Noname manuscript No.
(will be inserted by the editor)

A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware

Qian Ge · Yuval Yarom · David Cock · Gernot Heiser

Received: date / Accepted: date

Abstract Microarchitectural timing channels expose hidden hardware states though timing. We survey recent attacks that exploit microarchitectural features in shared hardware, especially as they are relevant for cloud computing. We classify types of attacks according to a taxonomy of the shared resources leveraged for such attacks. Moreover, we take a detailed look at attacks used against shared caches. We survey existing countermeasures. We finally discuss trends in attacks, challenges to combating them, and future directions, especially with respect to hardware support.

Keywords Microarchitectural timing attacks · Cache-based timing attacks · Countermeasures · Trend in the attacks

1 Introduction

Computers are increasingly handling sensitive data (banking, voting), while at the same time we consolidate more services, sensitive or not, on a single hardware platform. This trend is driven by both cost savings and convenience. The most visible example is the cloud computing—infrastructure-as-a-service (IaaS) cloud computing supports multiple virtual machines (VMs) on a hardware platform managed by a virtual machine monitor (VMM) or hypervisor. These co-resident VMs are mutually distrusting: high-performance computing software may run alongside a data centre for financial markets, requiring the platform to ensure confidentiality and integrity of VMs. The cloud computing platform also has availability requirements: service providers have service-level agreements (SLAs) which specify availability targets, and malicious VMs being able to deny service to a co-resident VM could be costly to the service provider.

The last decade has seen significant progress in achieving such isolation. Specifically, the seL4 microkernel [93] has a proof of binary-level functional correctness, as well as proofs of (spatial) confidentiality, availability and integrity enforcement [94]. Thus, it seems that the goals of strong isolation are now achievable, although this has not yet made its way into mainstream hypervisors.

However, even the strong assurance provided by the formal proofs about information flow in seL4 remain restricted to spatial isolation and limited forms of temporal isolation via coarse-grained deterministic scheduling. While they, in principle, include covert storage channels [118], there exist other kinds of channels that they do not cover that must be addressed informally.

Stealing secrets solely through hardware means (for example, through differential power analysis [95, 135]) has an extensive treatment in the literature [15, 96]. While some systems, such as smartcard processors, are extensively hardened against physical attacks, some degree of trust in the physical environment will probably always be required [109, 114].

One area that has, until recently, received comparatively little attention is leakage at the interface between software and hardware. While hardware manufacturers have generally succeeded in hiding the internal CPU architecture from programmers, at least for *functional* behaviour, its *timing*

Qian Ge
Data61, CSIRO and UNSW, Australia,
E-mail: qian.ge@data61.csiro.au

Yuval Yarom
Data61, CSIRO and The University of Adelaide, Australia,
E-mail: yval@cs.adelaide.edu.au

David Cock
Data61, CSIRO and UNSW, Australia, Present address: Systems Group, Department of Computer Science, ETH Zürich,
E-mail: david.cock@inf.ethz.ch

Gernot Heiser
Data61, CSIRO and UNSW, Australia,
E-mail: gernot.heiser@data61.csiro.au

behaviour is highly visible. Over the past decade, such *microarchitectural timing channels* have received more and more attention, from the recognition that they can be used to attack cryptographic algorithms [25, 124, 129], to demonstrated attacks between virtual machines [107, 133, 169, 180, 181]. Some recommendations from the literature have already been applied by cloud providers, e.g. Microsoft's Azure disables Hyperthreading in their cloud offerings due to the threat of introducing unpredictability and side-channel attacks [110].

We argue that microarchitectural timing channels are a sufficiently distinct, and important, phenomenon to warrant investigation in their own right. We therefore summarise all microarchitectural attacks known to date, alongside existing mitigation strategies, and develop a taxonomy based on both the degree of hardware sharing and concurrency involved. From this, we draw insights regarding our current state of knowledge, predict likely future avenues of attack and suggest fruitful directions for mitigation development.

Acıiçmez and Koç [3] summarised the microarchitectural attacks between threads on a single core known at the time. This field is moving quickly, and many more attacks have since been published, also exploiting resources shared *between* cores and across packages. Given the prevalence of cloud computing, and the more recently demonstrated cross-VM attacks [83, 107, 169], we include in the scope of this survey all levels of the system sharing hierarchy, including the system interconnect, and other resources shared between cores (and even packages).

We also include denial-of-service (DoS) attacks, as this is an active concern in co-tenanted systems, and mechanisms introduced to ensure quality of service (QoS) for clients by partitioning hardware between VMs will likely also be useful in eliminating side and covert channels due to sharing. While we include some theoretical work of obvious applicability, we focus principally on *practical* and above all *demonstrated* attacks and countermeasures.

1.1 Structure of this paper

Section 2 presents a brief introduction to timing channels, and the architectural features that lead to information leakage. Section 3 presents the criteria we use to arrange the attacks and countermeasures discussed in the remainder of the paper. Section 4 covers all attacks published to date, while Section 5 does the same for countermeasures. In Section 6, we apply our taxonomy to summarise progress to date in the field and to suggest likely new avenues for both attack and defence.

2 Background

2.1 Types of channels

Interest in information leakage and secure data processing was historically centred on sensitive, and particularly cryptographic, military and government systems [44], although many, including Lampson [100], recognised the problems faced by the tenant of an untrusted commercial computing platform, of the sort that is now commonplace. The US government's "Orange Book" standard [45], for example, collected requirements for systems operating at various security classification levels. This document introduced standards for information leakage, in the form of limits on channel bandwidth and, while these were seldom if ever actually achieved, this approach strongly influenced the direction of later work.

Leakage channels are often classified according to the threat model: *side channels* refer to the *accidental* leakage of sensitive data (for example an encryption key) by a trusted party, while *covert channels* are those exploited by a Trojan to *deliberately* leak information. Covert channels are only of interest for systems that do not (or cannot) trust internal components, and thus have highly restrictive information-flow policies (such as multilevel secure systems). In general-purpose systems, we are therefore mostly concerned with side channels.

Channels are also typically classified as either *storage* or *timing* channels. In contrast to storage channels, timing channels are exploited through timing variation [134]. While Wray [163] argued convincingly that the distinction is fundamentally arbitrary, it nonetheless remains useful in practice. In current usage, storage channels are those that exploit some *functional* aspect of the system—something that is directly visible in software, such as a register value or the return value of a system call. Timing channels, in contrast, can occur even when the functional behaviour of the system is completely understood, and even, as in the case of seL4, with a formal proof of the absence of storage channels [118]. The precise temporal behaviour of a system is rarely, if ever, formally specified.

In this work, we are concerned with *microarchitectural timing channels*.

Some previous publications classify cryptanalytic cache-based side-channel attacks as *time driven*, *trace driven*, or *access driven*, based on the type of information the attacker learns about a victim cipher [3, 121]. In trace-driven attacks, the attacker learns the outcome of each of the victim's memory accesses in terms of cache hits and misses [2, 56, 126]. Due to the difficulty of extracting the trace of cache hits and misses in software, trace-driven attacks are mostly applied in hardware and are thus outside the scope of this work.

Time-driven attacks, like trace-driven attacks, observe cache hits and misses. However, rather than generating a trace, the attacks observe the aggregate number of cache hits and misses, typically through an indirect measurement of the total execution time of the victim. Examples of such attacks include the Bernstein attack on AES [25] and the EVICT+TIME attack of Osvik et al. [125].

Access-driven attacks, such as PRIME+PROBE [125, 129], observe partial information on the addresses the victim accesses.

This taxonomy is very useful for analysing the power of an attacker with respect to a specific cipher implementation. However, it provides little insight on the attack mechanisms and their applicability in various system configurations. As such, this taxonomy is mostly orthogonal to ours.

2.2 Relevant microarchitecture

The (instruction set) *architecture* (ISA) of a processor is a functional specification of its programming interface. It abstracts over functionally irrelevant implementation details, such as pipelines, issue slots and caches, which constitute the *microarchitecture*. A particular ISA, such as the ubiquitous x86 architecture, can have multiple, successive or concurrent, microarchitectural implementations.

While functionally transparent, the microarchitecture contains hidden state, which can be observed in the timing of program execution, typically as a result of contention for resources in the hidden state. Such contention can be between processes (*external contention*) or within a process (*internal contention*).

2.2.1 Caches

Caches are an essential feature of a modern architecture, as the clock rate of processors and the latency of memory have diverged dramatically in the course of the last three decades. A small quantity of fast (but expensive) memory effectively hides the latency of large and slow (but cheap) main memory. Cache effectiveness relies critically on the *hit rate*, the fraction of requests that are satisfied from the cache. Due to the large difference in latency, a small decrease in hit rate leads to a much larger decrease in performance. Cache-based side channel attacks exploit this variation to detect contention for space in the cache, both between processes (as in a PRIME+PROBE attack, see Section 4.1.1) and within a process (Section 4.3.2).

A few internal details are needed to understand such attacks:

Cache lines In order to exploit spatial locality, to cluster memory operations and to limit implementation complexity, caches are divided into *lines*. A cache line holds one aligned, power-of-two-sized block of adjacent bytes loaded from memory. If any byte needs to be replaced (*evicted* to make room for another), the entire line is reloaded.

Associativity Cache design is a trade-off between complexity (and hence speed) and the rate of *conflict misses*. Ideally, any memory location could be placed in any cache line, and an n-line cache could thus hold *any* n lines from memory. Such a *fully associative* cache is ideal in theory, as it can always be used to its full capacity—misses occur only when there is no free space in the cache, i.e. all misses are *capacity misses*. However, this cache architecture requires that all lines are matched in parallel to check for a hit, which increases complexity and energy consumption and limits speed. Fully associative designs are therefore limited to small, local caches, such as translation look-aside buffers (TLBs).

The other extreme is the *direct-mapped* cache, where each memory location can be held by exactly one cache line, determined by the *cache index* function, typically a consecutive string of bits taken from the address, although more recent systems employ a more complex hash function [74, 112, 170]. Two memory locations that map to the same line cannot be cached at the same time—loading one will evict the other, resulting in a *conflict miss* even though the cache may have unused lines.

In practice, a compromised design—the *set-associative* cache, is usually employed. Here, the cache is divided into small *sets* (usually of between 2 and 24 lines), within which addresses are matched in parallel, as for a fully associative cache. Which set an address maps to is calculated as for a direct-mapped cache—as a function of its address (again, usually just a string of consecutive bits). A cache with n-line sets is called n-way associative. Note that the direct-mapped and fully associative caches are special cases, with 1-way and N-way associativity, respectively (where N is the number of lines in the cache).

For both the direct-mapped and set-associative caches, the predictable map from address to line is exploited. In attacks, it is used to infer cache sets accessed by an algorithm under attack (Section 4.1.1). In cache colouring (Section 5.5.4), it is exploited to ensure that an attacker and its potential victim never share sets, thus cannot conflict.

Cache hierarchy As CPU cycle times and RAM latencies have continued to diverge, architects have used a growing number of hierarchical caches—each lower in the hierarchy being larger and slower than the one above. The size of each level is carefully chosen to balance service time to the next highest (faster) level, with the need to maintain hit rate.

Flushing caches is a simple software method for mitigating cache-based channels, measured by direct and indirect cost. The direct cost contains writing back dirty cache lines and invalidating all cache lines. The indirect cost represents the performance degradation of a program due to starting with cold caches after every context switch. Therefore, the cache level that an attack targets is important when considering countermeasures: as the L1 is relatively small, it can be flushed on infrequent events, such as timesliced VM switches, with minimal performance penalty. Flushing the larger L2 or L3 caches tends to be expensive (in terms of indirect cost), but they might be protected by other means (e.g. colouring, Section 5.5.4). This hierarchy becomes more prominent on multiprocessor and multicore machines—see Section 2.2.2.

Virtual vs. physical addressing Caches may be indexed by either the virtual or the physical addresses. Virtual address indexing is generally only used in L1 caches, where the virtual-to-physical translation is not yet available, while physically indexed caches are used at the outer levels. This distinction is important for cache colouring (Section 5.5.4), as virtual address assignment is not under operating system (OS) control, and thus, virtually indexed caches cannot be coloured.

Architects employ more than just data and instruction caches. The TLB, for example, caches virtual-to-physical address translations, avoiding an expensive table walk on every memory access. The branch target buffer (BTB) holds the targets of recently taken conditional branches, to improve the accuracy of branch prediction. The trace cache on the Intel Netburst (Pentium 4) microarchitecture was used to hold the micro-ops corresponding to previously decoded instructions. All these hardware features are forms of shared caches indexed by virtual addresses. All are vulnerable to exploitation—by exploiting either internal or external contention.

2.2.2 Multiprocessor and multicore

Traditional symmetric multiprocessing (SMP) and modern multicore systems complicate the pattern of resource sharing and allow both new attacks and new defensive strategies. Resources are shared hierarchically between cores in the same package (which may be further partitioned into either groups or pseudo-cores, as in the AMD Bulldozer architecture) and between packages. This sharing allows real-time probing of a shared resource (e.g. a cache) and also makes it possible to isolate mutually distrusting applications by using pairs of cores with minimal shared hardware (e.g. in separate packages).

