Düppel : Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud

Yinqian Zhang
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
yinqian@cs.unc.edu

Michael K. Reiter
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
reiter@cs.unc.edu

ABSTRACT

This paper presents the design, implementation and evaluation of a system called Düppel that enables a tenant virtual machine to defend itself from cache-based side-channel attacks in public clouds. Düppel includes defenses for time-shared caches such as per-core L1 and L2 caches. Experiments in the lab and on public clouds show that Düppel effectively obfuscates timing signals available to an attacker VM via these caches and incurs modest performance overheads (at most 7% and usually much less) in the common case of no side-channel attacks. Moreover, Düppel requires no changes to hypervisors or support from cloud operators.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Information flow controls*

General Terms

Security

Keywords

Side-channel attack; cross-VM side channel; cache-based side channel

1. INTRODUCTION

The threat of side-channel attacks in multi-tenant public clouds is becoming a realistic security concern [29, 38]. In such attacks as demonstrated to date, shared CPU caches enable virtual machines (VM) administered by competing organizations to exfiltrate sensitive information from each other. This has recently been shown to be possible despite considerable interference and background "noise" from the hypervisor and other activities on the machine [38].

Most approaches to address these attacks have focused on altering the cloud platform in some way (see Sec. 2). However, to our knowledge, these defenses have not gained traction in existing public clouds. Rather, a typical tenant of a public cloud concerned about these attacks is left with little choice but to try to *defend itself*. One approach is to construct its software to resist side channels (e.g., [21, 18, 11]), but these techniques can result in significant slowdown. A second approach is to run its VMs on an isolated machine (possibly verifying this isolation using side channels itself [37]), though existing cloud providers that offer this service charge more for it.

In this paper we explore a third possibility, namely a method by which a tenant can construct its VMs to automatically inject additional noise into the timings that an attacker might observe from caches. Since these timings are the most common side-channels by which an attacker infers sensitive information about a victim VM, injecting noise into them will generally make the attacker's job more difficult. Our implementation of this idea, called Düppel[1], modifies only the guest OS kernel and is general enough to protect arbitrary types of user-space applications. Düppel can be configured to protect the user-space application, any dynamically linked libraries it uses, or both. Düppel does not need support from hypervisors or cloud providers. To our knowledge, our scheme is the first work that provides tenant VM OS-layer mitigation of cross-VM side channels.

Unlike the noise overcome by previous attacks, the noise injected by Düppel is designed specifically to confound attacks mounted via timing the per-core L1 cache (or per-core L2 cache, if present) on the platform. Düppel does so by repeatedly cleaning the L1 cache alongside the execution of its tenant workload, at a pace that it adjusts based upon the possibility with which timings reflecting the workload execution could actually have been observed from another VM. We also discuss extensions of Düppel to defeat timing attacks via other time-shared resources such as the branch prediction cache, should attacks via this cache [4, 3, 14] someday be adapted to a virtualized setting. We emphasize, however, that even with just addressing the L1 cache, Düppel already interferes with all known cryptographic side-channel attacks that have been demonstrated in a virtualized SMP environment. (See Sec. 8 for further discussion on this point.)

Overhead of Düppel is modest. In tests on Amazon EC2, we show that file download latencies and server throughput over a TLS-protected connection from an Apache web server suffer by at most 4% when Düppel is configured to

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516741.

[1]Düppel takes its name from a radar countermeasure developed by the German Luftwaffe during World War II, in which aircraft disperse clouds of tiny pieces of material to interfere with radar.

protect the OpenSSL library. We also demonstrate using the PARSEC benchmarks and other programs that computational workloads suffer by up to 7% when the applications are protected by Düppel. We believe that these overheads are acceptable given the substantial challenge involved in defending against cache-based side-channels with no help from the underlying hardware or hypervisor.

The rest of this paper is organized as follows. We discuss related work in Sec. 2. We present necessary background in Sec. 3 and then present the design of Düppel in Sec. 4 and Sec. 5. We describe our implementation in Sec. 6 and its evaluation in Sec. 7. We discuss possible extensions and limitations in Sec. 8 and conclude in Sec. 9.

2. RELATED WORK

Side-channel attacks on CPU microarchitectures for extracting cryptographic keys from a victim entity have been studied in different contexts. The particular form of these attacks with which we are concerned are *access-driven attacks*, in which the attacker runs a program on the system that is performing the cryptographic operation of interest. The attacker program monitors usage of a shared hardware component to learn information about the key, e.g., the data cache [27, 25, 24, 31, 13], instruction cache [1, 2, 38], floating-point multiplier [5], or branch-prediction cache [4, 3]. Our work specifically aims at defending against access-driven cache-based side-channel attacks in the cloud, an example of which was recently demonstrated by Zhang et al. [38].

Countermeasures to cache side-channel attacks can be applied to various layers in a computer system. In mitigating cross-VM side channels, prior works can roughly be classified into one of the following categories, based on the layer in which the countermeasure is implemented.

Hardware-layer approaches. The first category includes proposals of new cache designs by applying the idea of resource partitioning (e.g., [26, 33, 34, 12]) or access randomization (e.g., [34, 35, 16]) to mitigate timing channels in CPU caches on the hardware layer. Other works have proposed approaches to eliminate fine-grained timing sources in hardware designs [20]. Compared to implementations in other layers, hardware methods usually have lower performance overhead. However, adoption of new hardware techniques is a complex process, which may involve considerations of side effects (e.g., power consumption) and economic feasibility. Therefore, it might take years before these techniques are merged into production and finally used in clouds.

Hypervisor-layer approaches. Countermeasures in the second category intend to mitigate cache timing side channels by adapting the hypervisor. One direction along this line is to hide nuances in the program execution time, either by providing a fuzzy timer [32] or by forcing all executions to be deterministic [6]. However these approaches will exclude many applications that rely on a fine-grained timer from running in the cloud. A conceptually similar but more comprehensive solution is provided by Li et al. [19], in which all sources of timing channels a VM can observe are identified and categorized; they are mitigated either by aggregating timing events among multiple VM replicas, or by making them deterministic functions of other channels. Another direction is to partition the shared resources in the hypervisor [28, 30, 17]. In particular, Raj et al. [28] statically partition the last level cache (LLC) into several regions and allow VMs to make use of different regions by partitioning physical memory pages accordingly. Kim et al. [17] improved the performance by dynamically partitioning the LLCs and extended the protection to L1 cache as well.

OS-layer approaches Previous OS-layer approaches were mostly proposed to defend against cross-process side-channel attacks, e.g., [27, 31, 13]. To our knowledge, our approach is the first to modify the OS layer to mitigate cross-VM side channels.

