Time Protection: The Missing OS Abstraction

Qian Ge
UNSW Australia and Data61 CSIRO
qian.ge@data61.csiro.au

Yuval Yarom
The University of Adelaide and Data61 CSIRO
yval@cs.adelaide.edu.au

Tom Chothia
University of Birmingham
T.P.Chothia@cs.bham.ac.uk

Gernot Heiser
UNSW Australia and Data61 CSIRO
gernot@unsw.edu.au

Abstract

Timing channels enable data leakage that threatens the security of computer systems, from cloud platforms to smartphones and browsers executing untrusted third-party code. Preventing unauthorised information flow is a core duty of the operating system, however, present OSes are unable to prevent timing channels. We argue that OSes must provide *time protection*, the temporal equivalent of the established memory protection, for isolating security domains. We examine the requirements of time protection, present a design and its implementation in the seL4 microkernel, and evaluate efficacy and cost on x86 and Arm processors.

CCS Concepts • **Security and privacy → Trusted computing**; • **Software and its engineering →** *Multiprocessing / multiprogramming / multitasking*;

Keywords timing channels, covert channels, temporal isolation, time protection, microkernels, security, confidentiality, seL4

1 Introduction

One of the oldest problems in operating systems (OS) research is how to confine programs so they do not leak information [Lampson 1973]. To achieve confinement, the operating system needs to control all of the means of communication that the program can use. For that purpose, programs are typically grouped into *security domains*, with the operating system exerting control on cross-domain communication.

Programs can bypass OS protection by sending information over media not intended for communication. Historically, such *covert channels* were explored in the context of military systems [Department of Defence 1986]. Cloud computing, smartphone apps and server-provided JavaScript mean that we now routinely share computing platforms with untrusted, potentially malicious, third-party code.

OSes have traditionally enforced security through *memory protection*, i.e. spatial isolation by partitioning memory between security domains. Recent advances include formal proof of spatial security enforcement by the seL4 microkernel [Klein et al. 2014], including proof of the absence of covert *storage channels* [Murray et al. 2013], i.e. channels based on storing information that can be later loaded [Department of Defence 1986; Schaefer et al. 1977]. Spatial isolation is therefore a solved problem.

In contrast, *timing channels*, and in particular *microarchitectural channels* [Ge et al. 2018b], which exploit timing variations due to shared use of caches and other hardware, remain a fundamental OS security challenge that has eluded a comprehensive solution to date. Its importance is highlighted by recent attacks, including the extraction of encryption keys across cores through *side channels* [Irazoqui et al. 2015; Liu et al. 2015], i.e. without the cooperation of the key owner.

Unlike side channels, covert channels use insider help (Trojan) and are traditionally considered a lesser threat. However, in the recent Spectre attack [Kocher et al. 2019], an adversary uses a covert channel with a Trojan constructed from speculatively executed *gadgets* to leak information. This demonstrates that covert channels pose a real security risk even where no side-channel attack is known. Furthermore, a covert channel bears the risk of being exploitable as a side channel by an ingenious attacker.

We argue that it is time to take temporal isolation seriously, and make the OS[1] responsible for *time protection*, the prevention of temporal interference [Ge et al. 2018a], just as memory protection prevents spatial interference. This requires a design that eliminates, as far as possible, the sharing of hardware resources that is the underlying cause of timing channels. Ultimately we aim to obtain temporal isolation guarantees comparable to the spatial isolation proofs of seL4, but for now we focus on *mechanisms* that are suitable for a verifiable OS kernel, i.e. minimal, general and policy-free.

Specifically, we make the following contributions.

EuroSys '19, March 25–28, 2019, Dresden, Germany

[1] We use "OS" to refer to the most privileged software level that is responsible for security enforcement, which could be a hypervisor.

- We define *time protection* for preventing microarchitectural timing channels, and specify the requirements for its implementation (Section 3.2);
- we introduce a policy-free *kernel clone* operation that partitions a system almost perfectly, removing most sharing between domains, and explore how accessing the remaining shared state can be made sufficiently deterministic to prevent leakage (Section 3.3);
- we present an *implementation in seL4* (Section 4);
- we show that our implementation of time protection is *effective*, transparently removing timing channels, within limitations of present hardware (Section 5.3);
- we show that the *overhead is low* (Section 5.4).

