycles are an overkill for this problem. Instead we propose a lightweight method of tracking rapid inter-thread communication to eliminate false positives. Our solution is to keep track of lifetimes of cache accesses and trigger VTSC obfuscation only for recent updates. In the next section we describe this implementation in detail.

7 Implementation

In this section we first describe microarchitectural modifications to implement RDTSC fuzzing and then describe modifications for protecting against VTSC attacks.

RDTSC We propose two methods of implementing the modifications to the RDTSC instruction. The first is to modify the decode stage to translate all RDTSC instructions into four instructions when operating in obfuscation mode.

1. Stall D_{r_1} (until end of epoch)
2. Store TSC $\rightarrow R_1$
3. Stall $D_{r_2} \in [0, E)$
4. Add $D_A \rightarrow R_1$

where $D_R = D_{r_1} + D_{r_2}$ is the real offset and D_A is the apparent offset. Both these delays should be accessible in the decode stage from a true or pseudo random number generator.

The second method is to modify the execution of the RDTSC instruction. The TSC module will, when queried, return a value already fuzzed by the amount D_A. Additionally, the execution stage will not commit the RDTSC instruction for D_R cycles, causing a stall in the instruction pipeline.

VTSC Some hardware modifications may also be necessary to effectively catch all VTSC race events with a minimum number of false positives. We suggest adding two additional bits called TLU (Time of Last Use) to each private L1 cache line that represent how recently the core interacted with the line (see Figure 3). These bits will be initialized to `00` when a new cache line is installed in the cache, and set to `11` when a cache line is used by a thread. Periodically, any non-zero TLU will be decremented by 1. When another core or thread requires read or write access to a cache line and its TLU bits are non-zero, a VTSC violation is reported. This detection method will catch both $W \rightarrow R$ and $W \rightarrow W$ events, while filtering out false positives for old reads/writes that might otherwise cause additional slowdown. The VTSC violation is reported to the OS as an interrupt when inter thread communication occurs.

We show in the results section that this filtering mechanism can potentially filter out 96% of the false positives for decay period of 2^{15} cycles, when compared to a scheme that catches all inter-thread cache sharing. Our implementation is also minimal in terms of area requirements. This is because the filtering scheme cannot be implemented with one bit vectors. To see why consider a one bit implementation. With one bit, if VTSC communication events happen just before and just after flash clearing a single bit they will not be caught. This problem is avoided with two bits that slowly decay with time because we guarantee that there is a minimum separation between between a read and a write, which is basically the time period for the decay operation. On an architecture with 64 sets of 4-way associative cache lines, this modification results in 512 additional bits per core. If the cache lines are 64 bytes each, this amounts to a 0.4% overhead. The conditional logic associated with checking for non-zero TLU can be implemented in parallel with the cache line fetching without significantly impacting clock frequency or latency.

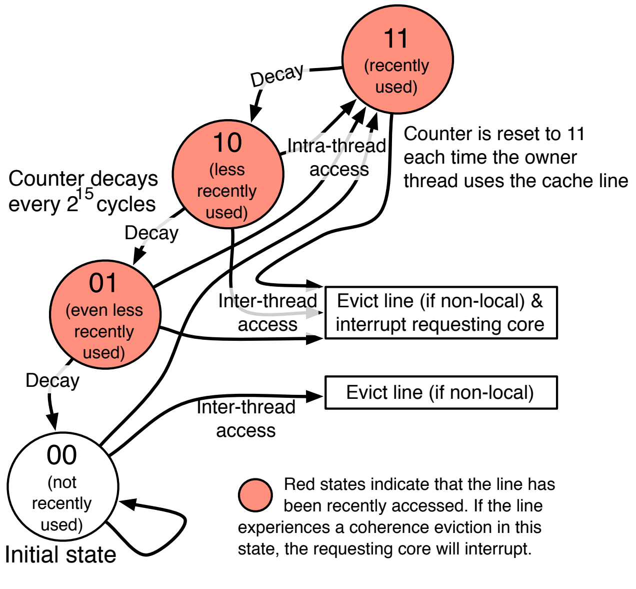

11
(recently used)
Decay
10
(less recently used)
Intra-thread access
Counter is reset to 11 each time the owner thread uses the cache line
Counter decays every 2^{15} cycles
Decay
01
(even less recently used)
Inter-thread access
Evict line (if non-local) & interrupt requesting core
Decay
Evict line (if non-local)
Inter-thread access
00
(not recently used)
Initial state
Red states indicate that the line has been recently accessed. If the line experiences a coherence eviction in this state, the requesting core will interrupt.