Application-level approaches. The last approach is to construct side-channel resistant software implementations. For example, the program counter security model [21, 11] eliminates key-dependent control flows by transforming the software source code. Other efforts focus on side-channel free cryptographic implementations (e.g., [18]). These approaches incur significant overheads in some cases (e.g., Coppens et al. [11] indicate up to 24×).

3. BACKGROUND

3.1 CPU Caching Architectures

Modern CPU microarchitectures extensively make use of hardware caches to speed up expensive operations. The hardware caches on an x86 platform may include data caches, instruction caches, translation lookaside buffers (TLB), trace caches, branch target buffers (BTB) in the branch prediction unit, etc. These caches are similar in their functionalities. The most commonly known caches are data and instruction caches sitting between the processor cores and the main memory, establishing a storage hierarchy in which each level stores the interim data for the next level storage system for quick reference. Most contemporary multi-core processors are equipped with separate L1 data and instruction caches for each core, plus one (or two) unified LLC for the whole CPU package. In some Intel CPUs, a unified L2 cache exists per core, mediating data exchange between L1 caches and the LLC. Modern cache sizes range from several kilobytes (KB) to several gigabytes (GB). They are usually organized as a sequence of blocks denoted *cache lines* with fixed size. Caches are usually *set-associative*. A w-way set-associative cache is partitioned into m sets, each with w lines of size b. If the size of the cache is denoted by c, we have $m = c/(w \times b)$.

3.2 Cache-based Side-Channel Attacks

A technique commonly used in cache-based access-driven side-channel attacks is the PRIME-PROBE protocol. Although in previous works, the attacker and victim in this protocol would typically be OS processes, we describe the protocol here with the attacker and defender being virtual machines, to illustrate the threats relevant to the cloud.

In a PRIME-PROBE protocol, as introduced by Osvik et al. [25], a VCPU (a virtual CPU belonging to a VM) U of the attacker's VM spies on a victim's VCPU V by measuring the cache load in a given cache in the following manner:

PRIME: U fills one or more cache sets.

IDLE: U waits for a prespecified PRIME-PROBE *interval* while the cache is utilized by V.

PROBE: U times the duration to refill the same cache sets with the same data to measure V's cache activity on those sets.

Cache activity induced by V during the interval between U's PRIME and PROBE will evict U's data from the cache sets and replace them with V's. This will result in a noticeably higher timing measurement in U's PROBE phase than if there had been little activity from V. Of course, PROBing also accomplishes PRIMing the cache sets (i.e., evicting all data other than U's), and so repeatedly PROBing, with one PRIME-PROBE interval between each PROBE, eliminates the need to separately conduct a PRIME step.

A CPU cache, in regard to side-channel attacks, is either time-shared by the attacker and victim, or simultaneously shared by the two entities. An example of a time-shared cache is the L1 cache in an SMT-disabled[2] processor, in which the cache is dedicated to a single CPU core and can only be shared by two threads running on the same core in turns. It has been shown in previous research that SMT facilitates CPU-based side channel attacks because two threads can simultaneously use CPU resources [27, 31, 1, 2, 4]. As most cloud infrastructures disable SMT for performance or security purposes (see Sec. 8.2), per-core caches in these environments are time-shared, and so Düppel focuses on defending against side channels in this case.

4. GOALS

The anticipated attack scenario is in public IaaS clouds, where the attacker has full control of a VM co-located with the Düppel-protected VM and is capable of exploiting per-core caches as side channels to exfiltrate sensitive information. We assume the underlying CPU is based on x86 architecture and is *not* equipped with SMT. Such hardware configurations dominate modern public clouds. The two VMs in question may time-share the same CPU core, thus time-sharing the L1 instruction cache, L1 data cache, the unified L2 cache (if any), BTBs, TLBs and other caching architectures in a CPU core. We further assume the attacker can obtain a copy of the software stack running in the Düppel-protected system and so can experiment with it to learn the cache behavior that it induces. The cloud provider controls the hypervisor, which operates as is—it neither facilitates the side-channel attacks nor does anything to thwart such threats.

In light of this threat model, we have the following goals for Düppel:

GOAL 1. *Düppel should mitigate side-channel attacks via time-shared caches.*

The implementation of Düppel that we develop here addresses side channels using the L1 (or, if any, L2) per-core caches, which are time-shared caches. The principles for addressing side channels in these caches should apply to addressing them in other time-shared caches, as well.

GOAL 2. *Düppel should not require any hypervisor modification.*

An important design principle of Düppel is to permit its adoption by cloud tenants on modern cloud infrastructures without any additional support from the cloud providers.

GOAL 3. *Düppel should not require modifying applications or libraries.*

Though many secrets that might be targeted in side-channel attacks are handled in applications and third-party linked libraries, the sheer number and diversity of applications and libraries makes modifying all of them an unattractive proposition. Instead, a goal for Düppel is to protect such secrets without modifications to these components. We therefore adopted the guest OS kernel as the (sole) location in which to implement our techniques.

GOAL 4. *Düppel should induce little performance overhead.*

To be used in practice, Düppel must impose little performance burden on the system. This may require Düppel to operate in different modes during its lifecycle.

5. CLEANSING TIME-SHARED CACHES

Because attacks on time-shared caches must involve cache PROBing via CPU preemption, the intuition behind Düppel is that a guest VM (and the OS that runs on it) can protect its own execution against side-channel attacks by adding noise to these caches very frequently so that side-channel readings by the attacker are confounded by noise added between PRIMEs and PROBEs.

5.1 Basic Design

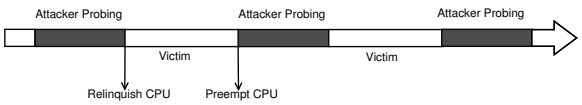
Düppel employs periodic cache cleansing to mitigate side channels in time-shared caches, e.g., L1 data/instruction caches and unified, per-core L2 caches, if any. The principles for addressing these caches could be used to address branch predication caches, as well. The basic idea is illustrated in Fig. 1. As described in Sec. 3.2, PRIME-PROBE protocols on time-shared caches work as illustrated in Fig. 1(a): the attacker periodically preempts the victim's VCPU by gaming the hypervisor scheduler and then PROBEs the cache, which takes some time due to cache/memory access latency. After the attacker relinquishes the physical CPU, the victim VCPU is scheduled for a short period until the attacker preempts it again.

