On-Demand Time Blurring to Support Side-Channel Defense

Weijie Liu[1], Debin Gao[2], and Michael K. Reiter[3]

[1] Computer School, Wuhan University, China
[2] Singapore Management University, Singapore
[3] University of North Carolina, Chapel Hill, NC, USA

Abstract. Side-channel attacks are a serious threat to multi-tenant public clouds. Past work showed how secret information in one virtual machine (VM) can be leaked to another, co-resident VM using timing side channels. Recent defenses against timing side channels focus on reducing the degree of resource sharing. However, such defenses necessarily limit the flexibility with which resources are shared. In this paper, we propose a technique that dynamically adjusts the granularity of platform time sources, to interfere with timing side-channel attacks. Our proposed technique supposes an interface by which a VM can request the temporary coarsening of platform time sources as seen by all VMs on the platform, which the hypervisor can effect since it virtualizes accesses to those timers. We show that the VM-Function (VMFUNC) mechanism provides a low-overhead such interface, thereby enabling applications to adjust timer granularity with minimal overhead. We present a proof-of-concept implementation using a Xen hypervisor running Linux-based VMs on a cloud server using commodity Intel processors and supporting adjustment of the timestamp-counter (TSC) granularity. We evaluate our implementation and show that our scheme mitigates timing side-channel attacks, while introducing negligible performance penalties.

1 Introduction

Computers that are simultaneously shared by multiple tenants introduce the risk of information leakage between tenants via *access-driven side-channel attacks*. In these attacks, one tenant (the attacker) infers information about another tenant (the victim) by measuring victim's impact on the resources that they share. A predominant form of such side-channel attacks is *timing* attacks, in which the attacker measures victim-induced perturbations in the time between events that the attacker can observe. For example, a widely studied form of access-driven timing attack is one in which the attacker observes which of the attacker's lines the victim evicted from a processor cache when it ran, which the attacker can observe by timing the durations of its own memory fetches when it runs after the victim [11]. Researchers have demonstrated that these attacks can be used to steal a victim VM's cryptographic keys over Xen [42,40,17], to collect potentially sensitive application data and hijack accounts on a victim web-server in a PaaS cloud [43,10], and others [13,4,3,9].

Many defenses against access-driven timing side channels focus on reducing the degree or granularity of resource sharing (e.g., [2,41,44]). Such defenses, however, necessarily limit the flexibility with which resources are shared, typically with costs to performance, efficiency, and/or utilization. For this reason, in this paper we advance a second class of defense, namely interfering with the adversary's ability to time events. Even though there are theoretically many ways for an adversary to time events [21], timing events with sufficiently fine granularity to extract cryptographic keys, for example, typically leverages the *time stamp counter* (TSC) on x86 platforms (e.g., [40,23]). As such, decreasing the fidelity of the TSC has been advocated in previous research as a method to interfere with the most potent of these attacks (e.g., [33,25]).

The important insight that we contribute in this paper is that the degradation of TSC fidelity need not be constant over time, but can be tuned according to the operations that a tenant application is performing. For example, since AES operations are much faster than RSA operations, the degradation needed to hide timing side channels during AES operations might be much smaller than what would be needed to hide (potentially much larger) timing signals during RSA operations.

In this paper we leverage this insight to advocate for a simple interface by which a tenant application can adjust the fidelity of the TSC observed by tenants on the platform to a specified level, i.e., specified as a number of low-order bits of the TSC to zero before reporting it to a tenant. Our prototype virtualizes the TSC and, at any point in time, zeros a number of bits requested by a tenant on the machine. Moreover, we show that VM-Function (VMFUNC) techniques offer a low-overhead mechanism to VMs to adjust TSC fidelity. Because of the efficiency of these VMFUNC techniques, VMs have the flexibility to adjust TSC fidelity frequently, without incurring significant performance impacts from doing so. We show, for example, that these fidelity adjustments are sufficiently quick to allow fine-grained adjustments to TSC fidelity with virtually no performance impact to, e.g., encryption libraries and web and file servers employing encryption. Moreover, we quantify the degree of TSC degradation that is needed to overcome simple but powerful covert-channel attacks, in which we reveal different settings of the TSC degradation that are "just enough" to disable a last level cache and a memory bus contention attack.

Allowing tenant applications to adjust the fidelity of the TSC could potentially open the platform to detrimental effects caused by tenants overzealously degrading the TSC fidelity for other tenants. We conduct experiments to show that VMs are not particularly sensitive to TSC degradation, but there is obviously a point at which this is no longer true. While our goal here is not to measure the potential for abuse, we note that it is straightforward to impose limits on the amount or duration of degradation that is allowed, e.g., as a cap on the number of bits that will be dropped or a per-tenant budget on the time during which it can request that the TSC fidelity be kept at a reduced level. In the limit, a tenant in a public cloud could be charged extra, in proportion to the time and amount of degradation that it requests, and likewise, tenants could be

partially reimbursed for durations in which their executions were subjected to lower TSC fidelity at the request of co-located tenants.

The remainder of the paper is organized as follows. In Section 2 we establish our requirements. In Section 3 we outline our design and provide in-depth details about our prototype implementations. Section 4 describes the experiments performed using recent, state-of-the-art cross-VM side channels also the performance evaluation of the prototype. Section 5 describes related work and finally Section 6 provides brief concluding remarks and discusses limitations of our technique.

2 Threat Model and Requirements

We focus on infrastructure-as-a-service (IaaS) clouds that potentially execute multiple virtual machines from customers on the same, shared computer hardware. Examples of such cloud providers include Amazon EC2 and Rackspace. Due to the nature of shared computer hardware on this architecture, access-driven side-channel attacks become possible among virtual machines (VMs) executing simultaneously, which might originate from different customers.