The cache hierarchy is matched to the layout of cores. In a contemporary, server-style multiprocessor system with, for example, 4 packages (processors) and 8 cores per package, there would usually be 3 levels of cache: private (per-core) L1 caches (split into data and instruction caches), unified L2 that is core private (e.g. in the Intel Core i7 2600) or shared between a small cluster of cores (e.g. 2 in the AMD FX-8150, Bulldozer architecture), and one large L3 per package, which may be non-uniformly partitioned between cores, as in the Intel Sandy Bridge and later microarchitectures.

2.2.3 Memory controllers

Contemporary server-class multiprocessor systems almost universally feature non-uniform memory access (NUMA). There are typically between one and four memory controllers per package, with those in remote packages accessible over the system interconnect. Memory bandwidth is always under-provisioned: the total load/store bandwidth that can be generated by all cores is greater than that available. Thus, a subset of the cores, working together, can saturate the system memory controllers, filling internal buses, therefore dramatically increasing the latency of memory loads and stores for all processes [117].

Besides potential use as a covert channel, this allows a DoS attack between untrusting processes (or VMs) and needs to be addressed to ensure QoS. The same effect is visible at other levels: the cores in a package can easily saturate that package's last-level cache (LLC); although, in this case, the effect is localised per package.

2.2.4 Buses and interconnects

Historically, systems used a single bus to connect CPUs, memory and device buses (e.g. PCI). This bus could easily be saturated by a single bus agent, effecting a DoS for other users (Section 4.6.1). Contemporary interconnects are more sophisticated, generally forming a network among components on a chip, called networks-on-chip. These network components are packet buffers, crossbar switches, and individual ports and channels. The interconnect networks are still vulnerable to saturation [159], being under-provisioned, and to new attacks, such as routing-table hijacking [138]. Contention for memory controllers has so far only been exploited for DoS (Section 4.5), but exploiting them as covert channels would not be hard.

2.2.5 Hardware multithreading

The advent of simultaneous multithreading (SMT), such as Intel's Hyperthreading, exposed resource contention at a finer level than ever before. In SMT, multiple execution contexts (threads) are active at once, with their own private state, such as register sets. This can increase utilisation, by allowing a second thread to use whatever execution resources are left idle by the first. The number of concurrent contexts ranges from 2 (in Intel's Hyperthreading) to 8 (in IBM's POWER8). In SMT, instructions from multiple threads are issued together and compete for access to the CPU's functional units.

A closely related technique, temporal multithreading (TMT), instead executes only one thread at a time, but rapidly switches between them, either on a fixed schedule (e.g. Sun/Oracle's T1 and later) or on a high-latency event (an L3 cache miss on Intel's Montecito Itanium); later Itanium processors switched from TMT to SMT.

Both hardware thread systems (SMT and TMT) expose contention *within* the execution core. In SMT, the threads effectively compete in real time for access to functional units, the L1 cache, and speculation resources (such as the BTB). This is similar to the real-time sharing that occurs between separate cores, but includes *all* levels of the architecture. In contrast, TMT is effectively a very fast version of the OS's context switch, and attackers rely on persistent state effects (such as cache pollution). The fine-grained nature though should allow events with a much shorter lifetime, such as pipeline bubbles, to be observed. SMT has been exploited in known attacks (Sections 4.2.1 and 4.3.1), while TMT, so far, has not. (Presumably because it is not widespread in contemporary processors.)

2.2.6 *Pipelines*

Instructions are rarely completed in a single CPU cycle. Instead, they are broken into a series of operations, which may be handled by a number of different functional units (e.g. the arithmetic logic unit, or the load/store unit). The very fine-grained sharing imposed by hardware multithreading (Section 2.2.5) makes pipeline effects (such as stalls) and contention on non-uniformly distributed resources (such as floating-point-capable pipelines) visible. This has been exploited (Section 4.2.1).

2.2.7 *The sharing hierarchy*

Figure 1 gathers various of the contended resources we have described, grouped by the degree of sharing, for a representative contemporary system (e.g. Intel Nehalem). The lowest layer, labelled *system shared*, contains the interconnect (Section 2.2.4), which is shared between all processes in the system, while the highest layer contains the BTB (Section 2.2.1), and the pipelines and functional units (Section 2.2.6); these are *thread shared*—only shared between hardware threads (Section 2.2.5) on the same core. The intermediate layers are more broadly shared, the lower they are: *core-shared* resources, such as the L1/L2 caches and TLB; *package-shared* resources, the L3 cache; and the *NUMA-shared* memory controllers (per package, but globally accessed, Section 2.2.3).

In Table 1 we survey published attacks at each level. Those at higher levels tend to achieve higher severity by

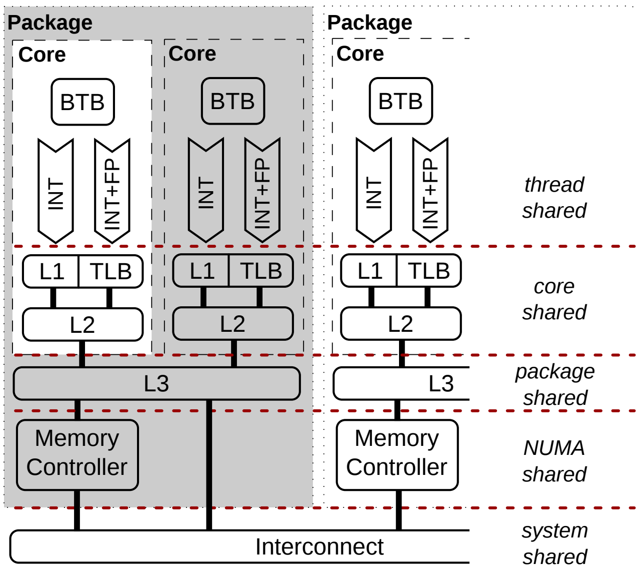

Fig. 1 Contended resources in a hierarchical multicore system.

exploiting the higher-precision information available, while those at lower levels (e.g. the interconnect) are cruder, mostly being DoS. Simultaneously, thanks to the smaller degree of sharing, the thread-shared resources are the easiest to protect (by disabling Hyperthreading (Section 5.5.1), for example).

2.3 Time and exploitation

Exploiting a timing channel naturally requires some way of measuring the passage of time. This can be achieved with access to real wall-clock time, but any monotonically increasing counter can be used. A timing channel can be viewed, abstractly, as a pair of clocks [163]. If the relative rate of these two clocks varies in a way that depends on supposedly hidden state, a timing channel exists. These clocks need not be actual time sources; they are simply sequences of events, e.g. real clock ticks, network packets, CPU instructions being retired.

In theory, it should be possible to prevent the exploitation of any timing channel, without actually eliminating the underlying contention, by ensuring that all clocks visible to a process are either synchronised (as in *deterministic execution* and *instruction-based scheduling*, see Section 5.3.1), or sufficiently coarse (or noisy) that they are of no use for precise measurements (as in *fuzzy time*, see Section 5.2). In practice, this approach is very restrictive and makes it very difficult to incorporate real-world interactions, including networking.

3 Taxonomy

We arrange the attacks and countermeasures that we will explore in the next two sections along two axes: Firstly, according to the sharing levels indicated in Figure 1, and secondly by the degree of concurrency used to exploit the vulnerability.

As we shall see, the types of attack possible vary with the level of sharing. At the top level, the attacker has very fine-grained access to the victim's state, while at the level of the bus, nothing more than detecting throughput variation is possible. The degree of concurrency, either full concurrency (i.e. multicore), timesliced execution on a single core, or hardware threading (e.g. SMT, which is intermediate), also varies predictably with level: the lower-level attacks require true concurrency, as the bus has no real persistent state. Likewise, the top-level (fine-grained) attacks also require a high degree of concurrency and are usually more effective under SMT than time slicing, as the components involved (L1, TLB, BTB) have relatively small states that are rapidly overwritten. The intermediate levels (those targeting the L3 cache, for example) are exploitable at a relatively coarse granularity, thanks to the much larger persistent state. The attacks are presented in Table 1. Table 2 lists cache-based side-channel attacks due to resource contention between processes or VMs in more detail, as they form the largest class of known attacks.

Table 3 presents the known countermeasures, arranged in the same taxonomy. The shaded cells indicate where known attacks exist, giving a visual impression of the coverage of known countermeasures against known attacks. The type of countermeasure possible naturally varies with the physical and temporal sharing level. Table 1 and Table 3 link to the sections in the text where the relevant attacks or countermeasures are discussed in detail. In Table 4, we list the known countermeasures that are deployed on current systems, based on published information. However, our list may not be complete, as product details may not be fully disclosed. We note that developers of cryptographic packages usually provide patches for published attacks.

4 Attacks

This section presents all known microarchitectural timing attacks. Section 4.1 summarises the common strategies used, while Section 4.2 through Section 4.6 list attacks, arranged first by sharing level (*thread shared* through *system shared*), and then by degree of temporal concurrency used in the reported exploit, from truly concurrent attacks exploiting multicores, through hardware-thread-based attacks, to those only requiring coarse interleaving (e.g. pre-emptive multitasking). Refer to Table 1 for an overview.

4.1 Attack types

4.1.1 Exploiting caches

*Prime and probe (*PRIME+PROBE*)* This technique detects the eviction of the attacker's working set by the victim: The attacker first *primes* the cache by filling one or more sets with its own lines. Once the victim has executed, the attacker *probes* by timing accesses to its previously loaded lines, to see if any were evicted. If so, the victim must have touched an address that maps to the same set.

*Flush and reload (*FLUSH+RELOAD*)* This is the inverse of prime and probe and relies on the existence of shared virtual memory (such as shared libraries or page de-duplication) and the ability to flush by virtual address. The attacker first *flushes* a shared line of interest (by using dedicated instructions or by eviction through contention). Once the victim has executed, the attacker then *reloads* the evicted line by touching it, measuring the time taken. A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it did not. On x86, the two steps of the attack can be combined by measuring timing variations of the `clflush` instruction. The advantage of FLUSH+RELOAD over PRIME+PROBE is that the attacker can target a specific line, rather than just a cache set.

*Evict and time (*EVICT+TIME*)* This approach uses the targeted eviction of lines, together with overall execution time measurement. The attacker first causes the victim to run, preloading its working set and establishing a baseline execution time. The attacker then *evicts* a line of interest and runs the victim again. A variation in execution *time* indicates that the line of interest was accessed.

Covert channel techniques To use the cache as a *covert channel*, the parties employ a mutual PRIME+PROBE attack. The sender encodes a bit by accessing some number of cache sets. The receiver decodes the message by measuring the time to access its own colliding lines. Covert channels have been demonstrated at all cache levels.

Denial of service When a lower-level cache is shared by multiple cores, a malicious thread can compete for space in real time with those on other cores. Doing so also saturates the lower-level cache's shared bus. The performance degradation suffered by victim threads is worse than that in a time-multiplexed attack [162]. If the cache architecture is inclusive, evicting contents from a lower-level cache also invalidates higher-level caches on other cores.

Table 1 Known microarchitectural timing attacks. S, C, and D denote side channel, covert channel, and DoS attacks.

	Multicore	Hardware threading	Time slicing
Thread shared		Section 4.2.1 – Multiplier **S, C** [9, 157] – BTB **S** [5, 6] – Trace Cache **D** [62] – Cache Banks **S** [171]	Section 4.2.2 – FPU **S, C** [16, 73] – BP **S, C** [4, 41] – BTB **S** [52] – RSB **S** [37]
Core shared		Section 4.3.1 – L1(I) **S** [1, 7] – L1(D) **S, C** [35, 125, 129, 143]	Section 4.3.2 – L1(I) **S** [1, 8, 180] – L1(D) **S, C** [10, 25, 30, 73, 82, 120, 125, 143–145, 150, 160, 161] – TLB **S** [74]
Package shared	Section 4.4.1 – Bus **C, D** [38, 162] – LLC **S, C, D** [12, 24, 38, 63, 65, 76, 85, 105, 107, 113, 147, 162, 165, 167–169, 177, 181]		Section 4.4.2 – LLC **S, C** [57, 64, 71, 73, 74, 81, 83, 89, 123, 129, 133, 167]
NUMA hared	Section 4.5.1 – Memory controller **D** [117, 177] – DRAM row buffer **C, S** [130]		Section 4.5.2 – Intel TSX **S** [87]
System shared	Section 4.6.1 – Bus **C, S, D** [72, 155, 162, 165, 177] – Processor-Interconnect **C, D** [86, 138] – PCI **D** [132] – IPI **D** [180]		

Unmasking the victim's layout In order to perform a cache-contention-based attack, the attacker needs to be able to collide in the cache with the victim. In order to know which parts of its own address spaces will do so, it needs to have some knowledge about the virtual and/or physical layout of the victim. This was historically a simple task for the virtual address space, as this was generally set at compile time, and highly predictable. The introduction of address space layout randomisation (ASLR, initially to make code-injection attacks harder to achieve) made this much harder. Hund et al. [74] demonstrated that contention in the TLB can be exploited to expose both the kernel's and the victim's address space layout.