2 Background

2.1 Covert channels and side channels

A covert channel is an information flow that uses a mechanism not intended for information transfer [Lampson 1973]. By allowing communication between *security domains* that should be isolated, covert channels may violate the system's security policy. Here a (security) domain is the granularity of restrictions imposed by a system's security policy; it may consists of multiple OS protection domains (set of access rights), which may further restrict access for software-engineering/safety reasons.

There is a traditional distinction between *storage* and *timing channels*, where exploitation of the latter requires the communicating domains to have a common notion of time [Department of Defence 1986; Schaefer et al. 1977; Wray 1991]. In principle, it is possible to completely eliminate storage channels, as was done in the information-flow proof of seL4 [Murray et al. 2013].[2]

Despite progress on proving upper bounds for the cache side channels of cryptographic implementations [Doychev et al. 2013; Köpf et al. 2012], proofs of elimination of timing channels in a non-trivial system are beyond current formal approaches; measurements are essential for their analysis.

In a narrow sense, a *covert channel* requires collusion between the domains, one acting as a *sender* and the other as a *receiver*. Typical cases of senders are Trojans, i.e. trusted code that operates maliciously, or untrusted code that is being *confined* [Lampson 1973]. Due to the collusion, a covert channel represents a worst case for bandwidth of a channel.

In contrast, a *side channel* has an unwitting sender, called the *victim*, which, through its normal operation, is leaking information to an *attacker* acting as the receiver. An important example is a victim executing in a virtual machine (VM) on a public cloud, which is being attacked by a malicious co-resident VM [İnci et al. 2016; Yarom and Falkner 2014].

[2]Specifically, the proof shows that no machine state that is touched by the kernel can be used as a storage channel, it does not exclude channels through state of which the kernel is unaware.

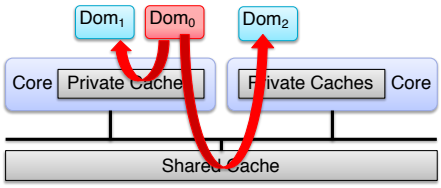

Figure 1. Competition for limited hardware resources result in interference that can leak information intra-core (Dom_0 to Dom_1) or inter-core (Dom_0 to Dom_2).

The achievable bandwidth of a side channel attack is generally orders of magnitude lower than the covert channel using the same mechanism. Bandwidth is obviously less of an issue if the secret is a small but long-lived asset, such as a web server's SSL key.

2.2 Microarchitectural channels

Microarchitectural timing channels result from competition for hardware resources that are functionally transparent to software [Ge et al. 2018b]. The instruction-set architecture (ISA), i.e. the hardware-software contract, abstracts these resources away, as they are irrelevant for functional correctness. However, the abstraction leaks, as it affects observable execution speed, leading to timing channels.

These resources come in two categories, see Figure 1.

1. **Microarchitectural state** leverages temporal and spatial locality to improve average-case performance. It includes data and instruction caches, TLBs, branch predictors, instruction- and data-prefetcher state machines, and DRAM row buffers. Without hyperthreading, core-local resources are time-shared, else they are concurrently accessed like shared caches.
2. **Stateless interconnects** include buses and on-chip networks. Time sharing cannot produce interference on these, while concurrent access can be observed as a reduction of available bandwidth.

Cache channels work by the sender (intentionally or incidentally) modulating its footprint in the cache through its execution, and the receiver probing this footprint by systematically touching cache lines and measuring memory latency by observing its own execution speed. Low latency means that a line is still in the cache from an earlier access, while high latency means that the corresponding line has been replaced by the sender competing for cache space.

Side-channel attacks are similar, except that the sender does not actively cooperate, but accesses cache lines according to its computational needs. Thus, the attacker must synchronise its attack with the victim's execution and eliminate any noise with more advanced techniques. Side-channel attacks have been demonstrated against the L1-D [Hu 1992] and L1-I caches [Acıiçmez 2007], the TLB [Gras et al. 2018;

Hund et al. 2013] and the BP [Acıiçmez et al. 2007]. Cross-core side-channel attacks through the last-level cache (LLC) have also been demonstrated [Irazoqui et al. 2015; Liu et al. 2015; Maurice et al. 2017]. Side-channel attacks through hyperthreading are plentiful [Acıiçmez and Seifert 2007; Percival 2005; Yarom et al. 2016].