Threat model. Our threat model is an attacker VM that tries to infer information about a victim VM running simultaneously on the same computer hardware. Specifically, the attacks that we consider in this paper are those that perform side-channel timing attacks by measuring victim-VM-induced perturbations in the time between certain events on the shared hardware that the attacker VM can observe. We assume that the infrastructure provides reliable access control to prevent the attacker VM from accessing the victim VM directly or via forms of privilege escalation. To this end, we assume that the cloud's virtualization software is trusted.

Under such a threat model, we focus our attention on defense mechanisms that reduce the fidelity of the time stamp counter (TSC) to interfere with the attacker's ability to time events. We consider the following important requirements of a flexible and effective defense.

"Just-enough" masking. We want to differentiate various side-channel timing attacks that measure the time between events at different granularities. For example, since AES operations are much faster than RSA operations, their corresponding successful attacks need to obtain timing information at finer granularity. Therefore, a victim might demand that fine-grained timing information be hidden while it performs AES operations and more coarse-grained timing information be hidden during its RSA operations.

On-demand protection. While a potential victim has sensitive information in the VM that requires protection and the victim VM knows precisely when sensitive operations involving that data happen, we assume a powerful attacker who also has precise information of such operations (what they are and when they happen) and the corresponding events that the attacker VM can observe in an attempt to infer the sensitive information. That said, we also assume that

such operations constitute a small percentage of the entire workload of the victim VM, and so the victim would prefer an "on-demand" protection to minimize the performance overhead. Specifically, we aim for a solution where the victim can dynamically change the protection at low cost, e.g., allowing the victim to enable protection when encrypting blocks of an `https` response (a sensitive operation since the cryptographic key is involved) while turning off the protection when sending out encrypted blocks (a non-sensitive networking operation).

Timing information available to VMs. Besides the security features cloud customers desire when running virtual machines on a cloud, they may also demand timing information from the cloud platform for their general computing purposes. For example, a VM running a web server may need timestamp information at microsecond precision for logging purposes. Therefore, a solution to defend against the side-channel timing attacks should not have noticeable impact on such uses of timing information.

3 Design and Implementation

As discussed in Section 1, attackers typically need to leverage the time stamp counter (TSC) to obtain fine-grained timing information for the purpose of inferring sensitive information from a victim VM. Therefore, decreasing the fidelity of TSC has been proposed as a potential defense against side-channel timing attacks [33,25]. However, existing such approaches do not satisfy our requirements on just-enough masking and on-demand protection.

Our technique allows a tenant to dynamically adjust the fidelity of the TSC observed by tenants on a virtualized platform to a specified level, i.e., to request a number n of low-order bits of the TSC be zeroed before reporting it to tenants. The request n can be determined by the nature of the sensitive operation that the victim VM is to perform and known exploits available to attacker VMs, so that only *just enough* bits are removed to disable these attacks. After performing its operation, the victim VM can retract its request to coarsen the TSC (i.e., by requesting that $n = 0$ bits be removed).

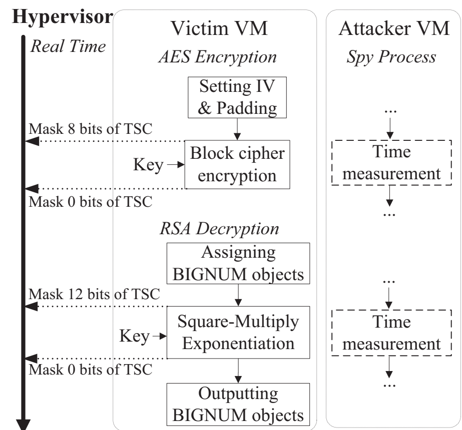


Fig. 1. Overview with hypothetical victim program

Figure 1 shows an overview of our design with a hypothetical victim program that performs on-demand requests to reduce the fidelity of TSC. The victim VM in our example performs

both AES and RSA operations, while the attacker VM performs time measurements to exercise a side-channel attack. To defend against such attacks, the victim VM sends on-demand requests to the hypervisor to degrade the fidelity of the TSC during only key-dependent operations that are vulnerable to timing side-channel attacks. Moreover, the degree of degradation is set so that "just enough" low-order bits of TSC are masked for the specific cryptographic operation ($n = 8$ bits for AES and $n = 12$ bits for RSA in Figure 1).

Although it is relatively simple to enable TSC emulation (by setting the `RDTSC_EXITING` bit in VMCS) and to zero n bits from TSC readings, two implementation issues require further elaboration. The first is how to set n when multiple threads from multiple VMs request different values of n at overlapping times. In this case, the value of n used should be the maximum among those requested (up to a limit). That is, we take a conservative approach, zeroing the maximum number of bits for which any threads on any guest VMs has a request in effect, which implies that the (lesser) protection requested by other threads/VMs is additionally enforced, *a fortiori*.[4]

The second issue warranting further comment is how to achieve *on-demand* protection without inducing substantial overhead. Specifically, we need to allow a guest VM to dynamically request changing the value of n with little cost. It could make these requests with a VMCALL/hypercall, but doing so causes the guest VM to encounter a VM-exit and so is expected to result in prohibitive overhead when a VM changes n frequently.

Here, we propose leveraging VM-Function (VMFUNC), a feature of the Intel micro-architecture instruction set, to reduce the overhead. VMFUNC allows VMs to use hypervisor functionality without a VM-Exit. We performed a simple experiment to compare the overhead of a VMCALL/hypercall interface and a VMFUNC call interface with empty implementation, and find that they cost 1,622 and 160 CPU cycles, respectively, on our i7-4790 CPU. The VMFUNC interface therefore promises substantial cost savings if it can be used.