Figure 3: The TLU bits for each line maintain a state machine which indicate whether or not a line has been recently accessed. This information is used to cause interrupts during coherence evictions, thus disrupting tight W→R and W→W communications.

The delay producer must cause a delay that obscures the timing information associated with the race condition. Experimentally, we found that allowing the operating system to handle an empty interrupt was enough to obscure the timing information. Our results are presented in Section 9.

We note that our proposed VTSC modifications may cause noticeable slowdowns in programs that have tight inter-thread locking mechanisms. Fortunately, such locking mechanisms are usually found in system programs, where our modifications do not apply. This is because attackers with privileged access do not need side channels to steal information. For locks in highly contended user mode programs, performance pathologies may result, but overall many locks in production applications appear to be acquired without contention [6].

7.1 ISA Modifications

We recommend a new instruction to control the amount of obfuscation that we call TSCWF, short for "TimeStamp Counter Warp Factor". TSCWF will take one parameter, which will be an integer between 0 and 15 (0 disables obfuscation). This parameter, previously referred to as e, sets the epoch length to be in $[2^{e-1}, 2^e - 1]$. For instance, TSCWF 10 will set the epoch size to be between 512 and 1023 cycles. When the CPU resets, the TSCWF value will be initialized to 0 so that operating systems unaware of the

TSCWF instruction will still function properly. Additionally the CPU needs to identify to the software if it supports the TSCWF instruction. To prevent processes from simply granting themselves full RDTSC access, the TSCWF instruction must be a protected instruction. The TSCWF instruction will only affect calls to RDTSC when the CPU is in user-mode. It will not affect the accuracy of kernel-mode calls to RDTSC.

7.2 Deployment on Existing Hardware

Our modifications for obscuring RDTSC could be deployed on existing hardware using virtualization. The CPU would trap upon encountering a RDTSC instruction in user-mode. The virtualization software would return a RDTSC value with the appropriate apparent offset, and also spin for enough cycles to effect the necessary real offset.

However, such a deployment would only be secure on a single-threaded virtual machine. Attackers on a multi-threaded virtual machine could use software clocks for timing information, and existing hardware does not provide enough infrastructure to prevent all such communication.

7.3 Software Management

Despite our precautions, some programs may exist that cease to function when RDTSC is obscured. Moreover, while other programs may continue to function, they may suffer noticeably from the inaccuracy of RDTSC (e.g., performance measurements).

To solve this problem, the operating system can provide functionality to track which threads have been granted RDTSC permissions from the system administrator and raise or lower the core's RDTSC permissions immediately before and after context switches. In Linux, for example, a bit could be added to the task structure that indicates whether or not that process should be obscured. For all programs that are obscured, the scheduler can decide on an appropriate value for e. A value of 0 indicates that RDTSCs will function normally. As the kernel is trusted, we never obscure a call to RDTSC in kernel mode.

8 Security Analysis

In this section we demonstrate why our proposed changes are sufficient to prevent an attacker from learning about the presence microarchitectural events. We also show the resistance of our changes to statistical analysis.

8.1 Completeness of Approach

All programs can be classified into two categories: non-deterministic or deterministic programs. The attacker's program cannot be deterministic because it would produce the same output irrespective of the microarchitectural state, which is directly counter to the attacker's objective in a microarchitectural side channel attack. Therefore, a microarchitectural attack requires non-determinism to succeed.

The attacker's program does not have root access, and therefore must exist within the operating system abstraction of a process or set of processes. In this abstraction, a process can only undergo non-deterministic behavior in three ways: non-deterministic hardware instructions; accessing shared memory; and system calls. We discuss the feasibility of extracting microarchitectural state from each of these sources.