4.1.2 Exploiting real-time contention

Real-time attacks exploit resource contention by maliciously increasing their consumption of a shared resource. For instance, adversaries can exhaust shared bus bandwidth, with a large number of memory requests [155]. Other currently running entities suffer serious performance effects: a DoS attack.

4.2 Thread-shared state

4.2.1 Hardware threads

Contention between hardware threads on the multiplier unit is detectable and exploitable. Wang and Lee [157] demonstrated its use to implement a covert channel, and Acıiçmez and Seifert [9] further showed that it in fact forms a side channel, allowing a malicious thread to distinguish multiplications from squarings in OpenSSL's RSA implementation. These attacks both measure the delay caused by competing threads being forced to wait for access to functional units (here the multiplier).

Acıiçmez et al. [5] presented two attacks that exploit contention on the BTB (Section 2.2.1). The first is an

Table 2 Timing side-channel attacks due to cache contention between hardware threads, processes or VMs. P, E, and F denote PRIME+PROBE, EVICT+TIME and FLUSH+RELOAD attacks.

Attack	Method	Cache	Target	Experimental system	Features leveraged
[129]	P	L1-D	RSA (OpenSSL 0.9.7c)	Pentium 4	SMT
[120]	P	L1-D	AES (using a separate lookup table in the last round)	Not specified in the paper	Pre-empting the AES process in short intervals
[125, 143]	P	L1-D	AES (OpenSSL 0.9.8)	Pentium 4E	SMT
[125, 143]	P	L1-D	AES (OpenSSL 0.9.8, Linux 2.6.11 dm-crypt)	Athlon 64, Pentium 4E	Virtual and physical addresses of lookup tables
[125, 143]	E	L1-D	AES (OpenSSL 0.9.8)	Athlon 64	Virtual and physical addresses of lookup tables
[1]	P	L1-I	RSA (OpenSSL 0.9.8d)	Proof of concept	SMT or pre-empting the RSA process in short intervals
[8]	P	L1-I	RSA (OpenSSL 0.9.8e)	Simulation	Embedding spy process into the RSA process, calling spy routine with a certain frequency
[35]	P	L1-D	ECDSA (OpenSSL 0.9.8k)	Pentium 4, Atom	SMT
[133]	P	LLC	Detecting co-residency, estimating traffic rates, keystroke timing attack	Amazon EC2	-
[7]	P	L1-I	DSA (OpenSSL 0.9.8d)	Intel Atom	SMT
[71]	F	LLC	AES (OpenSSL 0.9.8n)	Pentium M, Linux 2.6.33.4	Shared library, completely fair scheduler in Linux
[180]	P	L1-I	ElGamal (GnuPG v.2.0.19, libgcrypt v.1.5.0)	Core 2 Q9650, Xen 4.0	Inter-processor interrupts deliveries in Xen's credit scheduler
[74]	E	LLC	Kernel address space layout (Windows 7 Enterprise, Ubuntu Desktop 11.10)	Intel i7-870, Intel i7-950, Intel i7-2600, AMD Athlon II X3 455, VMWare Player 4.0.2 on Intel i7-870	Large page or physical address of eviction buffer
[169]	F	LLC	RSA (GnuPG 1.4.13)	Intel Core i5-3470 (Ivy Bridge), Intel Xeon E5-2430, VMware ESXi5.1, KVM	Memory mapping or page de-duplication
[81]	F	LLC	AES (OpenSSL 1.0.1f)	Intel i5-3320M, VMware ESXi5.5.0	Page de-duplication
[12, 24, 147, 168]	F	LLC	ECDSA (OpenSSL 1.0.1e)	Intel Core i5-3470	Memory mapping
[181]	F	LLC	Activities of co-residential victim VMs (e.g. number of items in user's shopping cart)	Platform-as-a-Service Cloud (DotCloud)	Shared libraries
[85]	F	LLC	TLS, DTLS (PolarSSL 1.3.6, CyaSSL 3.0.0, GnuTLS 3.2.0)	Intel i5-650, VMware ESXi5.5.0	Page de-duplication
[63]	F	LLC	Keystroke, AES (OpenSSL 1.0.2)	Windows, Linux	Shared libraries
[107]	P	LLC	ElGamal (GnuPG 1.4.13 and 1.4.18)	Xen 4.4 (Intel Xeon E5 2690), VMware ESXi 5.1 (Intel Core i5-3470)	Huge page
[83]	P	LLC	AES (OpenSSL 1.0.1f)	Intel i5-650, Xen 4.1 VMware ESXi5.5	Huge page
[123]	P	LLC	User behaviours	Firefox 34 running on Ubuntu 14.01 inside VMWare Fusion 7.1.0	High-resolution time and typed array in HTML5
[76]	P	LLC	Cloud co-location, ElGamal (GnuPG 1.4.18)	Amazon EC2	Huge page
[105]	F	LLC	User behaviours	Android on Krait-400 and ARM Cortex-A53	Shared libraries
[105]	P	LLC	AES (Bouncy Castle)	Android on ARM Cortex-A53	-
[89]	P	LLC	AES	Intel Sandy Bridge	Completely fair scheduler
[57]	F	LLC	DSA in OpenSSL	Intel Haswell (Core i5-4570)	Shared libraries

EVICT+TIME attack (Section 4.1.1), that selectively evicts entries from the BTB by executing branches at appropriate addresses and then observes the effect on the execution time of an RSA encryption (in OpenSSL 0.9.7e). The second attack improves on the first by also measuring the time required to perform the initial eviction (thus inferring whether OpenSSL had previously executed the branch), thereby reducing the number of observations required from 10^7 to 10^4. Shortly thereafter, the same authors presented a further improved attack, *simple branch prediction analysis* (SBPA), that retrieved most key bits in a single RSA operation [6]. All three attacks rely on the fine-grained sharing involved in SMT.

Yarom et al. [171] presented CacheBleed, a side channel attack that exploits cache-bank collisions [54, 77] to create measurable timing variations. The attack is able to identify the cache bank that stores each of the multipliers used during the exponentiation in the OpenSSL "constant time" RSA implementation [34, 68], allowing a complete private key recovery after observing 16,000 decryptions.

Handling exceptions on behalf of one thread can also cause performance degradation for another SMT thread, as the pipeline is flushed during handling. Further, on the Intel Pentium 4, self-modifying code flushes the trace cache (see Section 2.2.1), reducing performance by 95%. Both of these DoS attacks were demonstrated by Grunwald and Ghiasi [62].

4.2.2 Time slicing

Saving and restoring CPU state on a context switch is not free. A common optimisation is to note that most processes do not use the floating-point unit (FPU), which has a large internal state, and save and restore it only when necessary. When switching, the OS simply disables access to the FPU and performs the costly context switch if and when the process causes an exception by attempting a floating-point operation. The latency of the first such operation after a context switch thus varies depending on whether any other process is using the FPU (if it was never saved, it need not be restored). Hu [73] demonstrated that this constitutes a covert channel.

Andrysco et al. [16] demonstrated that the timings of floating-point operations vary wildly, by benchmarking each combination of instructions and operands. In particular, multiplying or dividing with subnormal values causes slowdown on all tested Intel and AMD processors, whether using single instruction multiple data (SIMD) or x87 instructions [79]. With such measurable effects, they implemented a timing attack on a scalable vector graphics (SVG) filter, which reads arbitrary pixels from any victim web page though the Firefox browser (from version 23 though 27).

Acıiçmez et al. [4] argued that the previously published SBPA attack [6] should work without SMT, instead using only pre-emptive multitasking. This has not been demonstrated in practice, however.

Cock et al. [41] discovered that the reported cycle counter value on the ARM Cortex A8 (AM3358) varies with branch mis-predictions in a separate process. This forms a covert channel and might be exploitable as a side channel by using SBPA attack.

Evtyushkin et al. [52] implemented a side channel attack on BTB collisions, which can find both kernel-level and user-level virtual address space layout on ASLR enabled Linux platforms. Firstly, they found that different virtual addresses in the same address space can create a BTB collision, because Haswell platforms use only part of the virtual address bit as hash tags. Secondly, they found that the same virtual address from two different address spaces can also create a BTB collision.

Bulygin [37] demonstrated a side channel attack on the return stack buffer (RSB). The RSB is a small rolling cache of function-call addresses used to predict the return address of return statements. By counting the number of RSB entries, a victim replaces the attack can detect the case of an end reduction in the Montgomery modular multiplication [115] which leads to breaking RSA [36]. Bulygin [37] also showed how to use the RSB side channel to detect whether software executes on an hypervisor.

4.3 Core-shared state

4.3.1 Hardware threads

A number of attacks have exploited hardware threading to probe a competing thread's L1 cache usage in real time. Percival [129] demonstrated that contention on both the L1 (data) and L2 caches of a Pentium 4 employing Hyperthreading can be exploited to construct a covert channel, and also as a side channel, to distinguish between square and multiply steps and identify the multipliers used in OpenSSL 0.9.7c's implementation of RSA, thus leaking the key. Similarly, Osvik et al. [125] and Tromer et al. [143] attacked AES in OpenSSL 0.9.8 with PRIME+PROBE on the L1 data-cache (D-cache). Acıiçmez [1] used L1 instruction-cache (I-cache) contention to determine the victim's control flow, distinguishing squares and multiplies in OpenSSL 0.9.8d RSA. Brumley and Hakala [35] used a simulation (based on empirical channel measurements) to show that an attack against elliptic-curve cryptography (ECC) in OpenSSL 0.9.8k was likely to be practical. Acıiçmez et al. [7] extended their previous attack [1] to digital signature algorithm (DSA) in OpenSSL 0.9.8l, again exploiting I-cache contention, this time on Intel's Atom processor employing Hyperthreading.

4.3.2 Time slicing

As part of the evaluation of the VAX/VMM security operating system, Hu [73] noted that sharing caches would lead to channels. Tsunoo et al. [144] demonstrated that timing variations due to *self-contention* can be used to cryptanalyse MISTY1. The same attack was later applied to DES [145].

Bernstein [25] showed that timing variations due to self-contention can be exploited for mounting a practical remote attack against AES. The attack was conducted on Intel Pentium III. Bonneau and Mironov [30] and Acıiçmez et al. [10] both further explored these self-contention channels in AES in OpenSSL (versions 0.9.8a and 0.9.7e). Weiß et al. [160] demonstrated that it still works on the ARM Cortex A8. Irazoqui et al. [82] demonstrated Bernstein's correlation attack on Xen and VMware, lifting the native attack to across VMs. More recently, Weiß et al. [161] extended their previous attack [160] on the microkernel virtualisation framework, using the PikeOS microkernel [88] on ARM platform.

Neve and Seifert [120] demonstrated that SMT was not necessary to exploit these cache side channels, again successfully attacking AES's S-boxes. Similarly, PRIME+PROBE and EVICT+TIME techniques can also detect the secret key of AES through L1 D-cache, with knowledge of the virtual and physical addresses of AES's S-boxes [125, 143]. Acıiçmez [1] presented an RSA attack using the PRIME+PROBE technique on the L1 I-cache, assuming frequent preemptions on RSA processes. Acıiçmez and Schindler [8] demonstrated an RSA attack based on L1 I-cache contention. In order to facilitate the experiment, their attack embeds a spy process into the RSA process and frequently calling the spy routine. According to Neve [119], embedded spy process attains similar empirical results with standalone spy process, therefore Acıiçmez and Schindler [8] argued that their simplified attack model would be applicable in practice. Zhang et al. [180] took the SMT-free approach and showed that key-recovery attacks using L1 I-cache contention were practical between virtual machines. Using inter-processor interrupts (IPIs) to frequently displace the victim, they successfully employed a PRIME+PROBE attack to recover ElGamal keys [51].

Vateva-Gurova et al. [150] measured the signal of a cross-VM L1 cache covert channel on Xen with different scheduling parameters, including load balancing, weight, cap, timeslice and rate limiting.

Hund et al. [74] showed that TLB contention allows an attacker to defeat ASLR [29], with a variant of the FLUSH+RELOAD technique (Section 4.1.1): they exploited the fact that invalid mappings are not loaded into the TLB, and so a subsequent access to an invalid address will trigger another table walk, while a valid address will produce a much more rapid segmentation fault. Hund et al. evaluated this technique on Windows 7 Enterprise and Ubuntu Desktop 11.10 on three different Intel architectures, with a worst-case accuracy of 95% on the Intel i7-2600 (Sandy Bridge).