Although VMFUNC is designed to be general purpose with up to 64 different functions [1], current processors have implemented only one of them, specifically to enable a VM to switch its Extended Page Table (EPT). A VM specifies the EPT pointer by putting the corresponding index into the `ecx` register, and then executing the VMFUNC instruction either from user mode or kernel mode. Execution then traps into the hypervisor without any VM-exit, which switches the EPT pointer to that specified by `ecx`, and subsequently returns to the VM without a VM-enter.

Although the VMFUNC instruction is attractive as a general interface to use hypervisor functionality (in our case to request the hypervisor to change the value of n) with low overhead, no new function besides EPT switching can be

[4] To adjust n to the second highest value when the most demanding thread/VM has finished its sensitive operation, each VM kernel should track all masking requests from its threads and the hypervisor should track all masking requests from VMs. Our evaluation prototype supports this tracking at the VM kernel only, owing to our inability to add or modify VMFUNC instructions, as discussed below.

added without changes to the processor hardware. To evaluate the corresponding performance overhead should processor hardware support our functionality using VMFUNC, we commandeered the EPT switching mechanism in our evaluation prototype, reserving a few specific settings of the EPT pointer (`ecx` register value when calling VMFUNC) for the purpose of our on-demand request to change the value of n. That is, when the `ecx` register contains one of these specific values (0 to k in our prototype), we use it to set the value of n for TSC fidelity reduction. This design is not viable in practice, since it disrupts EPT switching and supports TSC degradation requests from only a single VM at a time. However, it suffices to estimate the overheads associated with dynamically adjusting TSC fidelity via the VMFUNC interface, should it be extended to support our mechanism.

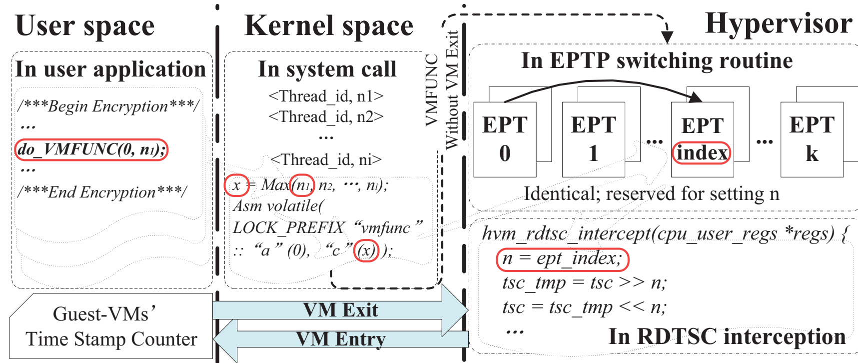


Fig. 2. Design of our evaluation prototype; `do_VMFUNC` is a system call that tracks requests from VM threads and communicates the VM's current maximum masking request to the hypervisor via VMFUNC invocations

Figure 2 shows the design of our evaluation prototype. We first create a few identical EPT tables denoted EPT 0 to EPT k. A user-space process makes a customized system call to request masking n bits of the TSC. Such a system call is processed by the VM kernel, which keeps track on different requests from different threads and executes a VMFUNC instruction with `ecx` $\in [0, k]$. In the hypervisor, switching the EPT table thus has no effect on guest execution (due to identical EPT tables). When any `rdtsc` instruction is executed, the hypervisor reconstructs the setting of n by calculating the offset between the current EPT pointer from EPT base address (effectively reconstructing the value of `ecx` in the most recent VMFUNC call) and masks the low-order n bits of TSC value. With this, we create an efficient communication channel from VMs to the hypervisor for on-demand requests to modify n.

4 Evaluation

In this section, we evaluate our proposal. We present two representative covert-channel timing attacks and show how n can be set accordingly to stop them. Finally, we focus on a number of realistic workloads on virtual machines to

illustrate the impact of setting n at various values and modifying n at different granularities.

4.1 Defending against timing attacks

In this subsection, we evaluate our prototype's effectiveness in defending against real timing attacks. Specifically, we want to see how n should be configured dynamically to provide "just-enough" protection to mask out precise time measurement by the attacker VMs to disable the attacks.

We choose two representative covert-channel timing attacks, the Last Level Cache (LLC) attack and the memory bus contention attack, which are practical on modern computers. We specifically choose these two attacks because they typically require time measurement at different granularities, and we would like to see how our proposed method can be configured to defend against them with different values of n. Moreover, if we can show that our technique defends against these *covert* channels, in which the sender cooperates with the receiver to communicate information, then this provides strong assurance that it will also defend against similar *side* channels, in which the sender does not knowingly cooperate with the receiver.

Cross-VM covert-channel attacks. We briefly outline the two attacks here. We target cross-VM covert-channel attacks in a general setting where a sender process and a receiver process running on different VMs on the same physical host are trying to communicate via a covert timing channel.

In the case of the Last Level Cache (LLC) attack [15], the receiver process 1) fills one or more cache sets with its own code or data; 2) waits for the sender process to utilize the same cache set(s); and 3) measures the time to load his code or data again. This follows a typical PRIME+PROBE technique [26] where the sender process sends a bit of information by utilizing (or not utilizing) the same cache sets, which results in different amount of time taken in the receiver's last PROBE operation. We follow the LLC attack by Liu et al. [23] and implement it on two HVM guests running on an Acer Veriton M4630 machine running Xen. We used an eviction set consisting of cache lines from all four cache slices on our i7-4790 CPU to conservatively prime and probe the cache sets.