Hardware Instructions Most instructions executed by the CPU are deterministic, such as adds, multiplies, jumps, etc. Trivially, any serial execution of deterministic instructions is itself deterministic. However, instructions that read hardware processor state are non-deterministic, as their input values do not determine their outputs. We must explicitly examine each non-deterministic instruction on an architecture to verify that it cannot depend on or leak microarchitectural information.

If the output of an instruction does not change with repeated execution, the instruction cannot be used in a microarchitectural attack. An example of this is the CPUID instruction on x86.

Performance counters are non-deterministic and are a viable way to learn about microarchitectural events – in fact, some performance counters are designed to reveal the occurrance of such events. However, most ISAs only allow privileged programs to query performance counters, and therefore currently do not pose a threat. The time stamp counter, however, is an exception: it is available to all programs. We therefore consider fine grain timing instructions like RDTSC to be one of the feasible ways that a process can create the non-determinism necessary to detect a microarchitectural event.

Shared Memory For the program to detect a microarchitectural event using shared memory, there must be a load or store whose outcome depends on that event. As loads and stores are themselves deterministic in the value that they read or write, it is only by the interleaving of these instructions that we can sense the event.

Without loss of generality, let us consider two threads of execution that share memory and conspire to detect a microarchitectural event. The interleaved instructions must be a load and a store, or a store and a store – two loads cannot discover their own interleaving. Furthermore the interleaving of the two instructions must depend upon a microarchitectural event, meaning that they must happen at very nearly the same time. From these observations, we can conclude that rapid inter-thread communication through shared memory using either load/store or store/store interleavings is the only viable way for a microarchitectural event to be detected.

System Calls System calls introduce non-determinism into a process, as their return values cannot always be determined by their inputs. Barring a scenario where the operating system exposes performance counters via system calls, a process would need to gain timing information through the system call sufficiently accurate to detect microarchitectural events.

In practice, the overhead of a system call is far too great to allow such timing information to propagate to the process. Even with newer instructions that allow low-overhead system call entry and exit, a system call usually takes hundreds or thousands of cycles purely in overhead. With such an unpredictable and lengthy timing overhead, system calls cannot provide useful timing information. Even if we consider that the system call returns with data from a highly accurate clock (such as a network read from another computer using RDTSC), the accuracy of the timing information will be degraded too greatly to be useful. Thus, we do not take any further steps to obscure timing from system calls.

Improving system call speed is an ongoing field of research. If system calls eventually become fast enough to detect microarchitectural events, additional steps may be needed to obscure this timing information source.

In sum, these arguments show that depriving a process of fine grain ISA timers like RDTSC and disrupting the timing of inter-thread communication is sufficient to deter microarchitectural side channel attacks. Since we have shown methods of protecting against RDTSC misuse in Section 5 and concurrency races in Section 6, our solution is complete.

8.2 Resilience Against Statistical Analysis

Previous work [15, 13] expressed concern that any RDTSC fuzzing could be easily defeated by statistical analysis. We demonstrate that defeating our proposal using statistical analysis would be exceedingly difficult for fine grain timing measurements.

Consider the most general case of using RDTSC to detect the presence or absence of a microarchitectural event for some contiguous sequence of instructions. To make timing measurements on this sequence, an attacker will have to execute RDTSC once before the section and once afterward. We will denote these two calls RDTSC$_1$ and RDTSC$_2$, which will yield the return values R_1 and R_2. The attacker's aim is to distinguish between two possibilities, one of which is slower and one which is faster. We denote the faster possibility as taking T_F cycles and the slower as taking T_S cycles, where T_F and T_S are very close but not equal. We call the difference between them $T_\triangle$ so that the following equation holds:

$$T_S = T_F + T_\triangle \tag{1}$$

Figure 4: Table of variables used in analysis.

R_1, R_2	The return values of RDTSC$_1$ and RDTSC$_2$.
E	The length of an epoch, in cycles.
T_F, T_S	The number of cycles the attacker expects to measure in the fast, slow case.
n	Defined as T_F/E, discarding any fractional remainder.
$T_\triangle$	Defined as $T_S - T_F$, where $T_\triangle \ll E$.
T_0	Defined as $T_F \mod E$. Note $T_0 \in [0, E)$.
N_F, N_S	The attacker's measurement results, in epochs.