Periodic cache cleansing, as illustrated in Fig. 1(b), works by cleansing the time-shared cache (ideally) between the attacker's PROBEs. Specifically, cleansing involves Düppel itself PRIMing the cache in random order until all of the entries have been evicted. The cache cleansing process is accomplished by an kernel thread that is periodically invoked by Düppel.

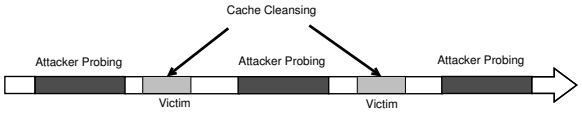
Key to the success of this technique is a fine-grained resolution timer in the guest OS kernel that can be periodically triggered to execute cache cleansing. There are several possible timers that might be utilized for this purpose:

- **Hardware Timers** On x86 platforms, the hardware timers include the programmable interval timer (PIT), the local advanced programmable interrupt controller timer (Local APIC), the high precision event timer (HPET), and the advanced configuration and power interface (ACPI) power management timer. Performance monitoring units (PMU) can also be regarded as a type of hardware timer, since they can be set to trigger upon hardware events that occur with predictable rates. In modern IaaS clouds, e.g., Amazon EC2 and Rackspace, most guest VMs run on paravirtualized Xen platforms. However, in such settings, none of the above hardware timers can be set to trigger by a guest VM.

[2]"SMT" is Simultaneous Multi-Threading, or HyperThreading in Intel's terminology.



(a) Attackers exploits time-shared caches by preempting victim VCPUs.



(b) Periodic time-shared cache cleansing.

Figure 1: Attack and defense on time-shared caches

- **Software Timers** A software timer may be as complex as a Linux high-resolution timer (hrtimer) or as simple as a piece of code that busy loops and periodically invokes the execution of the timer handling code using interrupts. These interrupts can include network I/O interrupts and inter-processor interrupts (IPI), the latter of which was used by Zhang et al. [38]. In paravirtualized VMs, although IPIs are also virtualized (as are I/O interrupts), they offer more stable and shorter interrupt delays. The delay of an IPI timer can be as low as 3μs, while that of the hrtimer is usually higher than 15 to 20μs.

The timers used in Düppel are hrtimers and software timers generated by IPIs. Periodic cache cleansing activities can operate in two modes: *sentinel* mode and *battle* mode. The sentinel mode is used when frequent cache cleansing is not critical, and thus the interval between two cleansings can be longer—as long as Düppel can detect abnormal activities and switch to battle mode quickly. The hrtimers are employed in sentinel mode because it induces lower performance overhead (conforming to Goal 4). The much more rapid IPI timers are used in the battle mode, where intervals between cache cleansings are required to be as short as possible and the performance overhead is less of a concern when under active side-channel attacks.

5.2 Optimizations

We discuss in this section a few design optimizations that reduce the performance impact of Düppel.

5.2.1 Limiting the Protection Scope

Düppel's periodic cleansing of the L1 cache impacts the performance of the victim's application. As such, ideally Düppel would be triggered only when necessary. Fortunately, we can limit the scope of protection by allowing the users of Düppel to define specific operations as sensitive so that cache cleansing is triggered only when sensitive operations run. For instance, in a TLS-protected web server, cryptographic routines in the OpenSSL library might be considered sensitive operations. The goal is to automatically enable Düppel when needed and disable Düppel when sensitive operations finish.

Düppel implements this mechanism by exploiting CPU page-level execution protections. Düppel first marks the memory pages that contain the instructions of the sensitive operations as non-executable. Thus every time these instructions are executed, page faults trigger Düppel. Then Düppel can disable the execution protection to prevent further page faults and start the cache cleansing timer to trigger cleansing each of the time-shared caches N times (Sec. 5.1). At the end of the N-th cleansing, the execution protection is re-enabled.

If N is very small, unfinished sensitive operations will trigger the page fault again to initiate another N timer interrupts, and the overhead of frequently changing page-level protections can be huge. If N is too big, Düppel will keep running after the sensitive operation ends. In our design, Düppel dynamically adjusts N with the following algorithms: N changes value in the range of $[N_{min}, N_{max}]$, each time by adding Δ. Δ may vary in the range of $[\Delta_{min}, \Delta_{max}]$ and $[-\Delta_{max}, -\Delta_{min}]$. If the page fault takes place right after (i.e., no more than a threshold of L cycles) re-enabling execution protection, Δ doubles if it was positive or becomes Δ_{min} otherwise; if the page fault happens later than L, Δ doubles in the negative direction or becomes $-\Delta_{min}$.

Alternative Approaches. Instead of exploiting page execution protection, another way to dynamically enable and disable Düppel is to modify the user level application. For example, to protect dynamically linked libraries, the most convenient way is to modify the procedure linkage table entries of the relevant processes so that every library call related to the protected library function will first go through a wrapper function. Similarly it is possible to use preloaded libraries to overwrite (to provide a wrapper for) the library routines. In order to protect sensitive operations in the executables, one either needs to instrument the functions in the executables or insert breakpoints and use `ptrace` to intercept `SIGTRAP` signals sent to the protected process. These approaches, however, violate Goal 3 and so are not adopted by Düppel.

5.2.2 Skipping Unnecessary Cache Cleansings

We further optimize Düppel to skip unnecessary cache cleansings. Before cleansing the L1 cache (and any other time-shared caches), Düppel first determines if the VCPU it is running on has been preempted since it last ran on the VCPU. If preemption took place, it then determines if the process interrupted by the current timer interrupt is related to the sensitive operations to be protected. If both answers are yes, Düppel will cleanse the cache; otherwise, it skips the cache cleansing step.

A paravirtualized Xen guest VM can detect VCPU preemption by exploiting shared data structures between the guest and the hypervisor. In particular, Xen uses a shared memory page that contains a value of the timestamp counter to help the guest VM keep track of the system time. The counter value is updated at every hypervisor context switch if it is different from the version stored in the hypervisor, which is only changed every one second. At the end of each cache cleansing, Düppel modifies the shared counter value by one, which compared to the 64-bit counter value is rather small. (Greater changes may confuse the guest OS.) The next context switch will force a change of the counter value, and therefore will be detected by Düppel. One caveat is that the guest can voluntarily relinquish the VCPU, as well, which occurs when the VCPU is idle or at certain points during VCPU setup. Düppel instruments the code locations at which the guest VM kernel issues hypercalls to relinquish

the VCPU, to help distinguish voluntary context switches and VCPU preemptions.