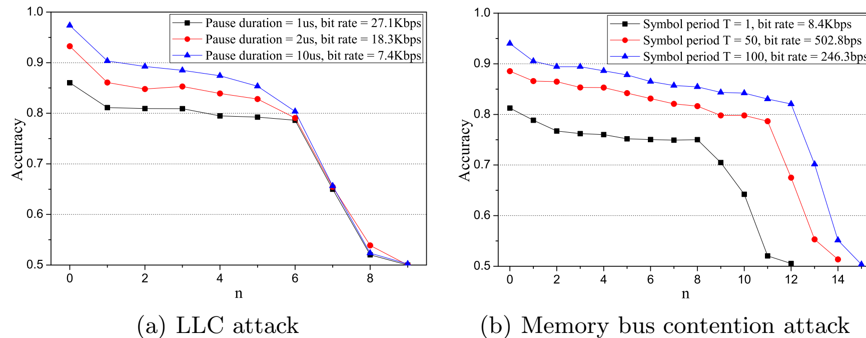
The memory bus contention attack works in a similar way, while the time measurement is typically more coarse-grained compared to the LLC-based attacks. The sender selectively performs an exotic atomic memory operation that triggers a bus locking behavior [39], which causes longer access time by the receiver and therefore effectively creates the covert channel. We configure the receiver to use the latest Streaming SIMD Extensions (SSE) instructions to access the memory bus bypassing the cache lines to reduce noise from the cache lines (which could mask out the bus locking effect).

In both attacks, we do not implement channel error correction or other accuracy improvements leveraged in previous work, except the necessary encoding mechanism to handle VM scheduling and to provide transmission synchronization. We do this to uncover the impact of reducing fidelity of TSC on the bit-by-bit accuracy of the covert channel.

Impact of degrading TSC fidelity on timing attacks. Our intention here is to first execute the LLC and memory bus contention attacks with some realistic parameters on an unmodified Xen system and observe the corresponding accuracy of information received at the receiver process. We then turn on our protection and modify the value of n to observe the corresponding impact on the accuracy observed. We also want to enable some coarse comparison between the two different types of covert-channel attacks to demonstrate our scheme's flexibility in handling different types of threats.

To obtain some realistic attack settings, we configure various sending bit rates for both attacks. Since they use different encoding schemes [23,39] (LLC attack uses the RZ encoding while memory bus contention attack uses the Manchester encoding), the sending bit rates are configured indirectly as follows. For the LLC attack, we configure three different pause durations (waiting time between two consecutive bits) of 1 μs, 2 μs and 10 μs, which result in sending bit rates between 27 Kbps and 7 Kbps. For the memory bus contention attack, we set the symbol period T (number of consecutive exotic atomic operations to be repeated in sending out each bit of information) to be 1, 50, and 100, which result in sending bit rates between 8 Kbps and 246 bps.

Under the various sending bit rates, we measure the accuracy at the receiving process. We consider a powerful attacker who has access to our machine to perform experiments to find out the best threshold — a timing threshold used by the receiving process to infer whether the sender had performed the operation (loading data into the cache set for LLC attack or executing the exotic atomic memory operation for the memory bus contention attack) to signal a 1 or 0 on the covert channel. We further assume that the attacker can perform re-calibration of the threshold when we remove different number of bits from the the TSC readings (n). Figure 3 shows the results of our experiments.



(a) LLC attack (b) Memory bus contention attack

Fig. 3. Impact on mitigating covert-channel timing attacks

Figure 3 clearly shows that accuracy of the covert-channel attacks decreases when n increases. What is interesting is that the accuracy experiences a much

steeper drop when n is greater than a certain number, and the accuracy quickly approaches 50% (the same accuracy as random guessing).

Comparing the two types of attacks, we also notice that our scheme can obtain the same protection (in terms of lowering the accuracy) with smaller n for the LLC attack. This is due to the finer-grained time measurement needed in LLC attacks since the LLC operates much faster than physical memory. This observation reinforces our motivation of this paper — different victim programs and different attacks require different degree of degradation of the TSC fidelity, and the need of dynamically setting the value of n.

4.2 Performance evaluation

Having shown the effectiveness of our scheme in defending against covert-channel timing attacks, we now focus on the overhead evaluation. More specifically, we want to see the impact in terms of performance overhead when a potential victim VM dynamically sends on-demand requests to the hypervisor to change the value of n. With this, we hope to shed light on the recommended usage of our protection mechanism in striking a balance between timing-attack protection and performance overhead. We consider three different VM workloads in our evaluation that differ in the percentage of instructions that require protection against timing attacks.

Our experiments were carried out on a Dell XPS 8700-R39N8 desktop machine with Xen virtual machine monitor installed. Table 1 shows the hardware and software configurations of our prototype. In some experiments, we have a client-server setting where the server is running as a VM on this machine, and the client is running on a host OS on another physical machine with the same hardware configuration and Ubuntu 16.04 LTS.

Table 1. Experiment platform

Server Model:	Dell XPS 8700-R39N8
Processor:	Intel Core i7-4790 3.6GHz
Memory:	16GB
VMM:	Xen 4.6.0
Guest OS:	Linux kernel 3.14.60
vCPUs per VM:	1
Memory per VM:	2048MB

One subtlety we have in the performance overhead evaluation is to measure the precise timing overhead when TSC fidelity has been degraded. Our protection mechanism makes it impossible for any VMs to carry out precise time measurement, but we (for the purpose of doing performance overhead evaluation) still want to be able to obtain the finest-grained TSC readings. For this purpose, we introduce a customized hypercall in Xen which always returns the precise TSC readings when it is called. We use this customized hypercall for measuring the overhead experienced by a VM in all experiments in the rest of this section.