4.4 Package-shared state

4.4.1 Multicore

Programs concurrently executing on multicores can generate contention on the shared LLC, creating covert channels, side channels or DoS attacks. Xu et al. [167] evaluated the covert channel bandwidth based on LLC contention with PRIME+PROBE technique, improving the transmission protocol of the initial demonstration by [133] to the multicore setting. Xu et al. [167] not only improved the protocol, but also improved the bandwidth from 0.2 b/s to 3.2 b/s on Amazon EC2. Wu et al. [165] stated the challenges of establishing a reliable covert channel based on LLC contention, such as scheduling and addressing uncertainties. Despite demonstrating how these can be overcome, their cache-based covert channel is unreliable as the hypervisor frequently migrates virtual CPUs across physical cores. They invented a more reliable protocol based on bus contention (Section 4.6.1).

Yarom and Falkner [169] demonstrated a FLUSH+RELOAD attack on the LLC (Intel Core i5-3470 and Intel Xeon E5-2430) for attacking RSA, requiring read access to the in-memory RSA implementation, through either memory mapping or page de-duplication. Yarom and Benger [168], Benger et al. [24] and Van de Pol et al. [147] used similar attacks to steal ECDSA keys. Moreover, Zhang et al. [181] demonstrated that the same type of attack can steal granular information (e.g. the number of items in a user's shopping cart) from a co-resident VM on DotCloud [47], a commercial platform-as-a-service (PaaS) cloud platform. Gruss et al. [63] further generalised the FLUSH+RELOAD technique with a pre-computed cache template matrix, a technique that profiles and exploits cache-based side channels automatically. The cache template attack involves a profile phase and a exploitation phase. In the profiling phase, the attack generates a cache template matrix for the cache-hit ratio on a target address during processing of a secret information. In the exploitation phase, the attack conducts FLUSH+RELOAD or PRIME+PROBE attack for constantly monitoring cache hits and computes the similarity between collected traces and the corresponding profile from the cache template matrix. The cache template matrix attack is able to perform attacks on data and instructions accesses online, attacking both keystrokes and the T-table-based AES implementation of OpenSSL.

Because the LLC is physically indexed and tagged, constructing a prime buffer for a PRIME+PROBE attack requires knowledge of virtual-to-physical address mappings. Liu et al. [107] presented a solution that creates an eviction buffer with large (2 MiB) pages, as these occupy contiguous memory. By probing on one cache set, they demonstrated a covert channel attack with a peak throughput of 1.2 Mb/s and an error rate of 22 %. Based on the same PRIME+PROBE technique, they conducted side-channel attacks on both the square-and-multiply exponentiation algorithm in ElGamal (GnuPG 1.4.13) and the sliding-window exponentiation in ElGamal (GnuPG 1.4.18). Inci et al. [76] demonstrated that the same PRIME+PROBE technique can be used in a real cloud environment. They used the attack on the Amazon EC2 cloud to leak both cloud co-location information and ElGamal private keys (GnuPG 1.4.18). Maurice et al. [113] also demonstrated a PRIME+PROBE attack on a shared LLC, achieving covert channel throughput of 1291 b/s and 751 b/s for a virtualized set-up. The attack assumes an inclusive LLC: a sender primes the entire LLC with writes, and a receiver probes on a single set in its L1 D-cache.

Gruss et al. [63] suggested a variant of FLUSH+RELOAD that combines the eviction process of PRIME+PROBE with the reload step of FLUSH+RELOAD. This technique, called EVICT+RELOAD, is much slower and less accurate than FLUSH+RELOAD; however, it obviates the need for dedicated instructions for flushing cache lines. Lipp et al. [105] demonstrated the use of the EVICT+RELOAD attack to leak keystrokes and touch actions on Android platforms running on ARM processors.

FLUSH+FLUSH [65] is another variant of FLUSH+RELOAD, which measures variations in the execution time of the x86 `clflush` instruction to determine whether the entry was cached prior to being flushed. The advantages of the technique over FLUSH+RELOAD are that it is faster, allowing for a higher resolution, and that it generates less LLC activity which can be used for detecting cache timing attacks. FLUSH+FLUSH is, however, more noisy than FLUSH+RELOAD, resulting in a slightly higher error rate. Gruss et al. [65] demonstrated that the technique can be used to implement high-capacity covert channels, achieving a bandwidth of 496 KiB/s.

Both LLCs and internal buses are vulnerable to DoS attacks. Woo and Lee [162] showed that by causing a high rate of L1 misses, an attacker can monopolise LLC bandwidth (L2 in this case), to the exclusion of other cores. Also, by sweeping the L2 cache space, threads on other cores suffer a large number of L2 cache misses. Cardenas and Boppana [38] demonstrated the brute-force approach—aggressively polluting the LLC dramatically slows all other cores. These DoS attacks are much more effective when launched from a separate core, as in the pre-emptively scheduled case, the greatest penalty that the attacker can enforce is one cache refill per timeslice. Allan et al. [12] showed that repeatedly evicting cache lines in a tight loop of a victim program slows the victim by a factor of up to 160. They also demonstrated how slowing a victim down can improve the signal of side-channel attacks. Zhang et al. [177] conducted the *cache cleansing* DoS attack that generated up to 5.5x slowdown for program with poor memory locality and up to 4.4x slowdown to cryptographic operations. Because x86 platform implements the inclusive LLC, replacing a cache line from LLC can also evicts its copy from upper-level caches (Section 4.1.1). To build this attack, the attacker massively fetched cache lines into LLC sets, causing the victim suffering from a large number of cache misses due to cache conflicts.

4.4.2 Time slicing

The LLC maintains the footprint left by previously running threads or VMs, which is exploitable as a covert or a side channel. Hu [73] demonstrated that two processes can transmit information by interleaving on accessing the same portion of a shared cache. Although Hu did not demonstrate the attack on a hardware platform, the attack can be used on the shared LLC. Similarly, Percival [129] explored covert channels based on LLC collisions. Ristenpart et al. [133] explored the same technique on a public cloud (Amazon EC2), achieving a throughput of 0.2 b/s. Their attack demonstrated that two VMs can transmit information through a shared LLC.

The LLC contention can be used for detecting co-residency, estimating traffic rates and conducting keystroke timing attacks on the Amazon EC2 platform [133].

Gullasch et al. [71] attacked AES in OpenSSL 0.98n with a FLUSH+RELOAD attack, leveraging the design of Linux' *completely fair scheduler* to frequently pre-empt the AES thread. With the FLUSH+RELOAD technique, Irazoqui et al. [81] broke AES in OpenSSL 1.0.1f with page sharing enabled on an Intel i5-3320M running VMware ESXi5.5.0. More recently, García et al. [57] demonstrated the FLUSH+RELOAD attack on the DSA implementation in OpenSSL, the first key-recovery cache-timing attack on transport layer security (TLS) and secure shell (SSH) protocols.

Hund et al. [74] detected the kernel address space layout with the EVICT+TIME technique, measuring the effect of evicting a cache set on system-call latencies on Linux.

Gruss et al. [64] discovered that the prefetch instructions not only expose the virtual address space layout via timing but also allow prefetching kernel memory from user-space because of lack of privilege checking. Their discoveries were applicable on both x86 and ARM architectures. They demonstrated two attacks: the translation-level oracle and address-translation oracle. Firstly, the translation-level oracle measures execution time for prefetching on a randomly selected virtual address and then compares against the calibrated execution time. To translate the virtual address into a physical address, the prefetch instruction traverses multiple levels of page directories (4 levels for PML4E on x86), and terminates as soon as the correct entry is found. Therefore, the timing reveals at which level does address lookup terminates, revealing the virtual address space layout even though ASLR is enabled. Secondly, the address-translation oracle verifies if two virtual addresses p and q are mapped to the same physical address by flushing p, prefetching q, and then reloading p again. If the two addresses are mapped to the same physical address, reloading p encounters a cache hit. Also, the prefech instructions can be conducted on any virtual address, including kernel addresses, and thus can be used to locate the kernel window that contains direct physical mappings to further implement return-oriented programming attack [90].

To conduct a PRIME+PROBE attack on the LLC, Irazoqui et al. [83] additionally took advantages of large pages for probing on cache sets from the LLC without solving virtual-to-physical address mappings (Section 2.2.1). The attack monitors 4 cache sets from the LLC and recovers an AES key in less than 3 min in Xen 4.1 and less than 2 min in VMware ESXI 5.5. Oren et al. [123] implemented a PRIME+PROBE attack in JavaScript by profiling on cache conflicts with given virtual addresses. Rather than relying on large pages, they created a priming buffer with 4 KiB pages. Furthermore, the attack collects cache activity signatures of target actions by scanning cache traces, achieving covert channel bandwidth of 320 kb/s on their host machine and 8 kb/s on a virtual machine (Firefox 34 running on Ubuntu 14.01 inside VMWare Fusion 7.1.0). A side-channel version of this attack associates user behaviour (mouse and network activities) with cache access patterns. Kayaalp et al. [89] also demonstrated a probing technique that identifies LLC cache sets without relying on large pages. They used the technique in conjunction with the Gullasch et al. [71] attack on the completely fair scheduler for attacking AES.

Targeting the AES T-tables, Lipp et al. [105] showed that the ARM architecture is also vulnerable to the PRIME+PROBE attack.

4.5 NUMA-shared state

4.5.1 Multicore

Shared memory controllers in multicore systems allow DoS attacks. Moscibroda and Mutlu [117] simulated an exploit of the first-come first-served scheduling policy in memory controllers, which prioritises streaming access to impose a 2.9-fold performance hit on a program with a random access pattern. Zhang et al. [177] demonstrated a DoS attack by generating contentions on either bank or channel schedulers in memory controllers on the testing x86 platform (Dell PowerEdge R720). The attacker issued vast amount of memory accesses from a DRAM bank, causing up to 1.54x slowdown for a victim that accesses from either the same bank or a different bank in the same DRAM channel.

Modern DRAM comprises a hierarchical structure of channels, DIMMs, ranks and banks. Inside each bank, the DRAM cells are arranged as a two-dimensional array, located by rows and columns. Also, each bank has a *row buffer* that caches the currently active row. For any memory access, the DRAM firstly activates its row in the bank and then accesses the cell within that row. If the row is currently active, the request is served directly from the row buffer (a row hit). Otherwise, the DRAM closes the open row and fetches the selected row to the row buffer for access (a row miss). According to experiments conducted on both x86 and ARM platforms, a row miss leads to a higher memory latency than a row hit, which can be easily identified [130].

Pessl et al. [130] demonstrated how to use timing variances due to row buffer conflicts to reverse-engineer the DRAM addressing schemes on both x86 and ARM platforms. Based on that knowledge, they implemented a covert channel based on row buffer conflicts, which achieved a transfer rate of up to 2.1 Mb/s on the Haswell desktop platform (i7-4760) and 1.6 Mb/s on the Haswell-EP server platform (2x Xeon E5-2630 v3).

Furthermore, Pessl et al. [130] conducted a cross-CPU side channel attack by detecting row conflicts triggered by keystrokes in the Firefox address bar. In the preparation stage, a spy process allocates a physical address s that shares the same row of the target address v in a victim process. Also, she allocates another physical address q that maps to a different row in the same bank. To run the side channel attack, the spy firstly accesses q and then waits for the victim to execute before measuring the latency of accessing s. If the victim accessed v, the latency of accessing s is much less than a row conflict. Therefore, the spy can infer when the non-shared memory location is accessed by the victim.

4.5.2 Time slicing

Jang et al. [87] discovered that Intel transactional synchronisation extension (TSX) contains a security loophole that aborts a user-level transaction without informing kernel. In addition, the timing of TSX aborts on page faults reveals the attributes of corresponding translation entires (i.e., mapped, unmapped, executable or non-executable). They successfully demonstrated a timing channel attack on kernel ASLR of mainstream OSes, including Windows, Linux and OSx.

4.6 System-shared state

4.6.1 Multicore

Hu [72] noted that the system bus (systems of the time had a single, shared bus) could be used as a covert channel, by modulating the level of traffic (and hence contention). Wu et al. [165] demonstrated that this type of channel is still exploitable in modern systems, achieving a rate of 340 b/s between Amazon EC2 instances.

Bus contention is clearly also exploitable for DoS. On a machine with a single frontside bus, Woo and Lee [162] generated a DoS attack with L2 cache misses in a simulated environment. Zhang et al. [177] shown that an attacker can use atomic operations on either unaligned or non-cacheable memory blocks to lock the shared system bus on the testing x86 platform (Dell PowerEdge R720), therefore slow down the victim (up to 7.9x for low locality programs).