6. IMPLEMENTATION

We implemented Düppel as a kernel component (1.4K lines of C code) in paravirtualized Linux that runs on Xen guest virtual machines. Specifically, our prototype implementation modifies Linux kernel v2.6.32. It operates on both Xen 4.0 hypervisors in our lab setting and Xen 3.0 hypervisors (as reported) in Amazon Web Services.

Architecture. We provided an entry in the `/proc` file system which allows user-space tools to input user specified parameters to Düppel. These parameters include names of the protected applications and libraries. (It is currently possible to protect a subset or all of the libraries in one or multiple processes.) Our implementation extends the memory region management component of the Linux kernel to monitor the creation, deletion and splitting of the memory regions related to the protected applications and libraries. A list of pointers to these memory regions is maintained by Düppel. This list is adjusted dynamically by Düppel with the creation and termination of the protected processes. Read-Copy-Update (RCU) synchronization mechanisms are used to guarantee exclusive modifications and wait-free reads to this data structure with very low overhead. All memory pages belonging to the memory regions maintained in the list are disallowed to be executed by modifying the corresponding page table entries. This can be enforced by modifying the default page protection flags of the memory regions, which will be propagated to all newly mapped memory pages due to on-demand paging [9]. Although the same memory pages can also be mapped to other processes in the system, they can be executed normally since they have separate page tables.

We also instrumented the page fault handler so that every page fault goes through an additional check: faults caused by user-space execute accesses to pages that are already in memory are passed to Düppel to further examine if the faults take place in the protected memory regions. As such page faults are otherwise rare, the performance overhead caused by modifications to the critical page fault handling procedure is minimal.

Workflow of Düppel. The main logic of Düppel works as follows: Düppel traps the page faults caused by executing the protected memory regions, and it then enables the page execution for all pages belonging to the list of memory regions. Then it initiates the periodic cache cleansing. After N cleansings have been performed, Düppel stops and enables page-execution protection in the page tables, indicating the end of a protection epoch.

Additionally, a counter recording the times a VCPU is preempted is maintained by Düppel to determine in which mode it should operate cache cleansing, i.e., sentinel mode or battle mode (Sec. 5.1). If the counter is greater than a threshold—empirically determined as 10 in our evaluation (see Sec. 7)—Düppel enters the battle mode; otherwise it runs in sentinel mode. The counter is updated as an exponential moving average (or EMA) of the number of VCPU preemptions in the current and previous time epochs, with $\lambda = 0.5$ where λ is a constant that determines the depth of memory of the EMA [15] . Each time epoch is roughly 1ms (rounded down to a number of CPU cycles that is a power of 2 for efficiency). Even if the attacker is aware of Düppel's parameters and refrains from preempting the victim (and so PROBing) less than 10 times per ms to cause Düppel to stay in sentinel mode, the L1 caches will nevertheless be cleansed at least 50 times per millisecond.

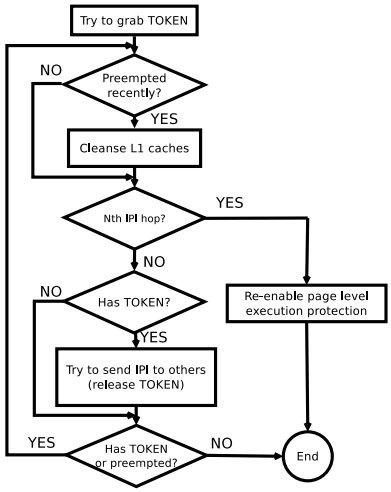


Figure 2: A token-based IPI synchronization algorithm.

Cache cleansing modes and operations. When operated in sentinel mode, Düppel induces periodic timer interrupts and then tries to detect preemption (see Sec. 5.2.2) on all VCPUs. Once a preemption on a VCPU is detected, it sends an IPI to launch the next timer on that VCPU. As more preemptions are detected, Düppel enters battle mode, in which it will send IPIs to all the other cores that are available to process the IPI interrupts. The reason for sending IPIs to all such VCPUs (vs. only one) is to avoid cases where the VCPU being sent an IPI is not running (e.g., is preempted or out of credit to run), thereby causing the cache-cleansing interrupts to pause.

We implemented a token-based VCPU synchronization algorithm in Düppel (shown in Fig. 2) to avoid race conditions. An atomic variable serves as the token and one or more VCPUs simultaneously in the IPI context will try an *atomic exchange* to grab the token. So, only one VCPU will get it. After the cache cleansing job is done (or skipped), the VCPU with the token will try to send IPIs to all other VCPUs that have finished the last-round cache cleansing to pass along the token. As such, releasing tokens may not always be successful. Failure in passing the token or detection of another VCPU preemption at this point will force it to go over the cache cleaning cycle again. If the preemption counter goes below the threshold value, Düppel jumps back to sentinel mode.

7. EVALUATION

7.1 Security Evaluation

In this section, we report the results of our security evaluation of Düppel in a lab environment. Our lab testbed was equipped with a single-socket quad-core Intel Core 2 Q9650 processor with an operating frequency of 3.0GHz. It had two levels of caches: both L1 data and instruction caches were

8-way set-associative and 32KB in size; two 24-way 6MB unified L2 caches, each served two CPU cores. All caches had 64-byte cache lines. We ran a Xen 4.0 hypervisor on the hardware; Dom0, the management domain in Xen, was given a single VCPU.

To evaluate Düppel under attack, i.e., when it operates with the presence of a side-channel attacker, we assigned two VCPUs to the VM that ran Düppel, and two VCPUs to a co-resident attacker VM. We facilitated the attack by pinning the VCPUs to the physical cores so that one attacker VCPU and one Düppel VCPU shared the same L1 caches. Düppel was configured to cleanse both the L1 data cache and the L1 instruction cache.

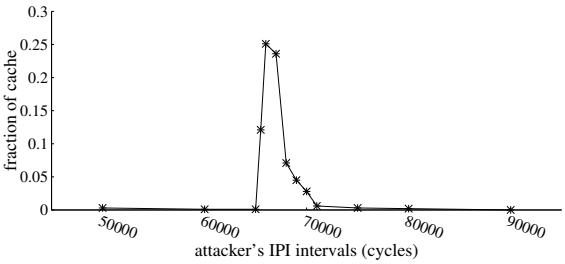
7.1.1 Attacking a "Dummy" Victim

Victim. The "dummy" victim application in these experiments had two "dummy" functions composed of "nop" instructions. The instructions in each of them were mapped to different regions in the L1 instruction cache, occupying all cache lines in several cache sets. The victim application executed in a loop, alternating between these two functions. As such, the victim had distinguishable access patterns in the L1 instruction cache—all cache sets associated with the two functions were thoroughly evicted while other cache sets were almost entirely untouched.