On stateless channels, the sender encodes information into its bandwidth consumption, and the receiver senses the available bandwidth. Interconnects have been exploited as covert channel [Hu 1991; Wu et al. 2012], but no side-channel attacks are known [Ge et al. 2018b]; as long as the interconnect does not leak data or address information, they are probably infeasible.[3]

2.3 Countermeasures

Countermeasures must prevent interference resulting from resource competition while processing secret information.

On **stateful resources**, this can be achieved by partitioning the resources either spatially or temporally (i.e. time sharing).[4] The latter requires flushing all history-dependent state between time slices, which is conceptually simple (but potentially difficult in practice, see Section 4.3). Temporal partitioning is obviously not possible where domains access a resource concurrently, e.g. a cache shared between cores. It would also be very costly in the case of large caches (LLC), as we demonstrate in Section 5.2.

Spatial partitioning by the OS is only possible where the OS has control over domains' access to the shared infrastructure. As the OS controls the allocation of physical memory frames to domains, it can control access to physically-indexed caches (generally the L2…LLC). The standard technique is *page colouring*, which makes use of the fact that in large set-associative caches, the set-selector bits in the address overlap with the page number. A particular page can therefore only ever be resident in a specific section of the cache, referred to as the "colour" of the page. The OS can partition the physically-indexed cache by allocating frames of disjoint colours to domains [Kessler and Hill 1992; Liedtke et al. 1997; Lynch et al. 1992]. With a page size of P, a cache of size S and associativity w has S/wP colours.

On most hardware the OS cannot colour the small L1 caches, because they only have a single colour, but also because they are generally indexed by virtual address, which is not under OS control. The same applies to other on-core state, such as the TLB and BP. Hence, *these on-core caches must be flushed on a domain switch.*

Some architectures provide hardware mechanisms for partitioning caches. For example, many Arm processors support pinning whole sets of the L1 I- and D-caches [ARM Ltd. 2008]. Prior work has used this feature to provide a small amount of safe, on-chip memory for storing encryption keys [Colp et al. 2015]. Similarly, Intel recently introduced a feature called *cache allocation technology* (CAT), which supports way-based partitioning of the LLC, and which also can be used to provide secure memory [Liu et al. 2016].

Although such secure memory areas can be used to protect secrets from side channels, they need to be actively (and correctly) used by the application holding the secret. However, enforcement of a system's security policy must not depend on correct application behaviour. Hence time protection, like memory protection, must be a *mandatory* (black-box) OS security enforcement mechanism. In particular, only mandatory enforcement can support confinement.

For **bandwidth-limited interconnects**, channel prevention requires partitioning the bandwidth, by time-multiplexing the interconnect or by using some hardware partitioning mechanism. No support for bandwidth partitioning exists on contemporary mainstream hardware.[5] Time-multiplex the interconnect requires the OS to explicitly manage cache content [Yun et al. 2013] and comes at the cost of severely degraded interconnect utilisation.

2.4 seL4

seL4 is a third-generation OS microkernel that is designed from the ground up for use in security- and safety-critical systems. Its unique assurance includes formal, machine-checked proofs that the implementation (at the level of the executable binary) is functionally correct against a formal model, and that the formal model enforces integrity and confidentiality (ignoring timing channels) [Klein et al. 2014].

Like other security-oriented systems [Bomberger et al. 1992; Shapiro et al. 1999], seL4 uses capabilities [Dennis and Van Horn 1966] for access control: any access must be authorised by an appropriate capability. seL4 takes a somewhat extreme view of policy-mechanism separation [Levin et al. 1975], by delegating all memory management to user level. After booting up, the kernel never allocates memory; it has no heap and uses a strictly bounded stack. Any memory that is free after the kernel boots is handed to the initial usermode process as "Untyped" (meaning unused) memory.

Memory needed by the kernel for object metadata, including page tables, thread control blocks (TCBs) and capability storage, must be supplied by the usermode process which creates the need for it. For example, if a process wants to create a new thread, besides providing memory for that thread's stack, also it must hand to the kernel memory for storing the TCB. This is done by "re-typing" some Untyped memory into the TCB kernel object type. While userland now

[3] A recently published bus side-channel attack [Wang and Suh 2012] was only demonstrated in a simulator. More importantly, it relies on the cache being small, making it inapplicable to modern processors.

[4] In principle, it is also possible to prevent timing channels by denying attackers access to real time, but in practice this is infeasible except in extremely constrained scenarios.

[5] Intel recently introduced *memory bandwidth allocation* (MBA) technology, which imposes *approximate* limits on the memory bandwidth available to a core [Intel Corporation 2016]. While a step towards bandwidth partitioning, the approximate enforcement is insufficient for preventing covert channels.