While many-core platforms increase performance, programs running on separated cores may suffer interference from competing on on-chip interconnections. Because networks-on-chip (Section 2.2.4) share internal resources, a program may inject network traffic by rapidly generating memory fetching requests, unbalancing the share of internal bandwidth. Wang and Suh [155] simulated both covert channels and side channels though network interference. A Trojan encodes a "1" bit through high and a "0" bit through low traffic load. A spy measures the throughput of its own memory requests. For the side channel, their simulation replaces the Trojan program with an RSA server. This side-channel attack simulated an RSA execution where every exponentiation execution suffers cache misses, resulting in memory fetches for "1 " bits in the secret key. By monitoring the network traffic, the spy program observes that the network throughput is highly related to the fraction of bits "1" in the RSA key.

Song et al. [138] discovered a security vulnerability contained in the programmable routing table in a Hypertranport-based processor-interconnect router on AMD processors [14]. With malicious modifications to the routing tables, Song et al. demonstrated degradation of both latency and bandwidth of the processor interconnect on an 8-node AMD server (the Dell PowerEdge R815 system).

Irazoqui et al. [86] demonstrated that due to the cache coherency maintained between multiple packages, variants of the FLUSH+RELOAD attack can be applied between packages. They used the technique to attack both the AES T-Tables and a square-and-multiply implementation of ElGamal.

Lower-level buses, including PCI Express, can also be exploited for DoS attacks. Richter et al. [132] showed that by saturating hardware buffers, the latency to access a Gigabit Ethernet network interface controller, from a separate VM, increases more than sixfold.

Zhang et al. [180] demonstrated that the slow handling of inter-processor interrupts (IPIs), together with their elevated priority, allows for a cross-processor DoS attack.

5 Countermeasures

5.1 Constant-time techniques

A common approach to protecting cryptographic code is to ensure that its behaviour is never data dependent: that the sequence of cache accesses or branches, for example, does not depend on either the key or the plaintext. This approach is widespread in dealing with overall execution time, as exploited in remote attacks, but is also being applied to local contention-based channels. Bernstein [25] noted the high degree of difficulty involved.

We look at the meaning of "sequence of cache acceses" as an example of the complexities involved. The question is at what resolution the sequence of accesses needs to be independent of secret data. Clearly, if the sequence of cache lines accessed depends on secret data, the program can leak information through the cache. However, some code, e.g. the "constant-time" implementation of modular exponentiation in OpenSSL, can access different memory addresses within a cache line, depending on the secret exponent. Brickell [32] suggested not having secret-dependent memory access *at coarser than cache line granularity*, hinting that such an implementation would not leak secret data. However, Osvik et al. [124] warned that processors may leak low-address-bit information, i.e. the offset within a cache line. Bernstein and Schwabe [26] demonstrated that under some circumstances this can indeed happen on Intel processors. This question has been recently resolved when Yarom et al. [171] demonstrated that the OpenSSL implementation is vulnerable to the CacheBleed attack.

Even a stronger form, where the sequence of memory accesses does not depend on secret information, may not be sufficient to prevent leaks. Coppens et al. [43] listed several possible leaks, including instructions with data-dependent execution times, register dependencies and data dependencies through memory. No implementation is yet known to be vulnerable to side-channel attacks through these leaks. Coppens et al. instead developed a compiler that automatically eliminates control-flow dependencies on secret keys of cryptographic algorithms (on x86).

To guide the design of constant-time code, several authors have presented analysis tools and formal frameworks: Langley [101] modified the Valgrind [146] program analyser to trace the flow of secret information and warn if it is used in branches or as a memory index. Köpf et al. [99] described a method for measuring an upper bound on the amount of secret information that leaks from an implementation of a cryptographic algorithm through timing variations. CacheAudit [48] extended their work to provide better abstractions and a higher precision. FlowTracker [137] modified the LLVM compiler [102] to use information flow analysis for side-channel leak detection.

Table 3 Published countermeasures

	Multicore	Hardware threading	Time slicing
Thread shared		– Disable hardware threading (Section 5.5.1) [110, 129] – Auditing (Section 5.6) [140]	– Constant-time techniques (Section 5.1) [16, 131] – Cache flushing (Section 5.4.1) [60, 149, 179] – Lattice scheduling (Section 5.4.2) [44, 73]
Core shared		– Hardware cache partition (Section 5.5.3) [42, 46, 128, 158] – RPCache (Section 5.2) [97, 158] – Random fill cache (Section 5.2) [106] – Disable Hardware Threading	– Hardware cache partition – Execution leases (Section 5.4.6) [141, 142] – Minimum timeslice (Section 5.4.3) [149] – Auditing (Section 5.6) [179] – RPCache – Random fill cache – Cache flushing – Lattice scheduling
Package shared	– Hardware cache partition – Cache colouring (Section 5.5.4) [41, 59, 92, 136] – Quasi-partitioning (Section 5.5.5) [182] – Auditing (Section 5.6) [178] – RPCache – Random fill cache		– Disable page sharing (Section 5.5.2) [152, 182] – Kernel address space isolation (Section 5.4.7) [64] – Auditing (Section 5.6) [39, 65, 178] – Hardware cache partitions – Cache colouring – Quasi-partitioning – Execution leases – RPCache – Random fill cache – Lattice scheduling
NUMA shared	– Memory controller partitioning (Section 5.4.5) [156] – Auditing (Section 5.6) [177]		– Kernel address space isolation
System shared	– Auditing (Section 5.6) [177] – Netoworks-on-chip partitioning (Section 5.4.4) [155, 159] – Spatial network partitioning (Section 5.5.6) [155] – Minimum Timeslice		

Constant-time techniques are used extensively in the design of the NaCl library [27], which is intended to avoid many of the vulnerabilities discovered in OpenSSL.

For addressing the timing channel on floating-point operations, Andrysco et al. [16] designed a fix-point constant-time math library, libfixedtimefixpoint. On the testing platform (Intel Core i7 2635QM at 2.00GHz), the library performs constant-time operations, which take longer than optimised hardware instructions. Later, Rane et al. [131] presented a compiler-based approach, by utilising SIMD lanes in x86 SSE and SSE2 instruction sets to provide fixed-time floating-point operations. Their key insight is that the latency of a SIMD instruction is determined by the longest running path in the SIMD lanes. Their compiler pairs origi-

Table 4 Countermeasures employed on current systems

Countermeasure	System
Constant-time techniques (Section 5.1)	Commonly used (at least to some extent) in many cryptographic libraries
Instruction set extension for AES (Section 5.1.1)	Intel (from Westmere), AMD (from Bulldozer), ARM (from V8-A) SPARC (from T4), and IBM (from Power7+)
PCLMULQDQ instruction (Section 5.1.1)	Intel (from Westmere)
Lattice scheduling (Section 5.4.2)	VAX/VMM security kernel, seL4
Disable hardware threading (Section 5.5.1)	A standard BIOS feature for x86-based machine. Used in Microsoft's Azure
Disable page sharing (Section 5.5.2)	VMware ESXi
Hardware cache partitions (Section 5.5.3)	Intel's Cache Allocation Technology, ARM cache lockdown through the L2 cache controller

nal floating-point operations with dummy slow-running subnormal inputs. After the SIMD operations is executed, the compiler only preserves the results of the authentic inputs. Their evaluation shown a 0–1.5% timing variation of floating-point operations with different types of inputs on the testing machine (i7-2600). This compiler-based approach introduced 32.6x overhead on SPECfp2006 benchmarks.

The main drawback of the constant-time approach is that a constant-time implementation on one hardware platform may not perform constantly on another hardware platform. For example, Cock et al. [41] demonstrated that the constant-time fix for mitigating Lucky 13 attack (a remote side-channel attack, [11]) in OpenSSL 1.0.1e still contains a side channel on the ARM AM3358 platform.

5.1.1 Hardware support

One approach to avoiding vulnerable table lookups is to provide constant-time hardware operations for important cryptographic primitives, as suggested by Page [127]. The x86 instruction set has now been extended in exactly this fashion [66, 67], which provides instruction support for AES encryption, decryption, key expansion and all modes of operations. At current stage, Intel (from Westmere) [166], AMD (from Bulldozer) [13], ARM (from V8-A) [19], SPARC (from T4) [122], and IBM (from Power7+) [172] processor architectures all support AES with instruction extensions. Furthermore, Intel introduced PCLMULQDQ instruction that is available from microarchitecture Westmere [69, 70]. With the help from the PCLMULQDQ instruction, the AES in Galois Counter Mode can be efficiently implemented using the AES instruction extension.

5.1.2 Language-based approaches

In order to control the externally observable timing channels, language-based approaches invent specialised language semantics with non-variant execution length, which require a corresponding hardware design [175]. The solutions in this category propose that timing channels should be handled at both hardware level and software level.

5.2 Injecting noise

In theory, it should be possible to prevent the exploitation of a timing channel without eliminating contention, by ensuring that the attacker's measurements contain so much noise as to be essentially useless. This is the idea behind *fuzzy time* [72], which injects noise into all events visible to a process (such as pre-emptions and interrupt delivery), as a covert channel countermeasure.

In a similar vein, Brickell et al. [33] suggested an alternative AES implementation, including compacting, randomising and preloading lookup tables. This introduced noise to the cache footprint left by AES executions, thus defending against cache-based timing attacks. As a result, the distribution of AES execution times follows a Gaussian distribution over random plaintexts. These techniques incurred 100%–120% performance overhead.

Wang and Lee [158] suggested the *random permutation cache* (RPcache), which provides a randomised cache indexing scheme and protection attributes in every cache line. Each process has a permutation table to store memory-to-cache mappings, and each cache line contains an ID representing its owner process. Cache lines with different IDs cannot evict each other. Rather, the RPcache randomly selects a cache set and evict a cache line in that set. Simulation results suggest a small (1%) performance hit in the SPEC2000 benchmark. Kong et al. [97] suggested an explicitly requested RPCache implementation, just for sensitive data such as AES tables. Simulations again suggest the potential for low overhead.

Vattikonda et al. [151] modified the Xen hypervisor to insert noise into the high-resolution time measurements in VMs by modifying the values returned by the `rdtsc` instruction. Martin et al. [111] argued that timing channels can be prevented by making internal time sources inaccurate. Consequently, they modified the implementation of the x86 `rdtsc` instruction so it stalls execution until the end of a predefined epoch and then adds a random number between zero and the size of the epoch, thus fuzzing the time counter. In addition, they proposed a hardware monitoring mechanism for detecting software clocks, for example inter-thread communication through shared variables. The paper presents statistical analysis that demonstrates timing-channel mitigation. However, the effectiveness of this solution is based on two system assumptions: that external events do not provide sufficient resolution to efficiently measure microarchitecture events and that their monitoring makes software clocks prohibitive.

Liu and Lee [106] suggested that the demand-fetch policy of a cache is a security vulnerability, which can be exploited by reuse-based attacks that leverage previously accessed data in a shared cache. To address these attacks, they designed a cache with random replacement. On a cache miss, the requested data are directly sent to processor without allocating a cache line. Instead, the mechanism allocates lines with randomised fetches within a configurable neighbourhood window of the missing memory line.

Zhang et al. [176] introduced a bystander VM for injecting noise on the cross-VM L2-cache covert channel with a configurable workload. With a continuous-time Markov process to model the PRIME+PROBE-based cache covert channel, they analysed the impact of bystander VMs on the error rate of cache-based covert channels under the scheduling algorithm used by Xen. They found that as long as the bystander VMs only adjust their CPU time consumption, they do not significantly impact cross-VM covert channel bandwidth. For effectively introducing noise into the PRIME+PROBE channel, the bystander VMs must also modulate their working sets and memory access rates.

However, noise injection is inefficient for obtaining high security [41]. Although anti-correlated "noise" can in principle completely close the channel, producing such compensation is not possible in many circumstances. The amount of actual (uncorrelated) noise required increases dramatically with decreasing channel capacity. This significantly degrades system performance and makes it infeasible to reduce channel bandwidth by more than about two orders of magnitude [41].

5.3 Enforcing determinism

As discussed in Section 2.3, it should be possible to eliminate timing channels by completely eliminating visible timing variation. There are two existing solutions that attempt this, *virtual time* (Section 5.3.1) and *black-box mitigation* (Section 5.3.2).

5.3.1 Virtual time

The virtual time approach tries to completely eliminate access to real time, providing only *virtual* clocks, whose progress is completely deterministic, and independent of the actions of vulnerable components. For example, Determinator [23] provides a deterministic execution framework for debugging concurrent programs. Aviram et al. [22] repurposed Determinator to provide a virtual time cloud computing environment. Ford [55] extended this model with queues, to allow carefully scheduled IO with the outside world, without importing real-time clocks. Wu et al. [164] further extended the model with both internal and external event sequences, producing a hypervisor-enforced deterministic execution system. It introduces the *mitigation interval*; if this is 1 ms, the information leakage rate is theoretically bounded at 1 Kb/s. They demonstrated that for the 1 ms setting, CPU-bound applications were barely impacted compared to executing on QEMU/KVM, while network-bound applications experienced throughput degradation of up to 50%.

StopWatch [103] instead runs three replicas of a system, and attempts to release externally visible events at the median of the times determined by the replicas, virtualising the x86 time-stamp counter.