According to our RDTSC obfuscation scheme, executing RDTSC$_1$ will delay the attacker until an unknown offset into the subsequent epoch. We call this physical offset D, where $D \in [0, E)$. The attacker cannot do anything to influence D. She cannot also directly measure D, as her return value will contain an unrelated apparent offset.

The attacker will then execute the instructions, which takes T_F cycles to complete in the "fast" case. In the "slow" case, it takes an additional $T_\triangle$ cycles.

Finally, the attacker will call RDTSC$_2$, which will delay the instruction until a random time in the subsequent epoch then return R_2. From the value R_2, the attacker can only determine how many integer epochs elapsed since R_1. In other words, the attacker can only know $\left\lfloor \frac{D+T_i}{E} \right\rfloor$, where T_i is either T_F or T_S.

In the "fast" case the attacker's offset into his epoch will be $(D + T_F) \mod E$. We denote the offset $T_F \mod E$ as T_0 such that $T_F = nE + T_0$ for some integer n, so that the attacker's offset can be re-written as $n + \left\lfloor \frac{D+T_0}{E} \right\rfloor$. We denote the attacker's offset as N_F. The attacker's measurement of N_F will always be either n or $n + 1$. Assuming D is evenly distributed, the probability of each measurement is:

$$\Pr(N_F = n) = \frac{T_0}{E} \tag{2}$$

$$\Pr(N_F = n + 1) = 1 - \frac{T_0}{E} \tag{3}$$

Now let us consider the "slow" case, where the timed code section takes $T_F + T_\triangle$ cycles. Using the same logic as above, the offset of the attacker immediately before RDTSC$_2$ will be $D + T_0 + T_\triangle$, and thus the attacker's only new information from the measurement will be N_S where $N_S = \left\lfloor \frac{D+T_S}{E} \right\rfloor$. The probability of all possible measurements is:

$$\Pr(N_S = n) = \max\left(1 - \frac{T_0 + T_\triangle}{E}, 0\right) \tag{4}$$

$$\Pr(N_S = n + 1) = 1 - \left|\frac{T_0 + T_\triangle}{E} - 1\right| \tag{5}$$

$$\Pr(N_S = n + 2) = \max\left(0, \frac{T_0 + T_\triangle}{E} - 1\right) \tag{6}$$

Recalling $T_\Delta \ll E$, we see that the probabilities for the attacker to measure the outcome n or $n+1$ are nearly identical, whether we are in the "fast" or "slow" case. We also see that the probability of measuring $n+2$ in the slow case is either very small or zero, depending on the value of T_0. In the fast case it is always zero. Thus, $n+2$ is a unique result that cannot be measured if the timed section is fast.

If the timed section is slow, the attacker will only have a zero probability of measuring $n+2$ unless T_0 is in the range $[E - T_\Delta, E)$. In this range, the attacker will only succeed if the random real offset is also in $[E - T_\Delta, E)$. This means that the attacker must make $\Theta\left(\left(\frac{E}{T_\Delta}\right)^2\right)$ measurements to observe an $n+2$ event. Therefore, the running time cost for an attacker to learn a bit of secret information is increased by this ratio as well. For example, if $\frac{E}{T_\Delta} = 1000$, then an attack that previously took 5 minutes will instead take on average about 10 years.

9 Experimental Results

9.1 Correctness Validation

To test our assertions about program durability when RDTSC behavior is altered, we used a virtual machine to emulate our proposed RDTSC modifications. We modified KVM, an open source virtual machine, to trap and emulate every RDTSC instruction. In the trap handler, we implemented our delay function and our fuzzing scheme. We used an epoch length of $2^{13} = 8192$ cycles. Using hardware virtualization we were able to run more cycles than we otherwise would using a full simulator, and we were better able to test responsiveness. However, we observed that the physical delay we introduced into KVM caused some RDTSC instructions to be emulated out of order. While the overall delay remains the same, the exact placement of the delay within the program execution may be vary. This may temper the efficacy of the protection, but should still allow us to correctly evaluate the performance effects.