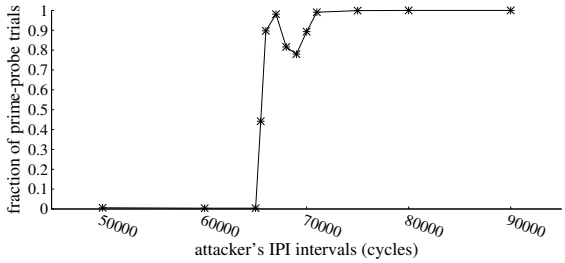
Attacker. We leveraged the side-channel attack code of Zhang et al. [38], which exploits IPIs to periodically preempt the victim VCPU and PRIME-PROBE the L1 instruction cache to collect timing results from each cache set. The two VCPUs in the attacker's VM have different roles: the *IPI VCPU* keeps looping and periodically sends IPIs to the *attacker VCPU*. The *attacker VCPU* is otherwise idle and only executes cache PROBing functions when triggered by receiving IPIs. The interval between two consecutive IPIs sent to the *attacker VCPU* is determined by the attacker, and is varied in our experiments as a tuned parameter. Each cache PROBing takes about 42000 CPU cycles (roughly 14μs), and one hypervisor context switch takes about 600 cycles. The IPI intervals, therefore, are the desired PRIME-PROBE intervals plus the time required for PROBing and context switches. Longer IPI intervals will give more chances to the victim to run, but multiple functions may run in the same PRIME-PROBE interval, leaving the cache too noisy to interpret. Shorter IPI intervals may cause more than one IPI to be delivered at the same time, making the PRIME-PROBE protocol fail. The range for the attacker's IPI interval that we evaluated was from 50000 CPU cycles to 90000 cycles.

Effectiveness of cache cleansing. Our first attempt in evaluating the effectiveness of Düppel's cache cleansing was to measure the fraction of cache lines *not* evicted by Düppel prior to the attacker PROBing the cache. Information leaks can happen either when Düppel is not run between the attacker's PRIME and PROBE, or when Düppel is run but it does not have enough time to cleanse the entire cache before being preempted by the attacker. The results of our tests are shown in Fig. 3(a). From the graph, we can see that when the attacker chooses a long IPI interval (>70000 cycles) or a short IPI interval (<65000 cycles), the potential information leak is minimal.

When the attacker's IPI interval is short (<65000 cycles), contention may result in IPIs issued by Düppel being significantly delayed or even "starved" by the attacker. However, recall that Düppel is designed to repeatedly cleanse the cache if it detects preemption; so even with less frequent IPI interrupts, Düppel will still consume most of the CPU time in cache cleansing to make sure the victim secret is not leaked through side channels. With longer IPI intervals (>70000 cycles), the IPIs issued by Düppel will be delivered on time and Düppel is able to finish the cache cleansing without being preempted, thus allowing the victim to run normally while being protected. However, when the IPI interval is between 65000 and 70000 cycles, Düppel only gets to cleanse a fraction of the cache before being preempted. This is the worst-case scenario for Düppel's protection scheme. Fig. 3(b) also helps illustrate this phenomenon. The y-axis in this figure is the fraction of attacker's PRIME-PROBE trials in which the victim application ran at all.



(a) Average fraction of cache potentially occupied by victim application data upon preemption

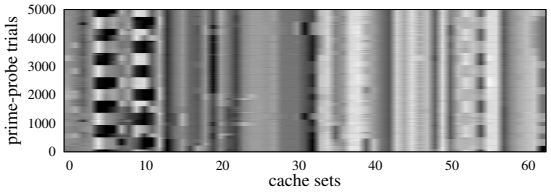


(b) IPI intervals during which the victim application executed at all

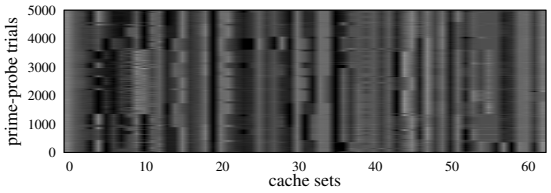
Figure 3: Cache cleansing effectiveness under different attacker IPI intervals.

In the worst case, when the attacker's IPI interval is 66000 CPU cycles, about 25% of the cache lines are not evicted by Düppel but might contain victim application data. This provides the opportunity for some information leakage to the attacker, but the risk is far less than in the absence of Düppel. Fig. 4 shows the attacker's cache readings from the PRIME-PROBE trials in this case. Each column in the heatmap represents the cache set indicated on the x-axis; the y-axis is the index of PRIME-PROBE readings. In Fig. 4(a) we can see that without Düppel, the attacker can easily observe the cache pattern (the alternating cache usage on the left) of the victim application. In contrast, even with 25% of the cache lines possibly containing victim application data not evicted by Düppel, as shown in Fig. 4(b), the victim's cache patterns are substantially obfuscated.

More quantitatively, in this worst-case scenario we measured the difference, for each cache set, of the average PROBE results (averaged over 100000 PRIME-PROBE trials) for that cache set when one dummy function ran versus the aver-



(a) w/o Düppel



(b) w/ Düppel

Figure 4: Attacker's view of L1 instruction cache timings. The x-axis represents the cache sets; the y-axis represents the attacker's prime-probe trials. The darker the cell in the heatmap, the longer it takes the attacker to probe the cache.

age for that cache set when the other dummy function did. Without Düppel enabled, this difference-per-cache-set, averaged over all cache sets, was 32.7 cycles with a standard deviation of 13.2 cycles. However, cache sets to which one or the other dummy function mapped clearly divulged which dummy function had executed, exhibiting a difference-per-cache-set up to 190 CPU cycles — almost 12 standard deviations above the mean. (By comparison, the maximum difference-per-cache-set among cache sets not associated with either dummy function was only 22 cycles.) With Düppel enabled, the difference-per-cache-set, averaged over all cache sets, was 3.3 cycles with a standard deviation of 2.1 cycles. Among those cache sets to which either dummy function mapped, the maximum difference-per-cache-set dropped to 8 cycles, which is less than three standard deviations above the mean and, moreover, even less than the maximum difference-per-cache-set among cache sets not associated with either dummy function (which stayed basically unchanged from the Düppel-disabled case). In this respect, Düppel brought the timings of the cache sets occupied by the dummy functions largely "in line" with the timings of other cache sets.