Because it virtualises both internally and externally visible events, complete system-time virtualisation is effective against all types of timing channels, in principle. However, it cannot prevents cross-VM DoS attacks, as generating resource contentions will still be possible on a shared platform. The first downside is that these systems pay a heavy performance penalty. For StopWatch, latency increases 2.8 times for network-intensive benchmarks and 2.3 times for computate-intensive benchmarks [103]. Secondly, systems such as Determinator [23] rely on custom-written software and cannot easily support legacy applications.

Instruction-based scheduling (IBS) is a more limited form of deterministic execution that simply attempts to prevent the OS's pre-emption tick providing a clock source to an attacker. The idea is to use the CPU's performance counters to generate the pre-emption interrupt after a fixed number of instructions, ensuring that the attacker always completes the same amount of work in every interval. Dunlap et al. [49] proposed this as a debugging technique, and it was later incorporated into Determinator [22]. Stefan et al. [139] suggested it as an approach to eliminating timing channels. Cock et al. [41] evaluated IBS on a range of platforms and reported that the imprecise delivery of exceptions on modern CPUs negatively affects the effectiveness of this technique. In order to schedule threads after executing a precise number of instructions, the system has to configure the performance counters to fire earlier than the target instruction count, then single-steps the processor until the target is reached [50]. In Deterland [164], single-stepping adds 30% overhead to CPU-bound benchmarks for a 1-ms mitigation interval, and 5% for a 100-ms interval.

5.3.2 Black-box mitigation

Instead of trying to synchronise all observable clocks with the execution of individual processes, this approach attempts to achieve determinism for the system as a whole, by controlling the timing of *externally visible* events. It is thus only applicable to remotely exploitable attacks that depend on variation in system response time. This avoids much of the performance cost of virtual time, and the thorny issue of trying to track down all possible clocks available inside a system.

Köpf and Dürmuth [98] suggested *bucketing*, or quantising, response times, to allow an upper bound on information-theoretic leakage to be calculated. Askarov et al. [21] presented a practical approach to such a system, using exponential back-off to provide a hard upper bound on the amount that can ever be leaked. This work was later revised and extended [174]. Cock et al. [41] showed that such a policy could be efficiently implemented, using real-time scheduling on seL4 to provide the required delays, in an approach termed *scheduled delivery* (SD). SD was applied to mitigate the Lucky 13 attack on OpenSSL TLS [11], with better performance than the upstream constant-time implementation.

Braun et al. [31] suggested compiler directives that annotate fixed-time functions. For such functions, the compiler automatically generates code for temporal padding. The padding includes a timing randomisation step that masks timing leaks due to the limited temporal resolution of the padding.

5.4 Partitioning time

Attacks which rely on either concurrent or consecutive access to shared hardware can be approached by either providing timesliced exclusive access (in the first case) or carefully managing the transition between timeslices (in the second).

5.4.1 Cache flushing

The obvious solution to attacks based on persistent state effects (e.g. in caches) is to flush on switches. Zhang and Reiter [179] suggested flushing all local state, including BTB and TLB. Godfrey and Zulkernine [60] suggested flushing all levels of caches during VM switches in cloud computing, when CPU switches security domains. There is obviously a cost to be paid, however. Varadarajan et al. [149] benchmarked the effect of flushing the L1 cache: They measured an 8.4 μs direct cost, and a substantial overall cost, resulting in a 17% latency increase in the ping benchmark that issues ping command at 1-ms interval. The experimental hardware is a 6 core Intel Xeon E5645 processor.

Flushing the top-level cache is not unreasonable on a VM switch. The sizes of L1 caches are relatively small (32 KiB on x86 platforms), and the typical VM switch rates are low, e.g. the credit scheduler in Xen normally makes scheduling decisions every 30 ms [180]. Therefore, there is a low likelihood of a newly scheduled VM finding any data or instructions hot in the cache, which implies that the indirect cost (Section 2.2.1) of flushing the L1 caches on a VM switch is negligible. For the much larger lower-level caches, flushing is likely to lead to significant performance degradation.

5.4.2 Lattice scheduling

Proposed by Denning [44], and implemented in the VAX/VMM security kernel [73], lattice scheduling attempts to amortise the cost of cache flushes on a context switch, by limiting them to switches from sensitive partitions to untrusted ones. In practice, this approach is limited to systems with a hierarchical trust model, unlike the mutually distrusting world of cloud computing. Cock [40] presented a verified lattice scheduler for the domain-switched version of seL4.

5.4.3 Minimum timeslice

Some attacks, such as the PRIME+PROBE approach of Zhang et al. [180], relied on the ability to frequently inspect the victim's state by triggering pre-emptions. Enforcing a minimum timeslice for the vulnerable component, as Varadarajan et al. [149] suggested, prevents the attacker inspecting the state in the middle of a sensitive operation, at the price of increased latency. As this is a defence specific to one attack, it is likely that it can be circumvented by more sophisticated attacks.

5.4.4 Networks-on-chip partitioning

To prevent building a covert channel by competing on on-chip interconnects (Section 4.6.1), network capacity can be dynamically allocated to domains by temporally partitioning arbitrators. Wang and Suh [155] provided a priority-based solution for one-way information-flow systems, where information is only allowed to flow from lower to higher security levels. Their design assigns strict priority bounds to low-security traffic, thus ensuring non-interference [61] from high-security traffic.

Furthermore, time multiplexing shared network components constitute a straightforward scheme for preventing interference on network throughput, wherein the packets of each security domain are only propagated during time periods allocated to that domain. Therefore, the latency and throughput of each domain are independent of other domains. The main disadvantage of this scheme is that latency scales with the number of available domains, D, as each packet waits for $D - 1$ cycles per hop [159]. An improved policy transfers alternate packets from different domains, such that packets are pipelined in each dimension on on-chip networks [159].

5.4.5 Memory controller partitioning

In order to prevent attacks that exploit contention in the memory controller [117], Wang et al. [156] suggested a static time-division multiplexing of the controller, to reserve bandwidth to each domain. Their hardware simulation results suggest a 1.5% overhead on SPEC2006, but up to 150% increase in latency, and a necessary drop in peak throughput.

5.4.6 Execution leases

Tiwari et al. [141] suggested a hardware-level approach to sharing execution resources between threads, *leasing* execution resources. A new hardware mechanism would guarantee bounds on resource usage and side effects. After one lease expires, a trusted entity obtains control, and any remaining untrusted operations are expelled. Importantly, the prototype CPU is an in-order, un-pipelined processor, containing hardware for maintaining lease contexts. Further, the model does not permit performance optimisations that introduce timing variations, such as TLBs, branch predictors, making the system inherently slow. Tiwari et al. [142] later proposed a system based on this work with top-to-bottom information flow guarantees. The prototype system includes a Star-CPU, a microkernel, and an I/O protocol. The microkernel contains a simple scheduler for avionic systems with context switching cost of 37 cycles.

5.4.7 Kernel address space isolation

Gruss et al. [64] proposed isolating kernel form user address space by using separated page directories for each, so switching context between user and kernel spaces includes switching the page directory. This technique is designed to mitigate the timing attack on prefetch instructions (Section 4.4.2).

5.5 Partitioning hardware

Truly concurrent attacks can only be prevented by partitioning hardware resources among competing threads or cores. The most visible target is the cache, and it has received the bulk of the attention to date.

5.5.1 Disable hardware threading

Percival [129] advocated that Hyperthreading (Section 2.2.2) should be disabled to prevent the attack he described. This has since become common practice on public cloud services, including Microsoft's Azure [110]. This eliminates all attacks that rely on hardware threading (Sections 4.2.1, 4.3.1), at some cost to throughput.

5.5.2 Disable page sharing

Because the FLUSH+RELOAD attack and its variations depend on shared memory pages, preventing sharing mitigates the attack. VMware Inc. [152] recommends disabling the transparent page sharing feature [154] to protect against cross-VM FLUSH+RELOAD attacks. CacheBar [182] prevents concurrent access to a shared pages by automatically detecting such access and creating multiple copies. The scheme they invented is called copy-on-access, which creates a copy of a page while another security domain is trying to access on that page.

5.5.3 Hardware cache partitions

Hardware-enforced cache partitions would provide a powerful mechanism against cache-based attacks. Percival [129] suggested partitioning the L1 cache between threads to eliminate the cache contention (Section 4.3.1). While some, including Page [128], have proposed hardware designs, no commercially available hardware provides this option.

Wang and Lee [158] suggested an alternative approach, described as a *partition-locked cache* (PLcache). They aim to provide hardware mechanisms to assign locking attributes to every cache line, allowing sensitive data such as AES tables to be selectively and temporarily locked into the cache.

A series of ARM processors support cache lockdown through the L2 cache controller, such as L2C-310 cache controller [20] on ARM Coretex A9 platforms.

Domnister et al. [46] suggested reserving for each hyperthread several cache lines in each cache set of the L1 cache.

Intel's *Cache Allocation Technology* (CAT) provides an implementation of a similar technique which prevents processes from replacing the contents of some of the ways in the LLC [80]. To defeat LLC cache attacks that conduct PRIME+PROBE and EVICT+TIME techniques (Sections 4.4.1 and 4.4.2), CATalyst [108] partitions the LLC into a hybrid hardware and software managed cache. It uses CAT to create two partitions in the LLC, a secure partition and a non-secure partition. The secure partition only stores cache-pinned secure pages, whereas the non-secure partition acts as normal cache lines managed by hardware replacement algorithms. To store secure data, a user-level program allocates secure pages and preloads the data into the secure

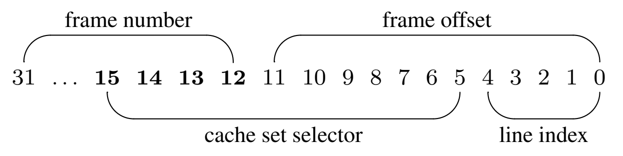

Fig. 2 Cache colouring on the Exynos4412, showing *colour bits* 15–12, where frame number and cache set selector overlap.

cache partition. As the secure data are locked in the LLC, the LLC timing attacks caused by cache line conflicts are mitigated. Liu et al. [108] demonstrated that the CATalyst is effective in protecting square-and-multiply algorithm in GnuPG 1.4.13. Furthermore, the system performance impact is insignificant, an average slowdown of 0.7% for SPEC and 0.5% for PARSEC.

The trend towards QoS features in hardware is encouraging—it may well be possible to repurpose these for security, especially if the community is able to provide early feedback to manufacturers, in order to influence the design of these mechanisms. We return to this point in Section 6.

Colp et al. [42] used the support for locking specific cache ways on ARM processors to protect encryption keys from hardware probes, and this approach will also protect against timing attacks.

5.5.4 Cache colouring

The cache colouring approach exploits set-associativity to partition caches in software. Originally proposed for improving overall system performance [28, 91] or performance of real-time tasks [104] by reducing cache conflicts, cache colouring has since been repurposed to protect against cache timing channels.

Figure 2 (reproduced from Cock et al. [41]) demonstrates how colouring operates, using as an example a cache of 2048 sets and a line size of 32 B on a processor with a 4 KiB page size (e.g. the 1 MiB 16-way associative L2 cache of the Exynos4412 processor, featuring ARM Cortex A9 cores).

The bits that belong to both the cache selector and the frame number are the *colour bits*—addresses (frames) that differ in any of these bits can never collide in the cache. In this example, the 5 least significant bits (4–0) of the physical address index 32 B cache lines, while the next 11 bits (15–5) maps a cache line to one of 2048 cache sets. The 20 most significant bits (31–11) identify a 4 KiB physical frame. Physical frames whose addresses diverge in any of these cache colouring bits are not mapped to the same cache set and thus never conflict. Cache colouring implementations divide memory into coloured memory pools and allocate memory from different pools to isolated security domains.

Shi et al. [136] designed a dynamic cache colouring solution for protecting cryptographic algorithms with threads running in the hypervisor, but did not demonstrate its effectiveness against cache-based timing attacks.

StealthMem [92] implemented a limited form of colouring, providing a small amount of coloured memory that is guaranteed to not contend in the cache. The aim is to provide *stealth pages* for storing security-sensitive data, such as the S-boxes of AES encryption. StealthMem reserves differently coloured stealth pages for each core. It disallows the usage of memory frames that have same colours as stealth pages, or monitors access to those pages through what they call the page table alert (PTA) scheme. This exploits the cache replacement algorithm to pin stealth pages in the LLC, much like the PLcache approach of Wang and Lee [158] (Section 5.5.3).

Specifically, the PTA scheme assumes that the cache implements a k-LRU replacement policy, where a miss is guaranteed not to evict any of the k most recently accessed lines. The authors experimentally determine the implementation-specific value of k for the target processor (Xeon W3520). They reserve a small number of cache colours and for each of these locks k page frames in the LLC. This is achieved through the MMU, by ensuring that attempts to access other pages of the reserved colours trigger a protection fault, which invokes StealthMem. Reserving cache colours reduces the amount of cache available to applications, which results in a relatively small overhead for SPEC 2006 CPU, measured as 5.9% for StealthMem, and 7.2% for PTA due to handling extra page faults. The overhead of using stealth pages is 2–5% for three block ciphers, AES, DES and Blowfish.