We were able to install and boot Ubuntu in a 4-core virtual machine, open Firefox, browse the web, and use Flash without any noticeable slowdown or unexpected process behavior. We were also able to install and boot Windows XP in a 2-core virtual machine, open Internet Explorer 8, and browse the web. Again, all behavior, including multimedia applications such as Flash, had no noticeable slowdown compared with a system without RDTSC obscured.

To test our ability to obscure VTSC, we wrote an attacker program capable of using VTSC to distinguish between an L1 cache hit and a main memory access. On our machine, these corresponded to measurements of 75 cycles and 224 cycles, respectively (including delay from serializing instructions).

We configured the performance monitoring unit on Nehalem to trigger an interrupt on every $W \rightarrow R$ event, and modified the Linux kernel to handle these interrupts. This approach is very similar to the approach presented by Greathouse *et al.* [7]. We found that the delay of the interrupt alone caused the VTSC to become inoperable, giving indistinguishable results for L1 hits and misses.

9.2 Performance Effects

We ran the PARSEC benchmark suite to determine the performance cost of such an implementation that relies on existing hardware. We found most benchmarks exhibited a slowdown of less than 1.2%, while two outliers, dedup and streamcluster, experienced slowdowns of 14% and 29%, respectively. The geometric average slowdown was 4%. This slowdown should be considered a very conservative upper bound. As shown in Figure 6, the vast majority of sharing events that we captured happened well outside of a ten-thousand cycle threshold, meaning that our proposed VTSC detector would not trigger on these events.

Due to the lack of available performance counter infrastructure, we were not able to measure the negative performance effects from $W \rightarrow W$. While we expect such events to be even more rare than $W \rightarrow R$ events, we note that additional performance degradation may occur as a result.

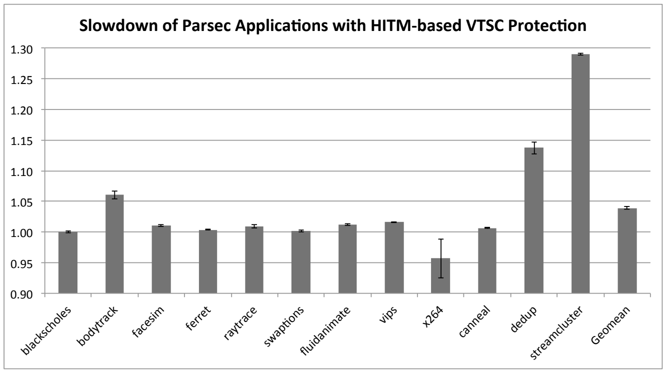


Figure 5: **The slowdown of Parsec Applications with our $W \rightarrow R$ interrupt in place. Most applications show little or no performance degradation.**

10 Applying Similar Modifications to Other Performance Counters

Modern processors have built-in counters of microarchitectural events that could potentially be used to create much more sophisticated side channel attacks than exist presently. For instance, a spy program with access to a branch-prediction-miss counter would be able to circumvent the inaccuracy associated with making timing measurements and guessing whether a given interval represents a hit or miss. This translates to faster and more powerful attacks by malicious spy programs. Traditionally these performance counters have been available only in kernel-

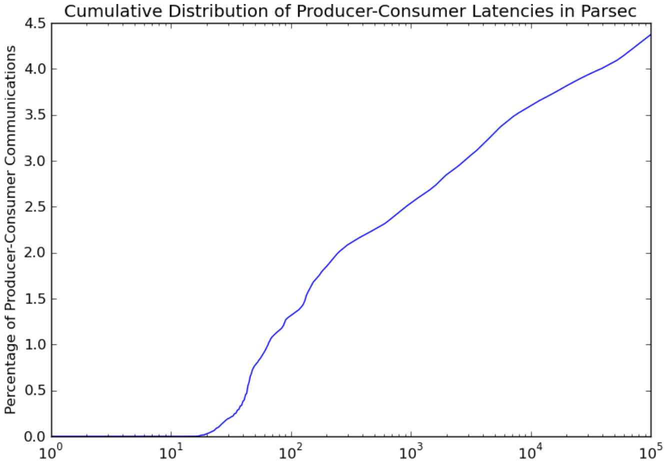


Figure 6: **A cumulative distribution function showing the communication latency in Parsec Applications, measured in uncore cycles. Less than 4% of $W \rightarrow R$ interactions happen faster than 10,000 cycles, meaning that the TLU bit implementation we suggest would filter a majority of false positives – thus further improving performance.**

space, but recent work [6] has suggested exposing them to user-space for fine grain performance measurements. The proposed modification allows a thread to only monitor its own microarchitectural events. Nevertheless, this information could still be used by an attacker to infer the microarchitectural events of another process and thus represents a possible side channel.