7.1.2 *Case Study: Square and Multiply*

As a case study, we examined the victim application attacked by Zhang et al. [38], namely the modular exponentiation implementation in the `libgcrypt` v.1.5.0 cryptographic library, which uses a textbook square-and-multiply algorithm. In this attack, the goal of the attacker was to extract a secret exponent used in this routine by PROBing the L1 instruction cache to infer the sequence of squares and multiplies executed. Here we focus on the accuracy of the first stage of the Zhang et al. attack, which involves classification of PRIME-PROBE timing vectors using a three-class support vector machine (SVM) — one class for "square", one for "multiply", and one for "other". While subsequent stages of the attack pipeline can filter out a small rate of misclassifications, a reasonably accuracy of the SVM classifications is necessary for the attack to work.

		Classification		
		Square	Multiply	Other
Op	Square	1740 (0.84)	43 (0.02)	298 (0.14)
	Multiply	14 (0.01)	2213 (0.94)	123 (0.05)
	Other	272 (0.05)	225 (0.04)	5072 (0.91)

(a) w/o Düppel

		Classification		
		Square	Multiply	Other
Op	Square	26 (0.01)	699 (0.39)	1055 (0.59)
	Multiply	26 (0.01)	938 (0.48)	987 (0.51)
	Other	86 (0.01)	3367 (0.54)	2816 (0.45)

(b) w/ Düppel

Figure 5: Confusion matrix of SVM classification in tests in Sec. 7.1.2.

Training the SVM was performed using the same approach as taken in the Zhang et al. work. In particular, to collect data for training, the "victim" repeatedly performed modular exponentiations, while informing the data collector via shared memory when it begins or ends a square or multiply. In this way, the data collector could associate its PROBE results of the L1 instruction cache with the ground-truth operation that was being performed when the PROBE was performed. Like Zhang et al. [38] we used a linear kernel in the SVM classifier. In each experiment, the SVM was trained with the labeled 90000 PRIME-PROBE results and then tested on an additional 10000 PRIME-PROBE results.

Fig. 5 shows the confusion matrix that resulted from an experiment in which both training and testing were conducted with Düppel disabled (Fig. 5(a)) and one in which both training and testing were conducted with Düppel enabled (Fig. 5(b)). With Düppel disabled, the SVM classifies the testing PROBE results with accuracy over 90%, which is well enough in our experience to enable the subsequent stages of the Zhang et al. attack. When Düppel is enabled, however, the SVM classification fails badly with an accuracy of only about 38%. In our experience, continuing the Zhang et al. attack from this point would be extremely difficult. This is, of course, not a proof that no exploitable side-channel still exists, and it is conceivable that a persistent and adaptive attacker could still make progress; however, we believe that Düppel should substantially increase the complexity of doing so.

7.2 Performance Evaluation

We conducted the performance evaluation of Düppel primarily in a public cloud environment, Amazon Web Services. The specification of processors in the Amazon EC2 cloud environment was not of our own choice. But through `cpuid` instructions, we determined that the platform was equipped with two 2.4GHz Intel Xeon E5645 processors, each of which had six cores per package with hyper-threading supported but disabled. Its L1 data caches were 32KB, 8-way set-associative; L1 instruction caches were also 32KB but with only 4 ways. One 8-way unified L2 cache was dedicated to each core, with 256KB capacity. Additionally, it had one 12MB 16-way L3 cache per CPU package. All caches had 64-byte cache lines. The shared info pages [10, Ch. 3] reported the Xen version to be `xen-3.0-x86_64`. The number

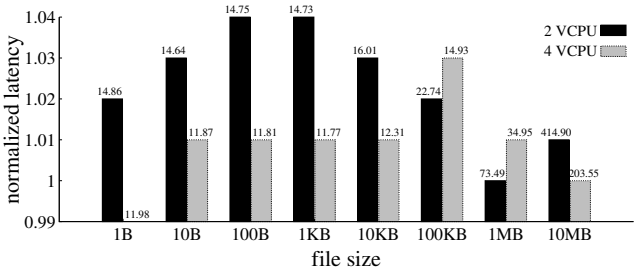
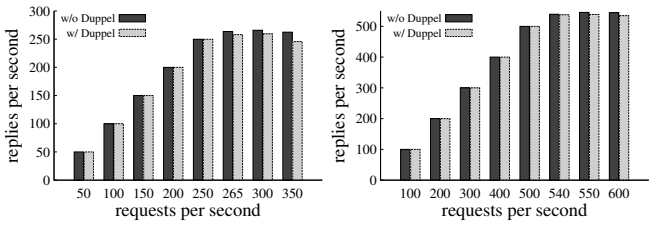


Figure 6: Düppel's overheads for file download latency with different file sizes. Labels on top of the bars represent the baseline latency (without Düppel) in ms.



(a) 2-VCPU server (b) 4-VCPU server

Figure 7: File download rate in replies per second with different requests per second, with and without Düppel.

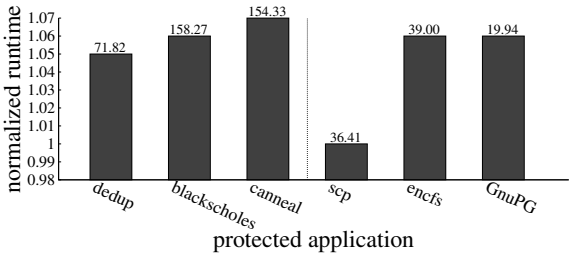


Figure 8: Runtime overhead of Düppel for different applications. Labels on top of the bars represent the baseline runtime (without Düppel) in seconds.

of Dom0 VCPUs was unknown. As in the lab settings, we implemented Düppel in Linux kernel v2.6.32 in the cloud. Düppel was configured to protect the L1 caches and the unified L2 cache.

7.2.1 Securing TLS Libraries

In the following experiments we evaluate the performance overhead of Düppel when protecting `apache2` processes and the OpenSSL library (`libcrypto.so`) to which they were linked. We are particularly interested in preventing side channel attacks against the cryptographic operations provided by `libcrypto.so` during the Transport Layer Security (TLS) protocol used in `https` connections. We ran an apache 2.2.24 web server with `libcrypto.so` version 1.0.0. We ran the web server in one VM in the EC2 cloud and ran the client in another VM in the same availability zone in order to minimize network latency.

File download latency Fig. 6 shows results of an experiment to test the impact of Düppel on file download latency. We ran `apache bench` on the client and requested static pages with varying file sizes. The results reported are average values of 10000 requests for each file size. These experiments show that the impact of Düppel on file download latency was less than 4% in the worst case.