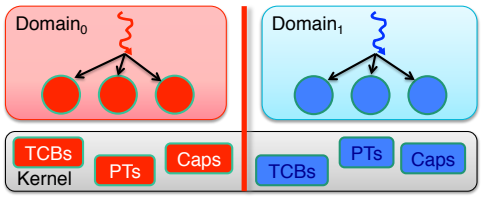

Figure 2. seL4's memory management model extends partitioning of user memory to kernel metadata.

holds a capability to a kernel object, it cannot access its data directly. Instead, the capability is the authentication token for performing system calls on the object (e.g. manipulating a thread's scheduling parameters) or destroying the object (and thereby recovering the original Untyped memory).

This model of memory management has profound consequences, it is an enabler of the proofs about seL4's isolation enforcement. For example, the initial process might do nothing but partition free memory into two pools, initiate a process in each pool, giving it complete control over its pool but no access to anything else, and then commit suicide. This system will then remain strictly (and provably) partitioned for the rest of its life, with no (overt) means of communication between the partitions. Furthermore, as kernel metadata is stored in memory provided to the kernel by userland, it is as partitioned as userland (see Figure 2).

3 Attacks and Defences

3.1 Threat scenarios

We aim to develop general time-protection mechanisms suitable for a wide range of use cases. However, we note the lack of hardware support for preventing interconnect channels (see Section 2.3). Recalling that these can only be exploited as covert channels between from concurrently executing domains, we have to restrict ourselves to threat scenarios which exclude these kinds of channels. Once suitable hardware support becomes available, time protection can be generalised.

Even with this restriction we can provide security to many use cases. We pick two threat scenarios which represent important use cases.

3.1.1 Confinement

The confinement scenario [Lampson 1973] uses mandatory security enforcement to stop a Trojan from leaking secrets. The Trojan may be malicious or compromised code in a library, third-party app, server-supplied JavaScript, or low-assurance code in a military-style *cross-domain device*, or it could be constructed from gadgets in a Spectre attack. A confined component would run in a security domain of its own, connected to the rest of the system by explicit (e.g. IPC) input and output channels.

To avoid the interconnect channel, we have to assume that the system either runs on a single core (at least while the sensitive code is executing), or co-schedules domains across the cores, such that at any time only one domain executes. This is clearly restrictive, but the best we can do on present hardware.

3.1.2 Cloud

A public cloud hosts VMs belonging to mutually-distrusting clients executing concurrently on the same processor. As the VMs are able to communicate with the outside world, covert channels are impossible to prevent, so the interconnect channel is of no relevance. Instead we aim to prevent side channels, where an attacking VM is trying to infer secrets held by a victim VM.

Hyperthreading is fundamentally based on improving throughput by sharing resources between execution contexts; partitioning those resources would result in separate cores. Timing channels between hyperthreads are thus inherent, and we assume that hyperthreading is either disabled or that all hyperthreads of a core belong to the same VM. This is consistent with advice from hypervisor providers [Marshall et al. 2010; Zhang et al. 2012]. We do allow time-multiplexing a core between domains.

Characteristic of the cloud scenario is that it is very performance-sensitive. The business model of the cloud is fundamentally based on maximising resource utilisation, which rules out restrictions such as not sharing processors between VMs. This also means that solutions that lead to significant overall performance degradation are not acceptable.

3.2 Time protection

We propose time protection to address these threats.

> **Definition: Time protection**
> A collection of OS mechanism which jointly prevent interference between security domains that would make execution speed in one domain dependent on the activities of another.

Time protection must spatially partition concurrently shared resources and flush time-multiplexed resources during domain switches. As discussed in Section 2.3, flushing the virtually indexed on-core state (L1, TLB, BP) is unavoidable where a core is time-multiplexed between domains.

> **Requirement 1: Flush on-core state**
> When time-sharing a core, the OS must flush on-core microarchitectural state on domain switch, unless the hardware supports partitioning such state.

Other core-private state, such as the (physically addressed) L2 in Intel processors, could be partitioned spatially or temporally. Caches shared between cores running separate domains, as the LLC in the Cloud scenario, must be spatially partitioned (e.g. using page colouring). (Flushing the LLC would also introduce high overhead, see Section 5.2.)