We recommend that such performance counters be obscured in a similar way to the timekeeping counters, for similar reasons. Such modifications would be easier to implement and enforce because little to no existing code relies on the accuracy of these performance counters. Counters that can represent data-dependent information, such as cache misses or branch misses, are particularly important to obscure. To implement our proposed algorithm for a new performance counter, the important task would be to determine an appropriate epoch length. We leave it to future work to more precisely determine an appropriate epoch length for all performance counters as they become available to user programs.

11 Conclusion

Microarchitectural side channel attacks are a growing threat due increased sharing promoted by the move to cloud based systems. In these systems multiple users are often multiplexed onto the same hardware. This hardware is usually not hardened against leaks allowing attackers to gain secret information with little effort. Recognizing the dangers of side channels, researchers have proposed countermeasures to plug some of the leaking structures such as the L1 data cache. While these countermeasures serve their intended purposes they do not and cannot claim to protect against undiscovered or unknown side channels, or even all known side channels (*e.g.,* branch predictor side channel). In this paper we propose a comprehensive approach to mitigate all microarchitectural side channels.

A key observation we make is that all microarchitectural side channel attacks require a high fidelity timing source. Our solution is to prevent the attacker from accessing this source. Specifically we obscure the on-chip performance counters that are used for timekeeping, and also fuzz the software methods that are used to emulate hardware timekeeping. The degree of security provided by this method is roughly proportional to the square of the degree of fuzzing, and is configurable.

We take great care to ensure that our changes are "backwards compatible" with existing system and application software. Our modifications were emulated on virtual machine. In terms of hardware modifications, we suggest adding one new instruction to the ISA to configure the level of fuzzing, and add a very small amount of storage (order of few hundred bits for a 32KB cache). For single-threaded virtual machines, our proposal can be implemented without hardware modifications.

Our modifications pave the way for systems that are secure against microarchitectural side channels. By providing an environment in which microarchitectural events cannot be detected, we allow hardware developers to design highly efficient shared microarchitectural structures and policies that would, under current standards, be considered to leak unacceptable amounts of information.

12 Acknowledgments

We would like to thank Joseph Greathouse for his help with the Intel HITM counter event and anonymous reviewers and the members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback.

This work was supported by grants FA 99500910389 (AFOSR), FA 865011C7190 (DARPA), FA 87501020253 (DARPA), CCF/TC 1054844 (NSF) and gifts from Microsoft Research, WindRiver Corp, Xilinx and Synopsys Inc. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities.

Appendix: Timewarp SVF Measurements

Since the publication of this paper (published at ISCA'12), we have measured the side-channel vulnerability factor for Timewarp. To review, SVF measures the leakage of execution patterns through a system under specific assumptions about the attacker and victim. Our simulator

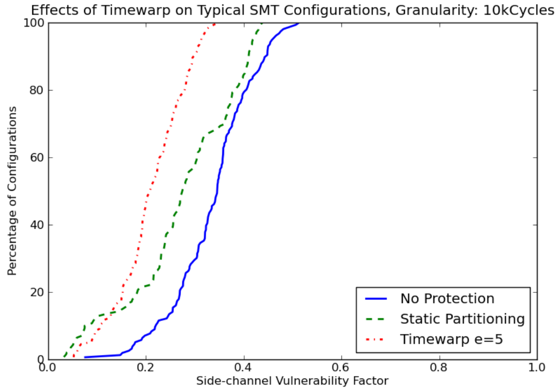


Figure 7: SVF measurements of Timewarp systems at the 10K victim instruction granularity. We see than a relatively small Warp-factor of 5 ($e = 5$ or up to 32 cycles) somewhat obscures the side channel. At $e = 13$, however, the attacker is not even able to complete a scan.