File download throughput Fig. 7 shows the file download throughput degradation induced by Düppel in our web server experiments. In these experiments, we used another web server performance measurement tool, `httperf` [22], to produce the largest rate of requests for a 177-byte file (the default `index.html` file installed with `apache2`) for which the server could keep up. In order to saturate a server with 2 VCPUs, the requests were sent from 2 clients running `httperf` from distinct machines in the same availability zone in AWS; for the 4-VCPU server, 4 clients were employed in the test. Only one request was issued per connection and the SSL sessions were not reused. The timeout value for which `httperf` waited for the server's response was set to 1s. We confirmed by running the `top` utility that at its saturation point the server was CPU-bound; the throughput and server CPU usage both reached their limits simultaneously. Thus the throughput degradation was actually caused by Düppel rather than other factors. As this figure illustrates, the maximum throughput of the 2-VCPU server dropped from 267.1 replies per second to 258 replies per second, yielding a degradation of 3.4%. In the 4-VCPU case, the server's maximum throughput dropped only 1.4%, decaying from 545.3 replies per second to 537.7 replies per second. In all cases, the throughput overheads induced by Düppel were modest.

Local experiments We repeated the above file download latency and throughput experiments in a more controlled environment in our lab. In these tests, the clients and server were connected through a 1Gb/s LAN. The qualitative results shown in Fig. 6 and Fig. 7 persisted in our local tests, e.g., with an induced latency degradation by Düppel of at most 3% and a throughput degradation of at most 2.1%.

7.2.2 Securing Other Applications

We also evaluated Düppel in protecting other applications. We first picked three applications, `blackscholes`, `canneal` and `dedup`, from the PARSEC benchmarks [7, 8] to simulate different types of applications. `blackscholes` simulates financial analysis and, in particular, calculates the prices of a portfolio of options using Black Scholes partial differential equations. `canneal` uses cache-aware simulated annealing to design chips that minimize routing costs. `dedup` is short for "deduplication," which is a compression approach that combines global and local compression in order to obtain a high compression ratio. We selected these benchmark applications to represent CPU-bound applications with different amounts of memory and cache usage. The inputs to these benchmarks were `native` (see [8]) and the number of threads was the same as the number of VCPUs in the VM. We specified the entire executables to be sensitive, and so Düppel was always enabled while the programs were running.

In addition, we selected three applications that involve cryptographic operations, which are more likely to be of interest to attackers: `GnuPG`, `encfs`, and `scp`. `GnuPG` is part of the GNU project and implements the OpenPGP stan-

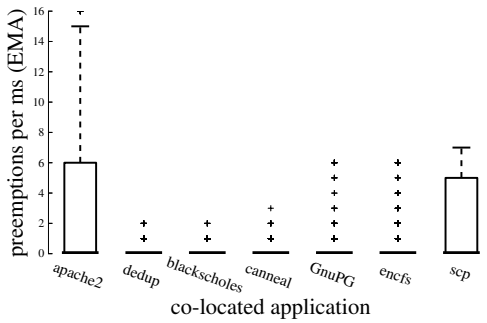


Figure 9: Victim VCPU preemptions per ms **(exponential moving average) induced by co-located application in lab.**

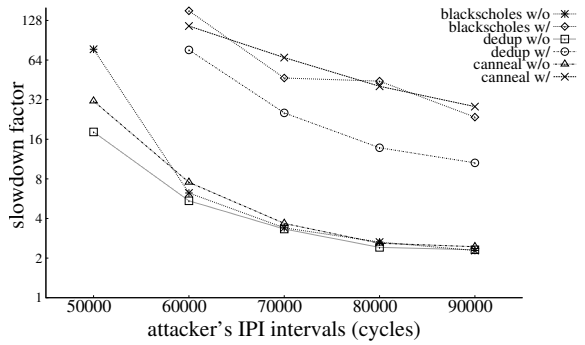


Figure 10: Performance overhead (averaged over five runs) when under prime-probe attacks on L1 instruction caches, w/ and w/o Düppel.

dard. We evaluated the time for it to decrypt a 1GB file encrypted using ElGamal encryption, with Düppel specified to protect `libgcrypt.so`. `encfs` is an encrypted filesystem in userspace. By encrypting a 1GB file, we evaluated the runtime of its file-encryption procedure with Düppel protecting its `libcrypto.so` library. `scp` is a network data transfer protocol built on top of secure shell (`ssh`). We transferred a 1GB file using `scp` to evaluate the latency overhead due to Düppel, again configured to protect `libcrypto.so`. All experiments above were run 10 times (except for `dedup` which was run 30 times due to large variance) on a 2-VCPU server in the AWS cloud. The average runtime degradations are shown in Fig. 8. As can be seen there, Düppel induced less than 7% performance overhead in all cases.

7.2.3 Sentinel/Battle Mode Switching

As described in Sec. 5.1 and Sec. 6, a switch from sentinel mode to battle mode occurs whenever the moving average of the number of VCPU preemptions per millisecond exceeds an empirically determined, fixed threshold of 10 in our prototype. We developed a set of experiments to examine if this threshold will trigger spurious mode switches when co-located with regular, benign applications. In particular, we employed techniques similar to that described in Sec. 5.2.2 to detect VCPU preemptions. Instead of trying to detect a VCPU preemption during each timer interrupt, we tested VCPU preemptions in a loop so that every preemption was captured. We first ran experiments in our local testbed in which an application running on another VM (with only one VCPU) was pinned to share a core with the VM in which we counted preemptions. As such, the number of VCPU preemptions caused by the benchmark application on the shared CPU core was counted. Fig. 9 illustrates the boxplot of the number of VCPU preemptions per millisecond (exponential moving average), where each box shows the first, second and third quartiles, each whisker extends to cover points within 1.5× the interquartile range, and outliers are marked with plus signs ("+"). As shown, these benchmark applications rarely caused the exponential moving average of VCPU preemptions to exceed the threshold of 10; e.g., the `apache2` web server induced a false alarm rate of 3% and all other benchmarks induced no false alarms. It is worth noting that in this test the `apache2` web server had been saturated already.

We also ran experiments in the Amazon EC2 cloud, specifically on four distinct instance types ("m1 medium", "m1 large", "m1 xlarge" and "c1 large") in each of three different availability zones in the US East region ("us-east-1a", "us-east-1c" and "us-east-1d"). The experiments were conducted from 8:00am Aug 10, 2013 to 8:00am Aug 11, 2013. Since we had no knowledge of the applications co-located with our VMs in the cloud, we assumed that the co-resident VMs and their applications were "benign" in terms of side channels. In these tests, the *maximum* false alarm rate witnessed on any of these machines was 8.22×10^{-6}.