WL-1 – Encryption. In this part of the evaluation on performance overhead, we no longer focus on timing attacks but legitimate workloads on victim VMs. The objective is to find out how much performance overhead such victim VMs experience when they request timing-attack protection. The first workloads (WL-1) we consider here include two encryption workloads, one where the victim VM

performs AES encryption and the other where the victim VM performs RSA encryption, both using implementations that are vulnerable to cross-VM timing attacks [14,17]. In both cases, the victim VM tries to protect its keys from being compromised via co-residency timing attacks by inserting VMFUNC instructions into its crypto library (`libcrypto.so` in OpenSSL 1.0.2g) to modify n.

When instrumenting OpenSSL source code to insert VMFUNC instructions, we focus on protecting *key-dependent* components. Figure 4 shows an overview of such instrumentation. For AES, we insert VMFUNC instructions before the first round (to turn on TSC masking) and after the last round in each block encryption (to turn off TSC masking), which translates to 32 pairs of VMFUNC instructions executed for each 1 KB text encrypted. For RSA, we insert a pair of VMFUNC instructions in the function `bn_mod_mul_montgomery()`, which performs Montgomery modular multiplications and is usually the target of timing attacks. A noticeable difference between our instrumentation on these two workloads is that our protection (code between a pair of VMFUNC instructions) covers about 90% of AES runtime but only 30% − 40% of RSA runtime.

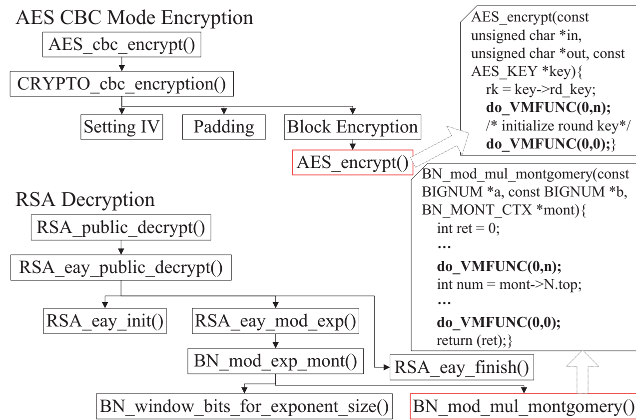


Fig. 4. Instrumenting AES and RSA crypto libraries

Figure 5 shows the normalized runtime overhead with the baseline being an uninstrumented OpenSSL on a VM running on unmodified Xen. Here we try to compare two different strategies of providing timing attack protection: S1 with a constant degradation of TSC fidelity throughout the VM's lifetime (with 8, 16, and 24 bits removed) where no communication is needed between the victim VM and the hypervisor, and S2 with on-demand protection where VMFUNC instructions signal the hypervisor to adjust TSC fidelity. Intuitively, S2 supports on-demand and "just-enough" protection, but experiences additional overhead due to the VMFUNC instructions, especially when they are invoked frequently in the crypto operations.

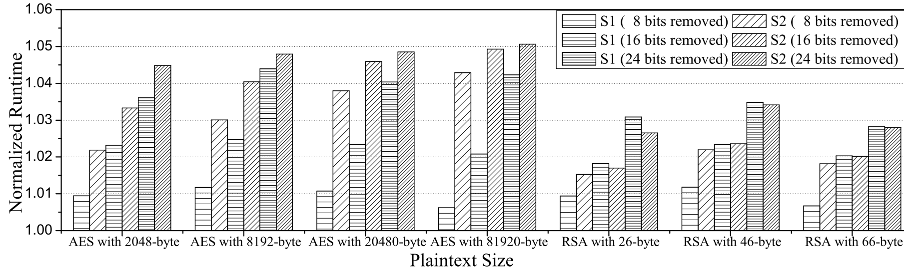


Fig. 5. Runtime overhead in AES/RSA workload

Surprisingly, our results show that the difference between S1 and S2 is not only small, but sometimes S2 actually outperforms S1, e.g., for RSA when 16 or 24 bits are masked. This is likely due to the smaller portion of key-dependent instructions in RSA (30% to 40%) compared to that in AES (about 90%). When key-dependent instructions constitute a small percentage, the overhead due to unnecessary protection of key-independent instructions in S1 could outweigh that due to VMFUNC instructions, which results in S2 experiencing lower overall overhead. This result shows that the additional benefit of our scheme in providing on-demand and "just-enough" security does not necessarily come with higher overhead. In other words, we could obtain better security and lower overhead at the same time in certain VM workloads.

WL-2 – HTTPS web server with PHP. In WL-1, we saw a case where S2 (our proposed on-demand and "just-enough" protection via frequent VMFUNC instructions from victim VM to hypervisor) may outperform S1 (constant protection throughout VM's lifetime without VM-hypervisor communication) in certain scenarios where key-dependent instructions constitute a relatively small percentage of all instructions executed in the VM program. Here, we focus on another realistic workload where the percentage of key-dependent instructions is even smaller — WL-2 where the VM runs an HTTPS web server with PHP. In this case, the workload mainly consists of encryption (part of which is key-dependent as shown in WL-1), web services (key-independent), and networking (key-independent).

The experiment is on an isolated 1 Gbps LAN. The victim VM runs Apache webserver with HTTPS and PHP, and a client running on another host (not in a VM) executes a webserver load tester Siege (`https://www.joedog.org/siege-home/`). Siege is configured to send continuous HTTPS requests (with no delay between two consecutive ones) to the webserver for 10 minutes. One concurrent request is deployed for measuring the response time, while a maximum number of concurrent requests (that result in a maximum throughput) are used as the input for measuring the maximum network I/O performance.