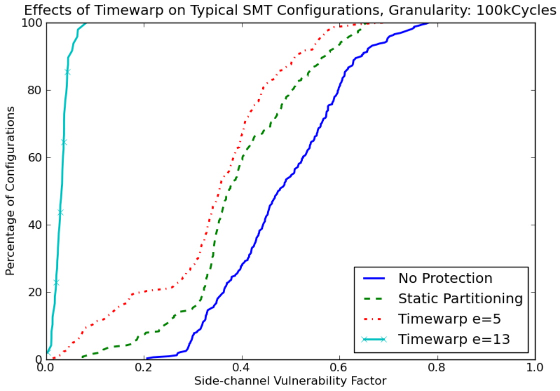


Figure 8: SVF measurements of Timewarp systems at the 100K victim instruction granularity. Again we see than $e = 5$ obscures the side channel. At this granularity the attacker is able to complete cache scans, so we see data for $e = 13$. Although some leakage still occurs, the side channel is heavily obscured.

measures the leakage of patterns in address load streams through a variety of microarchitectures to a prime-and-probe style software attacker. To implement Timewarp in the simulator, we simply add pipeline stalls and fuzzing to simulated RDTSC instructions – the first implementation approach outlined in Sec. 7. For this section, we run the same set of simulations as detailed in the original SVF paper [5], though we only use the "in order" attackers on SMT-enabled systems and add Timewarp.

The data from these new simulations can be found in Figures 7 and 8. We enable Timewarp for two warp factors, $e = 5$ and $e = 13$, creating epochs up to 32 and 8,192 cycles (respectively) in length. For comparison purposes, we also look at configurations with caches statically partitioned. As in the SVF paper [5], we present results in the form of a cumulative histogram, allowing us to quickly look at many configurations[3]. In Figure 7 we examine SMT systems at a fine granularity of 10,000 victim instructions. We see that a relatively low Timewarp factor of 5 obscures the side channel significantly and that there is no line for a factor of 13. This is because the attacker is sufficiently slowed down and unable to complete any cache scans in less than 10,000 victim instructions.

To examine the affect of Timewarp with higher warp factors, we also show data for a granularity of 100,000 victim instructions in Figure 8. Again, here we see that a factor of only 5 obscures the side channel for a large number of configurations. However, at a factor of 13 the side channel is very nearly entirely closed. Although some leakage still occurs, these data indicate that Timewarp is very effective for closing microarchitectural side channels.

References

[1] Possible rdtsc bug - intel®software network. WWW page, 2009. http://software.intel.com/en-us/forums/showthread.php?t=65593.

[2] O. Acıiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *14th ACM Conference on Computer and Communications Security (ACM CCS'07) Computer Security Architecture Workshop*, 2007.

[3] O. Acıiçmez, C. K. Koc, and J.-P. Seifert. On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006.

[4] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2004. http://cr.yp.to/papers.html/#cachetiming.

[5] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, New York, NY, USA, 2012. ACM.

[6] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. *SIGARCH Comput. Archit. News*, 39:353–364, June 2011.

[7] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. *SIGARCH Comput. Archit. News*, 39:165–176, June 2011.

[8] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy (SP)*, pages 490–505, May 2011.

[9] D. Jayasinghe, J. Fernando, R. Herath, and R. Ragel. Remote cache timing attack on advanced encryption standard and countermeasures. In *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*, pages 177 –182, dec. 2010.

[3] We used a newer version of the simulator for this study, changing several minor microarchitectural features and the way in which the attacker is simulated. Thus these results are not directly comparable to charts from the original paper.

[10] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. *SIGPLAN Not.*, 26:235–244, April 1991.

[11] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. *SIGARCH Comput. Archit. News*, 37:337–348, June 2009.

[12] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on aes. In *Proceedings of the 13th international conference on Selected areas in cryptography*, SAC'06, pages 147–162, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.

[14] D. Page. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280, 2005.

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

[16] M. Prvulovic and J. Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 110–121, New York, NY, USA, 2003. ACM.

[17] 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*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[18] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 41–46, New York, NY, USA, 2011. ACM.

[19] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83 –93, nov. 2008.

[20] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35:494–505, June 2007.