7.2.4 Performance Overhead When Being Attacked

Cache-based side-channel attacks, especially on the L1 cache, are essentially also performance degradation attacks. Moreover, Düppel will further degrade the whole system's performance when being attacked. In Fig. 10 we show the results of our lab evaluation of the runtime overhead on `blackscholes`, `canneal` and `dedup` benchmarks under side-channel attacks on the L1 instruction caches, with and without Düppel. The victim was pinned to run only on the core being attacked for these tests. As can be seen in the figure, the attacker alone degraded the victim's performance by up to two orders of magnitude, and Düppel only compounded that cost. Suffering such a denial-of-service may be preferable to succumbing to side-channel attacks. Moreover, since VCPUs are usually not pinned in real clouds, the duration of such a denial-of-service can be expected to be short.

8. DISCUSSION

8.1 Extensions of Düppel

Extension to other cache side channels. In principle, Düppel can defend against side-channel attacks on other types of time-shared caches, as well. For instance, the BTB and TLB can be cleansed together with L1 caches. However, BTB attacks were only demonstrated in non-virtualized systems, while cross-VM attacks on TLBs are unlikely because TLBs are flushed on context switch.

Extension to other clouds. Düppel can be implemented in the guest OS for cloud platforms other than Amazon, as well, as long as the virtualization substrate supports paravirtualized Xen guests and customized OS kernels. We expect Düppel to also work on Rackspace and GoGrid, for example.

Extension to kernel routines. Düppel can be extended to protect sensitive operations in the guest OS kernel, as well. The difficulty, however, is that the Linux kernel is monolithic, and so the code and data memory sections of kernel routines are not cleanly separated from the rest of the kernel. Düppel will have to look for all memory pages that contain kernel routines and mark them as non-executable.

8.2 Limitations

SMT-enabled cloud platforms. When SMT is enabled on processors, some time-shared caches can become simultaneously shared caches (e.g., L1 caches). Düppel may not be effective in this case. However, from our previous experience [38], as well as prior documented evidence [27, 25], SMT-enabled CPUs are more vulnerable to side-channel attacks. Moreover, as clouds often charge by CPU computing units, with SMT-enabled cores the estimated computing units per VM would become unreliable if more than one VCPU shared the same CPU core at the same time. Therefore, we anticipate that cloud providers will not enable SMT in production clouds for both security and accounting considerations.

The simultaneously shared caches. A simultaneously shared cache, e.g., the last level cache, might also be targeted by side-channel attacks. Again, Düppel does not address this possibility. Compared with time-shared caches, PROBing simultaneously shared caches can be done without VCPU preemption and with frequencies limited only by the cache and memory access latencies and the sizes of the caches utilized. Although fine-grained cache observations are possible in such situations, the challenges that attackers face to extract sensitive information may be prohibitive. The first challenge stems from the fact that most LLCs are physically indexed. As such, attackers must have prior knowledge of the physical addresses of the memory regions of the victim application [23], which is usually allocated dynamically and may vary each time it is run. Second, exploiting last level caches when the victim and attacker run on different cores can be sensitive to cache coherence properties. For example, PROBing on an LLC in an exclusive caching hierarchy (as compared with inclusive caches) will not invalidate contents in the lower-level caches of another core. As such, the victim's activity may be hidden in lower level caches and never leaked to the last level cache at all. Moreover, to our knowledge, last level caches in the processors used in cloud platforms usually serve more than two cores; in this case, side-channel observations are subject to more background noise from cores that are not running the victim application. We believe for these reasons, side-channel attacks in simultaneously shared caches in virtualized SMP environments have been limited so far to extracting only coarse information [29] or have needed to leverage additional hypervisor features (e.g., sharing memory pages between the victim and attacker VMs [36]).

9. CONCLUSION

In this paper, we presented Düppel, a system to mitigate cache side channels in public clouds by retrofitting commodity operating systems. Düppel cleanses time-shared caches (e.g., per-core L1 and L2 caches) to confound side channels through them. We detailed our implementation of Düppel in Linux for paravirtualized Xen, demonstrated the security effectiveness of our prototype in tests on lab machines, and evaluated the performance impact of our prototype on a public cloud. To our knowledge, Düppel is the first general and efficient technique to enable tenants to defend themselves from cache-based side-channel attacks in public clouds, without help from the hypervisor or cloud operator.

9.1 Acknowledgements

We thank Dr. Weidong Cui and the anonymous reviewers for suggestions that led to improvements to this paper. This work was supported in part by NSF grants 0910483 and 1330599, the Science of Security Lablet at North Carolina State University, a gift from VMWare and Google PhD Fellowship to Yinqian Zhang.

10. REFERENCES

[1] O. Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pages 11–18, 2007.

[2] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, pages 110–124, 2010.

[3] O. Aciiçmez, S. Gueron, and J.-P. Seifert. New branch prediction vulnerabilities in openSSL and necessary software countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, pages 185–203, 2007.

[4] O. Aciiçmez, C. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, pages 312–320, 2007.

[5] O. Aciicmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, 2007.

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

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[8] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[9] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates, 2005.

[10] D. Chisnall. *The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, Nov. 2007.

[11] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[12] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4), Jan. 2012.

[13] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[14] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, May 2013.

[15] J. S. Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210, 1986.

[16] G. Keramidas, A. Antonopoulos, D. Serpanos, and S. Kaxiras. Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12:221–230, 2008.

[17] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[18] R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202, 2008.

[19] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2013.

[20] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 118–129, 2012.

[21] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, pages 156–168, 2006.

[22] D. Mosberger and T. Jin. httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, Dec. 1998.

[23] K. Mowery, S. Keelveedhi, and H. Shacham. Are AES x86 cache timing attacks still feasible? In *Proceedings of the 2012 ACM Cloud Computing Security Workshop*, CCSW '12, pages 19–24, New York, NY, USA, 2012. ACM.

[24] 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*, pages 147–162, 2007.

[25] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20, 2006.

[26] D. Page. Partitioned cache architecture as a side-channel defense mechanism. `http://eprint.iacr.org/2005/280`, 2005.

[27] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCon 2005*, 2005.

[28] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Cloud Computing Security Workshop*, pages 77–84, 2009.

[29] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

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

[31] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(2):37–71, Jan. 2010.

[32] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop*, pages 41–46, 2011.

[33] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Computer Security Applications Conference*, pages 473–482, 2006.

[34] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 494–505, 2007.

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

[36] Y. Yarom and K. Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. `http://eprint.iacr.org/2013/448`, 2013.

[37] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 313–328, May 2011.

[38] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 305–316, 2012.