The Apache webserver is configured to use RSA_WITH_AES_256_CBC_SHA for key exchange/agreement and AES256 CBC for data encryption, both of which are key-dependent and require timing protection. We have four different configu-

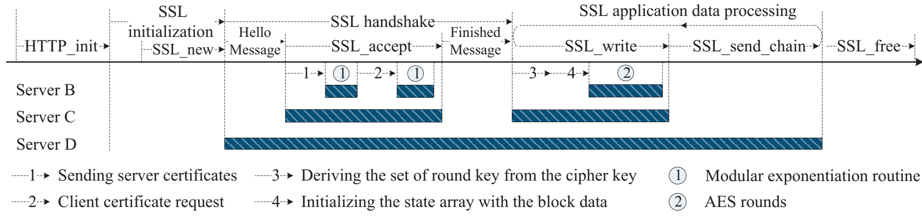


Fig. 6. Instrumenting HTTPS webserver at different granularity

rations of the Apache webserver. Server A is the unmodified version without any VMFUNC instrumentation, while Server B, C, and D are instrumented to insert VMFUNC instructions to protect key-dependent instructions at decreasing granularity; please see Figure 6.

Server B has the finest-grained instrumentation with VMFUNC pairs covering the cryptographic algorithms as described in WL-1. Server C and D have fewer VMFUNC instructions inserted with protection covering some other key-independent instructions of the webserver. Table 2 shows the number of VMFUNC instructions executed in preparing and sending off an HTTPS response. As expected, finer-grained instrumentation results in more VMFUNC instructions called, e.g., in Server B.

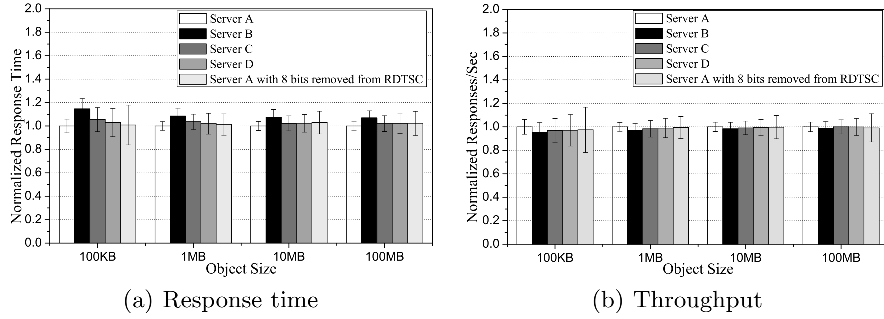
Table 2. Number of VMFUNC instructions executed in an HTTPS response

File Size	Server A	Server B	Server C	Server D
100KB	0	6576	142	2
1MB	0	65596	1402	2
10MB	0	655520	13986	2
100MB	0	6554006	139824	2

Figure 7 shows the normalized response times and normalized throughput for WL-2 for five different configurations: Server A without timing protection (the baseline of normalized response time and throughput), Server B, C, D, and Server A with constant timing protection. We only show results when removing 8 low-order bits of TSC readings here. We performed the same experiment when removing 12 bits, and the results were very similar with slightly bigger variance (due to a more coarse-grained timer).

We first make a comparison among the five different configurations of the webserver. Server A without any timing protection (the baseline) obviously gives the best throughput result. What is interesting, though, is that Server C (an instance of S2) very consistently outperforms Sever A with constant timing protection (S1), and the advantage is more pronounced for bigger file sizes. The reason is similar to that in WL-1 — the savings on unnecessary timing protection on key-independent instructions outweigh the cost of additional VMFUNC instructions called. Server B has the smallest throughput mainly because of its most fine-grained VMFUNC instrumentation.

WL-3 – SCP server. Both WL-1 and WL-2 show instances where our on-demand and "just-enough" timing protection (S2) outperforms constant timing protection (S1) in specific settings. We now turn to a third workload in which



(a) Response time (b) Throughput

Fig. 7. Evaluation of WL-2

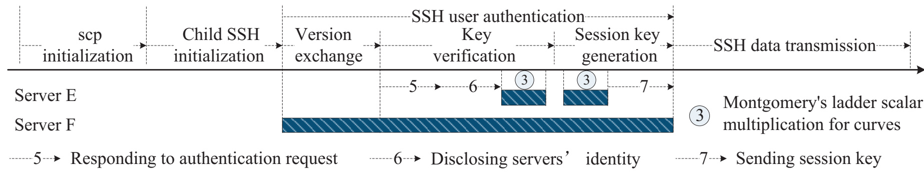


Fig. 8. Instrumenting SCP server at different granularity

the key-dependent operations account for a very small percentage of the entire workload — WL-3, of a secure file copying (SCP) server. In WL-3, only the very initial part of the transaction involves the authentication key which is to be protected from timing attacks (ECDSA computation in OPENSSL that is vulnerable to a timing attack [16,8]), while the remaining part of the transaction does not need to be protected.

Instrumentation of the SCP server results in Server E and F as shown in Figure 8, with Server E having finer-grained instrumentation and more VMFUNC instructions (12 VMFUNC instructions executed in Sever E compared to 2 in Server F for an SCP transmission).