Godfrey [59] integrated cache colouring into Xen's memory management module and demonstrated that cache colouring completely closed a side channel between a pair of virtual machines. Godfrey established that the cost of colouring is roughly what would be expected given a smaller cache—a 50% performance cost for an Apache [17] macrobenchmark, and no significant penalty for benchmarks with small working sets. One significant drawback of cache colouring is the inability to use large pages. Many modern processors support large pages (up to 1 GiB on x86), to reduce TLB pressure. With large pages, there is less (frequently no) overlap between frame number and cache-selector bits in the address, reducing the number of available colours (often to one).

Cock et al. [41] evaluated the effectiveness of cache colouring for eliminating cache-based covert channels. The technique is more effective on cores with simpler structures (iMX.31, AM3358 & DM3730), comparing against cores with more complex structures (Exynos4412 & E6550). The residual bandwidth in the latter cases was principally due to TLB contention, which is solved by flushing the TLB on a VM switch.

An emerging challenge is the move away from simple direct-mapped assignment to cache sets. On recent Intel processors, the LLC is partitioned among cores, which are connected by a ring bus. Locating the cache line for a physical address is thus divided into two parts: addressing a cache block and addressing a set within that block. Sandy Bridge and newer Intel microarchitectures apply a hash function to locate blocks [77]. Without the knowledge of the hash function, the number of available colours is restricted by colours within a cache block [170]. Several authors have reverse-engineered the hash function of multiple processor models [74, 76, 84, 112, 170], supporting the use of more colours; however, this may not be possible for future CPUs.

5.5.5 Quasi-partitioning

Preventing the attacker from monopolising resources reduces the contention and with it the effectiveness of side channel attacks. CacheBar [182] actively evicts memory contents from the cache to ensure that protection domains only occupy a limited number of ways in each cache set. CacheBar allocates a budget per cache set to each security domain, representing its cacheable allowance on that set. To record occupancy, CacheBar maintains a least recently used queue per security domain for a monitored cache set; only blocks listed in the queue are mapped to the cache set. This is, essentially, a software implementation of the countermeasure that Domnister et al. [46] suggested.

5.5.6 Spatial network partitioning

Spatial network partitioning separates hardware components used for transmitting network traffic of security domains, where domains are located on disjoint subsets of cores on a many-core system [155]. Although it eliminates network interference, system resources may not be dynamically allocated according to actual demands from domains. Therefore, one part of the system may have excess network bandwidth, while the rest of the system is suffering bandwidth starvation.

5.5.7 Migrating VMs

To mitigate information leakage across co-resident VMs, Nomad [116] implemented a migration-as-a-service cloud computing service model that runs placement algorithm for deployment model configured by cloud provider. Nomad takes current and past VM assignments in the past epochs as inputs, deciding the next placement. In order to minimise the impact of service running on VMs, Nomad provides client API allowing clients to identify non-relocatable VMs.

5.6 Auditing

A completely different approach to side-channel attacks and one mandated by older standards [45] is *auditing*. Fiorin et al. [53] designed a monitoring system for networks-on-chip (Section 2.2.4), providing online data collection for bandwidth consumption from entities. Based on traffic analyses, a system can design a network security manager for detecting any malicious usage of shared networks.

Zhang and Reiter [179] analysed the impact of periodically flushing the L1 instruction and data caches in kernel-level threads and automatically switched to a defensive mode once malicious pre-emption behaviour is detected [179]. This approach is also able to audit attacks that generate frequent system pre-emptions, such as that of Zhang et al. [180].

Tan et al. [140] proposed a hardware technique to defend against branch prediction analysis attacks [5], which employs a spy record table and an extra locking attribute in BTB entries. The spy record table contains the occupancy ratio of BTB entires. Once a suspicious thread is recognised as a spy, BTB entries most recently used by other threads are locked, meaning that they can only be replaced by their owner.

Both Gruss et al. [65] and Chiappetta et al. [39] suggested using the performance monitoring unit (PMU) to detect LLC side channel attacks. Similarly, Zhang et al. [178] presented a system to detect cross-VM cache-based side channel attacks using PMU counters to monitor cache miss or hit rates while protected VM executing cryptographic operations. They demonstrated the effectiveness on detecting PRIME+PROBE and FLUSH+RELOAD cache attacks. Moreover, Zhang et al. [177] developed a mechanism to monitor memory usage (cache misses, or bus bandwidth) of a program to detect abnormal memory accessing behaviours that can lead to DoS attacks. If the monitored memory usage differs significantly from the reference distribution, the system flags the monitored program as potentially malicious.

Each of these auditing techniques employs monitoring to identify activity consistent with an attack. Thus, auditing necessarily suffers from the well-understood problems of false positives (benign behaviour incorrectly identified as malicious) and false negatives (malicious behaviour incorrectly identified as benign). While statistical approaches may be useful against new attacks whose behaviour, when viewed by the monitor, is similar to existing attacks, monitoring cannot defend against an attacker who modulates their attack specifically to avoid being identified as malicious.

6 Discussion

6.1 Trend in attacks

There is a clear trend in attacks to go "down the stack", from highly local (thread- and core-level) shared resources to system-shared resources. This is not surprising: the high-level shared resources are easiest to exploit, and as countermeasures are deployed, attackers move to the next target. We clearly observe this trend from published cache-based side-channel attacks listed in Table 2.

At the beginning, attackers targeted shared L1 caches on hardware threading enabled platforms [1, 7, 35, 125, 129, 143], which are mitigated by disabling hardware threading [129]. Disabling hardware threading also advocated by cloud service providers, including Microsoft's Azure [110]. Therefore, thread-level attacks seem to be pretty much done to death, operators just disable hardware multithreading on shared machines.

Meanwhile, attackers also explored L1 caches without hardware threading support, by generating cache contentions between processes [1, 8, 120, 125, 143] or VMs [180]. For addressing these attacks, system designers proposed hardware cache partitioning mechanism [129, 158], RPcache [97, 158], random fill cache [106] and minimum timeslice techniques [149]. Core-level attacks are still fairly easy, but also easy to defend against. The simplest defence is not to share cores, at least between different owners, which may not be completely unreasonable in clouds. Even with sharing cores, it should be easy to prevent sharing the L1 caches: given the small size of the L1, and the coarse granularity of VM "world" switches (typical rates are 33 Hz or less [180]), the probability of any data being hot in the cache after a world switch is very low, and thus the indirect cost (Section 2.2.1) of flushing the L1 on a world switch is negligible. Similar arguments hold for the TLB. The main obstacle here is that on x86 hardware, which is presently the de facto standard architecture in clouds, the architecture does not support a selective flush of the L1. Flushing the complete cache hierarchy, even on the coarse granularity of world switches, incurs a significant performance penalty. Hardware support for L1 flushes should be easy to add and would go a long way to making core sharing safer. This would still have to be complemented by preventing VMs from forcing frequent world switches.

Later, attacks moved to LLC, the cache shared by processes on a platform. High-resolution, cross-core attacks through the LLC can be classified into two groups. Some attacks rely on shared memory [24, 63, 71, 81, 105, 147, 168, 169, 181], e.g. library code. While VMs do not directly share memory, the hypervisor may artificially create such sharing through memory de-duplication in order to increase memory utilisation [18, 154]. This is a bad idea from the security point of view, and hypervisor providers now advise against it [153].

High-resolution LLC attacks that do not rely on shared memory are a recent development [83, 105, 107, 123]. Some countermeasures against such attacks have been proposed, for example hardware cache partitions (Section 5.5.3) and cache colouring (Section 5.5.4). However, more work is required to evaluate these countermeasures both in terms of protection and in terms of performance overhead.

6.2 Challenges for defenders

Defences based on completely virtualising time are not applicable in the cloud context—it is just not feasible to deny access to real time to a cloud tenant [58]. Therefore, any defence must be based on resource isolation.

Fundamentally, clouds represent a classical trade-off between security and economics. It would be most secure not to share hardware between distrusting tenants, but the cloud business model is inherently based on the economic benefits of sharing resources. With per-package core counts increasing, the likelihood of an attacker being co-located on the same processor with a VM worth attacking is increasing too. All the attacker has to do is pay for some processing time.

We already mentioned the lack of a selective L1 cache flush on x86 as an impediment to reducing sharing. Cache colouring is not applicable to the L1 cache, as it is virtually indexed (Section 2.2.1) and on practically all recent processors is too small to colour (i.e. consists of a single colour). Colouring could be applied to other caches. However, as, by definition, successive pages are of different colour, this implies that the hypervisor would have to assign physical frames to VMs in a checkerboard fashion. This rules out the use of large page mappings in the hypervisor (as well as the TLB even if the guest uses large mappings). Large pages significantly reduce TLB pressure and are therefore generally used by hypervisors to improve the performance of the virtual memory system.

Memory controllers and interconnects have been used for DoS attacks and covert channels, although no side-channel attacks through the interconnect have been demonstrated yet. Besides, time multiplexing (Section 5.4.4) and spatial network partitioning (Section 5.5.6), techniques introduced by the real-time community to improve temporal isolation, such as budgets for bus time [173], could be used to prevent DoS attacks. Side-channel attacks though the interconnect seem very hard to execute, but, given the lack of effective defences, are a promising target and might appear in the future.

Alternatively, by forcing contention on a shared resource and measuring the throughput, attackers can detect the victim's use of the resource, creating a side channel [171].

6.3 Future directions for countermeasures

Improved resource isolation between cloud tenants is not only good for security, but also predictability of performance, so something providers should be economically motivated to pursue.

For example, resource-freeing attacks provide the potential for a malicious tenant to improve its own performance by bottlenecking another [148].

Fundamentally, improving isolation without undue performance degradation will require improved hardware support. At present, however, the opposite frequently happens: hardware manufacturers are still too focussed on average-case performance, and in the process of optimising the microarchitecture, frequently introduce new channels [41].

Existing hardware mechanisms that could be used often lack some of the features to make them effective for security. For example, the widely available PMU allows measuring relevant events, but does not provide sufficient information on whose execution triggered those events. For example, Inam et al. [75] demonstrated a scheduling algorithm for equally distributing bus bandwidth by monitoring bus usage through PMU. However, due to limitations in the information provided by the PMU, the approach requires an interrupt for every bus request, clearly impractical for a general-purpose system.

Some help may come from developments that are designed to support QoS isolation. For example, Intel recently introduced CAT, which associates particular ways of the set-associative cache with a class of service (COS) [78, 80]. However, the size of the LLC introduces the upper bound on the amount of data that can be concurrently locked in the cache.

7 Conclusions

Microarchitectural timing attacks can be used in cloud scenarios for purposes ranging from malicious performance interference to stealing encryption keys. They have moved from the easy targets (using hardware multithreading to attack local resources) down the hierarchy to system-wide shared resources, where they are harder (and less economical) to defend against.

We expect this trend to continue. The economic model of clouds is fundamentally based on sharing of hardware, and sharing is likely to increase with the growing number of cores per processor package, ruling out most simple defences based on reducing sharing. Satisfying both economics of clouds and security will require improved hardware support for partitioning hardware.

Acknowledgements

We would like to thank Toby Murray for his comments and feedback.

References

1. Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, US, 2007.
2. Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *International Conference on Information and Communications Security (ICICS)*, pages 112–121, Raleigh, NC, US, December 2006.
3. Onur Acıiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In *Cryptographic Engineering*, pages 475–504. 2009.
4. Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openSSL and necessary software countermeasures. In *11th IMA International Conference on Cryptography and Coding*, pages 185–203, Cirencester, UK, 2007.
5. Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 2007 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 225–242, 2007.
6. Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *2nd ACM symposium on Information, computer and communications security*, Singapore, 2007.
7. Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Workshop on Cryptographic Hardware and Embedded Systems*, Santa Barbara, CA, US, 2010.
8. Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 256–273, San Francisco, CA, US, 2008.
9. Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, Vienna, AT, 2007.
10. Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the AES. In *Proceedings of the 2007 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 271–286, San Francisco, CA, US, 2007.

11. Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 526–540, San Francisco, CA, May 2013. .
12. Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Annual Computer Security Applications Conference*, Los Angeles, CA, US, December 2016.
13. AMD. AMD FX processors. Online: http://www.amd.com/en-us/products/processors/desktop/fx.
14. Don Anderson and Jay Trodden. *Hypertransport system architecture*. 2003.
15. Ross J. Anderson. *Security Engineering: A Guide to Building Depandable Distributed Systems*. Second edition, 2008.
16. Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*, San Jose, CA, US, May 2015.
17. Apache. Apache http server benchmarking tool, 2013.
18. Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the 2009 Ottawa Linux Symposium*, pages 19–28, Montreal, Quebec, Canada, July 2009.
19. ARM. ARMv8 instruction set overview, . Online: https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf.
20. ARM. Corelink level 2 cache controller L2C-310 technical reference manual, . Online: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246h/DDI0246H_l2c310_r3p3_trm.pdf.
21. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 520–538, Chicago, IL, US, 2010.
22. Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *ACM Workshop on Cloud Computing Security*, pages 103–108, Chicago, IL, US, 2010.
23. Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Vancouver, BC, 2010.
24. Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah…, just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92, Busan, KR, September 2014.
25. Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Preprint available at http://cr.yp.to/papers.html#cachetiming.
26. Daniel J. Bernstein and Peter Schwabe. A word of warning. Workshop on Cryptographic Hardware and Embedded Systems'13 Rump Session, August 2013.
27. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, Santiago, CL, October 2012.
28. Brian N. Bershad, D. Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.
29. Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, US, 2003.
30. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Workshop on Cryptographic Hardware and Embedded Systems*. Yokohama, JP, 2006.
31. Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.
32. Ernie Brickell. Technologies to improve platform security. Workshop on Cryptographic Hardware and Embedded Systems'11 Invited Talk, September 2011. URL http://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf.
33. Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006: 52, 2006.
34. Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, February 2006.
35. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Proceedings of the 15th Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 667–684, Tokyo, JP, December 2009. .
36. David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, Washington, DC,

US, 2003. .

37. Yuriy Bulygin. CPU side-channels vs. virtualization malware: the good, the bad or the ugly. In *ToorCon: Seattle*, Seattle, WA, US, April 2008.
38. Carlos Cardenas and Rajendra V Boppana. Detection and mitigation of performance attacks in multi-tenant cloud computing. In *1st International IBM Cloud Academy Conference*, Research Triangle Park, NC, US, 2012.
39. Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. IACR Cryptology ePrint Archive, Report 2015/1034, October 2015.
40. David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. .
41. David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, November 2014.
42. Patrick J. Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Istambul, TK, March 2015.
43. Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 45–60, Oakland, CA, US, May 2009.
44. Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–242, 1976. .
45. DoD. *Trusted Computer System Evaluation Criteria*. Department of Defence, 1986. DoD 5200.28-STD.
46. Leonid Domnister, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4), January 2012.
47. DotCloud. DotClod developer cloud platform. URL https://www.dotcloud.com/.
48. Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 18(1):4, June 2015.
49. George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, US, 2002.
50. George Washington Dunlap, III. *Execution replay for intrusion analysis*. PhD thesis, University of Michigan, 2006.
51. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, Santa Barbara, CA, US, 1985.
52. Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th ACM/IEE International Symposium on Microarchitecture*, Taipei, Taiwan, October 2016.
53. Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. A security monitoring service for NoCs. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*, pages 197–202, Atlanta, GA, USA, 2008.
54. Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, January 2016.
55. Bryan Ford. Plugging side-channel leaks with timing information flow control. In *Proceedings of the 4th USENIX Workschop on Hot Topics in Cloud Computing*, pages 1–5, Boston, MA, USA, 2012.
56. Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved trace-driven cache-collision attacks against embedded AES implementations. In *Workshop on Information Security Applications*, pages 243–257, Jeju Islang, KR, August 2010.
57. Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.
58. Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Workshop on Hot Topics in Operating Systems*, San Diego, CA, US, 2007.
59. Michael Godfrey. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, Ont, CA, September 2013.
60. Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *Proceedings of the 6th IEEEInternational Conference on Cloud Computing*, Santa Clara, CA, US, 2013.
61. Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Sym-*

posium on Security and Privacy, pages 11–20, Oakland, California, USA, April 1982.

62. Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *Proceedings of the 35th ACM/IEE International Symposium on Microarchitecture*, pages 409–418, Istanbul, TR, November 2002.
63. Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium*, pages 897–912, Washington, DC, US, August 2015.
64. Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.
65. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, San Sebastián, Spain, July 2016.
66. Shay Gueron. Intels new AES instructions for enhanced performance and security. In *Fast Software Encryption*, pages 51–66. Springer, 2009.
67. Shay Gueron. Intel advanced encryption standard (AES) instructions set, 2010. Online: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.
68. Shay Gueron. Efficient software implementations of modular exponentiation. *Journal of Cryptographic Engineering*, 2(1):31–43, May 2012.
69. Shay Gueron and Michael Kounavis. Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Information Processing Letters*, 110(14–15):549–553, July 2010.
70. Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. Intel White Paper 323640-001 Revision 2.0, May 2010.
71. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 490–505, Oakland, CA, US, May 2011.
72. Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, Oakland, CA, US, 1991.
73. Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–61, Oakland, CA, US, 1992.
74. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205, San Francisco, CA, May 2013.
75. Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, Berlin, DE, 2014.
76. Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2016.
77. Intel 64 & IA-32 AORM. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, April 2012.
78. Intel 64 & IA-32 ASDM. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2*. Intel Corporation, June 2014.
79. Intel 64 & IA-32 ASDM. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, August 2015. http://www.intel.com.au/content/www/au/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html.
80. Intel CAT. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Intel Corporation, April 2015.
81. Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 299–319, Gothenburg, Sweden, September 2014.
82. Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-VM attacks on Xen and VMware. In *Proceedings of the 4th IEEE International Conference on Big Data and Cloud Computing*, Sydney, Australia, 2014.
83. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*, San Jose, CA, US, May 2015.
84. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conference on Digital System Design*, Funchal, Madeira, Portugal, August

2015.

85. Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Asia Conference on Computer and Communication Security (ASIA CCS)*, pages 85–96, Singapore, April 2015.
86. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Asia Conference on Computer and Communication Security (ASIA CCS)*, pages 353–364, Xi'an, CN, May 2016.
87. Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.
88. Robert Kaiser and Stephen Wagner. Evolution of the PikeOS microkernel. In *International Workshop on Microkernels for Embedded Systems*, pages 50–57, Sydney, AU, January 2007.
89. Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Design Automation Conference (DAC)*, Austin, TX, US, June 2016.
90. Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, US, August 2014.
91. R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.
92. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, pages 189–204, Bellevue, WA, US, August 2012.
93. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009.
94. Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. .
95. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *International Cryptology Conference—CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, 1999. .
96. Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1:5–27, April 2011.
97. Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture*, Raleigh, NC, US, 2009.
98. Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, New York, NY, US, 2009.
99. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 564–580, 2012.
100. Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973. .
101. Adam Langley. ctgrind. https://github.com/agl/ctgrind, 2010.
102. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, CA, US, March 2004.
103. Peng Li, Debin Gao, and Michael K Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, Budapest, HU, 2013.
104. Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, CA, June 1997. .
105. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, pages 549–564, Austin, TX, US, August 2016.
106. Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *Proceedings of the 47th ACM/IEE International Symposium on Microarchitecture*, Cambridge, UK, December 2014.
107. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015.
108. Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks

in cloud computing. In *IEEE Symposium on High-Performance Computer Architecture*, pages 406–418, Barcelona, Spain, March 2016.

109. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. 2008.
110. Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. Security best practices for developing windows azure applications. *Microsoft Corp*, 2010.
111. Robert Martin, John Demme, and Simha 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, Portland, OR, US, June 2012. . URL http://doi.acm.org/10.1145/2366231.2337173.
112. Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Kyoto, Japan, November 2015.
113. Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Milano, Italy, 2015.
114. Thomas S Messerges, Ezzat A Dabbish, and Robert H Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.
115. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170): 519–521, April 1985.
116. Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1595–1606, Denver, CO, US, October 2015.
117. Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, US, 2007.
118. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. .
119. Michael Neve. *Cache-based Vulnerabilities and SPAM analysis*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, July 2006.
120. Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *13th International Workshop on Selected Areas in Cryptography*, Montreal, CA, 2006.
121. Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, pages 147–162, Montreal, CA, August 2006.
122. ORACLE. SPARC T4 processor. Online: http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t4-processor-ds-497205.pdf.
123. Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1406–1418, Denver, CO, US, October 2015.
124. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, November 2005.
125. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*, 2006.
126. Daniel Page. Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive, Report 2002/169, November 2002.
127. Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.
128. Daniel Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005:280, 2005.
129. Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.
130. Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, US, August 2016.
131. Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, US, August 2016.
132. Andre Richter, Christian Herber, Holm Rauchfuss, Thomas Wild, and Andreas Herkersdorf. Performance isolation exposure in virtualized platforms with PCI passthrough I/O sharing. In *Architecture of Computing Systems*, pages 171–182. 2014.

133. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan 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, Chicago, IL, US, 2009.
134. Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the annual ACM Conference*, pages 404–410, 1977.
135. Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on AES. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 163–175, Boston, MA, US, 2004.
136. Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, HK, June 2011.
137. Bruno R. Silva, Diego Aranha, and Fernando M. Q. Pereira. Uma técnica de análise estática para detecção de canais laterais baseados em tempo. In *Brazilian Symposium on Information and Computational Systems Security*, pages 16–29, Florianópolis, SC, BR, December 2015.
138. WonJun Song, John Kim, Jae-Wook Lee, and Dennis Abts. Security vulnerability in processor-interconnect router design. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, AZ, US, 2014.
139. Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium On Research in Computer Security*, pages 718–735, Egham, UK, September 2013. .
140. Ya Tan, Jizeng Wei, and Wei Guo. The micro-architectural support countermeasures against the branch prediction analysis attack. In *Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Beijing, China, 2014.
141. Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd ACM/IEE International Symposium on Microarchitecture*, New York, NY, US, 2009.
142. Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th International Symposium on Computer Architecture*, San Jose, CA, US, 2011.
143. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
144. Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, Xi'an, CN, October 2002.
145. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76, Cologne, DE, 2003.
146. Valgrind. URL http://valgrind.org/.
147. Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *Proceedings of the 2015 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 3–21, San Francisco, CA, USA, April 2015.
148. Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, US, 2012.
149. Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, US, 2014.
150. Tsvetoslava Vateva-Gurova, Neeraj Suri, and Avi Mendelson. The impact of hypervisor scheduling on compromising virtualized environments. In *IEEE International Conference on Computer and Information Technology*, pages 1910–1917, 2015.
151. Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *ACM Workshop on Cloud Computing Security*, pages 41–46, Chicago, IL, October 2011. ACM.
152. VMware Inc. Security considerations and disallowing inter-virtual machine transparent page sharing. VMware Knowledge Base 2080735 http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735, October 2014.
153. VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing. VMware Knowledge Base 2080735 http://kb.vmware.com/selfservice/microsites/

search.do?language=en_US&cmd=displayKC&externalId=2080735, October 2014.

154. Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, US, 2002.
155. Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th ACM/IEEE International Symposium on Networks on Chip*, pages 142–151, Lyngby, Denmark, 2012.
156. Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. Timing channel protection for a shared memory controller. In *Proceedings of the 20th IEEE Symposium on High-Performance Computer Architecture*, Orlando, FL, US, 2014.
157. Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference*, Miami Beach, FL, US, 2006.
158. Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, US, June 2007.
159. Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 583–594, 2013.
160. Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In *Financial Cryptography and Data Security*, Bonaire, Dutch Caribbean, February 2012.
161. Michael Weiß, Benjamin Weggenmann, Moritz August, and Georg Sigl. On cache timing attacks considering multi-core aspects in virtualized embedded systems. In *Proceedings of the 6th International Conference on Trustworthy Systems*, Beijing, China, 2014.
162. Dong Hyuk Woo and Hsien-Hsin S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Phoenix, AZ, US, 2007.
163. John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7, Oakland, CA, US, May 1991.
164. Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in Deterland. In *Conference on Timely Results in Operating Systems*, Monterey, CS, US, October 2015.
165. Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, US, 2012.
166. Leslie Xu. Securing the enterprise with intel AES-NI, 2010. Online: http://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html.
167. Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Workshop on Cloud Computing Security*, pages 29–40, 2011.
168. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, February 2014.
169. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, CA, US, 2014.
170. Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. http://eprint.iacr.org/, September 2015.
171. Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Conference on Cryptographic Hardware and Embedded Systems 2016 (CHES 2016)*, Santa Barbara, CA, US, August 2016.
172. Kent Yoder. POWER7+ accelerated encryption and random number generation for Linux, 2013.
173. Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, Philadelphia, PA, US, April 2013.
174. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 563–574, Chicago, IL, US, 2011.
175. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, Beijing, CN, 2012.
176. Rongting Zhang, Xin Su, Jiacheng Wang, Chingyue Wang, Wenxin Liu, and Rynson WH Lau. On mitigating the risk of cross-VM covert channels in a public

cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26:2327–2339, 2014.

177. Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Memory DoS attacks in multi-tenant clouds: Severity and mitigation. *arXiv preprint arXiv:1603.03404v2*, 2016.
178. Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Proceedings of the 19th Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Telecom SudParis, France, September 2016.
179. Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 827–838, Berlin, DE, November 2013.
180. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 305–316, Raleigh, NC, US, October 2012.
181. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant side-channel attacks in PaaS clouds. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, AZ, US, 2014.
182. Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.