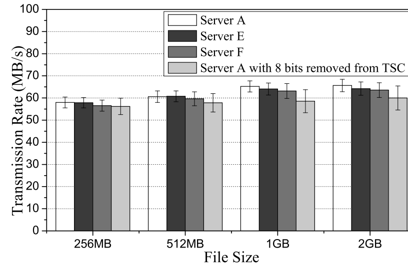
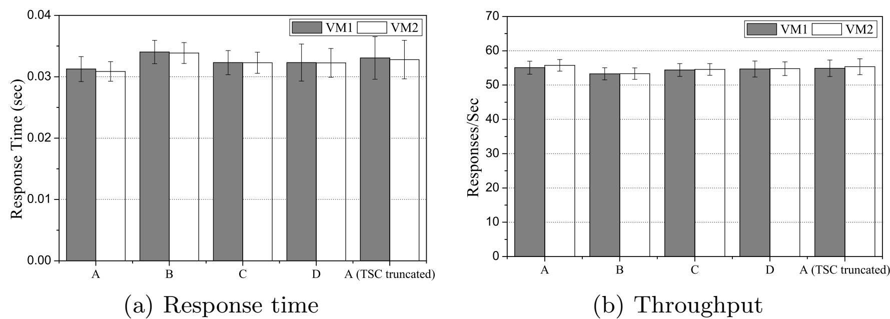


Fig. 9. Network performance in WL-3

Transmission rates of the servers are shown in Figure 9 (showing only results with 8 low-order bits of TSC masked since those with 12 bits masked are very similar). We, again, observe that S2 (Server E and F) outperforms S1 (Server A with timing protection throughout the VM lifetime), except that this time the difference is even more pronounced than that in WL-2, and both instrumentations in S2 outperform that in S1. With these three different workloads and the consistent results obtained, we clearly show that in many realistic workloads, the benefits of the frequent communication from a VM to the hypervisor (which saves on unnecessary timing

protection) could outweigh the overhead of the VMFUNC instructions. Therefore, our proposed mechanism of on-demand and "just-enough" masking of TSC readings not only provides better security, but also in many cases results in lower overhead.

Overhead on VMs that do not need timing protection. We now turn to the last part of our performance overhead evaluation, where we measure the impact of one VM demanding n bits of TSC masking on other co-resident "victim" VMs. These other VMs are not under any timing attacks but simply experience some performance overhead due to a co-resident VM that demands higher security.



(a) Response time (b) Throughput

Fig. 10. Results of being co-resident with a security-demanding VM

Figure 10 shows the response time and throughput of individual VMs when VM1 requests $n = 8$ low-order bits of the TSC be masked, while VM2 makes no such requests. Both VMs are running the Apache webservers and connected to a Siege client as in WL-2. These results show that the performance overhead on VM2 is minimal. Sever C and its co-resident innocent VM experiences the smallest overhead, which is consistent with the results obtained in WL-2.

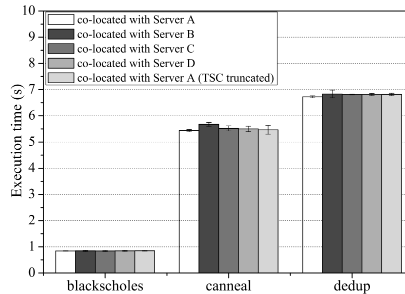


Fig. 11. Overheads of PARSEC benchmarks when co-located with a VM requesting TSC masking

Our last experiment includes a VM running a PARSEC benchmark while co-resident with another VM running Server A, B, C, or D with $n = 8$ bits of the TSC masked. Figure 11 shows that, in general, the performance impact is minimal. That said, `canneal` seems to be more affected by the TSC masking. We believe that it is due to `canneal` being most memory intensive, and the large amount of VMFUNC instructions (in the case of Server B) potentially leads to more cache-misses on the other VM.

5 Related work

Many existing works have proposed defenses against co-residence-based timing side-channel attacks. Adding noise to time sources was one of the first timing-attack mitigation schemes [12], and this has been applied specifically to the time stamp counter on Intel platforms [33,25], as discussed previously. Our scheme also belongs to this category; our contribution lies in showing how dynamically coarsening the time stamp counter can provide on-demand protection for very little cost. Other approaches seek to ensure that measurable intervals of executions are independent of secret values, by causing all such measurements to return the same, secret-independent result [2,41] or by aggregating timing events among multiple VM replicas [21]. These approaches come with greater costs, however.

Apart from general-purpose mitigations of timing side-channel attacks such as these, there are also defense mechanisms targeting specific side-channel attacks, i.e., by interfering with resource sharing so as to eliminate information conveyed by the events that the attacker times in its attack. For example, resource partitioning (e.g., [27,30,7]) and access randomization (e.g., [35,36,18]) have been proposed as hardware defenses against timing side channels in CPU caches. Other approaches have modified how caches are used to mitigate side channels in them (e.g., [19,44]). Some have coarsened CPU sharing by altering the CPU scheduler to mitigate timing side channels specifically in per-core caches (e.g., [31,32]). Wang et al. [34] analyzed timing channels through shared memory controllers and proposed techniques to close them. CATalyst uses an Intel-specific cache optimization [22] to fight against LLC-based side channels.

Yet another way of defending against side-channel attacks is to modify the applications to better protect their secrets. These solutions range from tools to limit branching on sensitive data (e.g., [6]) to application-specific side-channel-free implementations (e.g., [20]), or even to execute multiple program paths — as if the program were executed using many secret values [28]. Our proposal also requires changes to the application, although the changes are straightforward, involving inserting some VMFUNC instructions right before and after operations involving secrets.

While we have focused in this paper on demonstrating on-demand defense via adaptive coarsening of the time stamp counter, this technique could be equally well applied to other real-time clocks on the platform. Still, it remains possible for an application to implement its own timer thread that it can use to measure other events in the system [37]. For example, Shwarz et al. [29] and Chen et al. [5] used this technique to build a timer inside an SGX enclave (and thus without access to a real-time clock), for the purposes of mounting and defending against classes of side-channel attacks, respectively. Adapting the techniques we describe here to provide on-demand coarsening of such "clocks" is an intriguing area of future work.

We are not the first to use VMFUNC in the implementation of security techniques. SeCage [24] retrofits VMFUNC and nested paging in Intel processors to transparently provide different memory views for different compartments at a low cost, preventing the disclosure of private keys and memory scanning from

rootkits. We, on the other hand, simply leverage VMFUNC as a low-cost interface between the VM and the hypervisor to make frequent communications between them efficient.

6 Conclusion and Limitations

In conclusion, we propose a method to allow VMs to dynamically request that the time stamp counter (TSC) be coarsened temporarily (to a level requested by the VM) on the platform, to mitigate timing side channels that use it. We take advantage of hardware virtualization extensions to provide a lightweight yet effective method to enable system-wide side-channel mitigation. By leveraging the VMFUNC interface in a novel way, our technique allows a VM application to send on-demand requests to the hypervisor to mask just enough low-order bits of the TSC to disable precise time measurements by another co-resident VM. We demonstrated the efficacy of our defense against two covert channels, thereby shedding light on how many TSC bits should be zeroed in these attack scenarios. Our experiments with three different workloads showed that our proposal could have lower performance overhead than existing defenses that provide constant degradation of TSC fidelity throughout the VM's lifetime.

Our design does have a few limitations, however. First, our design depends on hardware support, specifically for invoking a new function via the VMFUNC instruction. Second, any defense that coarsens timing sources (ours or others [33,25,38]) might affect time-critical operations. Third, our approach relies on application developers to locate sensitive portions of the code that are vulnerable to timing side-channel attacks. Fourth, an attacker VM might request large values of n simply to degrade others' use of the TSC. As discussed in Section 1, we believe that policies could be put in place to discourage such activities. Lastly, allowing applications to vary n could itself potentially lead to a side-channel leakage by revealing *when* a sensitive operation is occurring. Because attackers can often infer these occurrences based on other circumstances, or even cause them to occur (e.g., by submitting requests to the victim VM), we consider this risk to be minimal.

Acknowledgment This work was supported in part by NSF grant 1330599.

References

1. Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.
2. A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *17th ACM Conf. Comp. & Comm. Sec.*, 2010.
3. A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies*, 2015.

4. N. Benger, J. Van De Pol, N. P. Smart, and Y. Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *16th Intern. Workshop on Cryptographic Hardware and Embedded Systems*, 2014.
5. S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *12th ACM Asia Conf. Comp. & Comm. Sec.*, 2017.
6. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *ISOC Network and Distributed System Security Symp.*, 2015.
7. L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Architecture and Code Optimization*, 8(4), 2012.
8. D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *23rd ACM Conf. Comp. & Comm. Sec.*, 2016.
9. B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *ISOC Network and Distributed System Security Symp.*, 2017.
10. D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symp.*, 2015.
11. D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *32nd IEEE Symp. Security and Privacy*, 2011.
12. W.-M. Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4), 1992.
13. R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symp. Security and Privacy*, 2013.
14. M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. IACR Cryptology ePrint Archive, Report 2015/898, 2015.
15. G. Irazoqui, T. Eisenbarth, and B. Sunar. A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *36th IEEE Symp. Security and Privacy*, 2015.
16. G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Fine grain cross-VM attacks on Xen and VMware. 2014.
17. G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, cross-VM attack on AES. In *17th Intern. Symp. Research in Attacks, Intrusions, and Defenses*, 2014.
18. G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3), 2008.
19. T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symp.*, 2012.
20. R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *RSA Conf., Cryptographers' Track*. Springer, 2008.
21. P. Li, D. Gao, and M. K. Reiter. StopWatch: A cloud architecture for timing channel mitigation. *ACM Trans. Information and System Security*, 17(2), 2014.
22. F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *22nd Intern. Symp. High Performance Comp. Arch.*, 2016.

23. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symp. Security and Privacy*, 2015.
24. Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *22nd ACM Conf. Comp. & Comm. Sec.*, 2015.
25. R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Intern. Symp. Comp. Arch.*, 2012.
26. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conf., Cryptographers' Track*, 2006.
27. H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *1st ACM Cloud Computing Security Workshop*, 2009.
28. A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symp.*, 2015.
29. M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. arXiv:1702.08719, 2017.
30. J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *41st IEEE/IFIP International Conf. Dependable Systems and Networks*, 2011.
31. D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *18th European Symp. Research in Computer Security*, 2013.
32. V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symp.*, 2014.
33. B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in xen. In *3rd ACM Cloud Computing Security Workshop*, 2011.
34. Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *20th Intern. Symp. High Performance Comp. Arch.*, 2014.
35. Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th Intern. Symp. Comp. Arch.*, 2007.
36. Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM Intern. Symp. Microarchitecture*, 2008.
37. J. C. Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4), 1992.
38. W. Wu, E. Zhai, D. I. Wolinsky, B. Ford, L. Gu, and D. Jackowitz. Warding off timing attacks in Deterland. In *Conf. Timely Results in Operating Systems*, 2015.
39. Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Networking*, 23(2), 2015.
40. Y. Yarom and K. Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symp.*, 2014.
41. D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *18th ACM Conf. Comp. & Comm. Sec.*, 2011.
42. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conf. Comp. & Comm. Sec.*, 2012.
43. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conf. Comp. & Comm. Sec.*, 2014.
44. Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conf. Comp. & Comm. Sec.*, 2016.