Page colouring rules out sharing of physical frames between domains, whether explicitly or transparently via page deduplication, and thus may increase the aggregate memory footprint of the system. However, this is unavoidable, as even (read-only) sharing of code has been shown to produce exploitable side channels [Gullasch et al. 2011; Yarom and Falkner 2014]. We are not aware of any public cloud provider that supports cross-VM deduplication and some hypervisor providers explicitly discourage the practice [VMware Knowledge Base 2014] due to the risks it presents.

This leaves the kernel itself. Similar to shared libraries, the kernel's code and data can also be used as a timing channel, as we will demonstrate in Section 5.3.1.

Requirement 2: Partition the OS
Each domain must have its private copy of OS text, stack and (as much as possible) global data.

As discussed in Section 2.4, partitioning most kernel data is straightforward in seL4: all dynamically allocated kernel memory is provided by userland. Hence, colouring user memory will colour all dynamic kernel data structures. This leaves an (in seL4 small) amount of global kernel data uncoloured.

Requirement 3: Deterministic data sharing
Access to any remaining shared OS data must be sufficiently deterministic to avoid information leakage.

The latency of flushing on-core caches can also be used as a channel, as we will show in Section 5.3.4. The reason is that flushing the L1-D cache forces a write-back of all dirty lines, which means that the latency depends on the amount of dirty data, and thus on the execution history:

Requirement 4: Flush deterministically
State flushing mut be padded to its worst-case latency.

Interrupts could also be used for a covert channel, as we will demonstrate in Section 5.3.5. These are irrelevant to the cloud scenario, as there is no evidence that interrupts could be used as side channels, they are likely infeasible as an interrupt carries little data. Hence interrupt channels are only a concern intra-core.

Requirement 5: Partition interrupts
When sharing a core, the OS must disable or partition any interrupts other than the preemption timer.

Strategies for satisfying most of these requirements are well understood. We will now describe an approach that satisfies Requirement 2, Requirement 5 and simplifies Requirement 3 as a side effect. Remember from Section 1 that we are looking for mechanisms that are simple and policy-free, to make them suitable for a verifiable kernel.

3.3 Partitioning the OS: Cloning the kernel

Requirement 2 demands per-domain copies of the kernel. It would certainly be possible to structure a system at boot-image configuration time, such that each domain is given a separate kernel text segment, as in some NUMA systems [Concurrent Real Time 2012]. The domains would still share global kernel data, which then requires careful handling as per Requirement 3. The latter can be simplified by reducing the amount of shared global kernel data to a minimum, and replicate as much of it as possible between kernel instances, resulting in something resembling a multi-kernel [Baumann et al. 2009] on a single core, although more extreme in that kernel text is also separate.

This approach would imply completely static partitioning, where the configuration of domains, and thus the system's security policy, is baked into the boot image. As changes of policy would require changes to the kernel itself, this reduces the degree of assurance (or increases its cost). Furthermore, such a static approach would not suit the Cloud scenario: while the domain of a terminated VM could be recycled for a newly created one, the total number of VM slots would be fixed, forcing the system to over-provision domains just in case more might be needed.

We therefore favour an approach where *the kernel is ignorant of the specific security policy*, only one kernel configuration (which should eventually be completely verified) is ever used, the security policy is defined by the initial user process (as is the case with the present seL4 kernel), and where domains can be added or removed on demand.

We can achieve this by introducing a policy-free *kernel clone* mechanism. Its high-level description is creating a copy of a kernel image in user-supplied memory, including a stack and replicas of almost all global kernel data. The initial user process, serving as resource and security manager, can use kernel clone to set up an almost perfectly partitioned system. Specifically, the initial process separates all free memory into coloured pools, one per domain, clones a kernel for each partition into memory from the domain's pool, starts a child process in each pool, and associates the child with the corresponding kernel image.

The existing mechanisms of seL4 are sufficient to guarantee that the system will remain coloured for its lifetime, e.g. if init commits suicide. Cloning can be undone as long as a process with authority over a kernel image remains runnable. Re-partitioning is possible by moving memory colours between partitions or revoking a complete kernel image. Partitioning can be nested: a partition can sub-divide with new kernel clones, as long as it has sufficient Untyped memory and more than one page colour left.

4 Implementation in seL4

4.1 Kernel clone overview

In seL4, all access is controlled by capabilities. To control cloning, we introduce a new object type, Kernel_Image, which represents a kernel. A holder of a `clone` capability to a Kernel_Image object, with access to sufficient Untyped

memory, can clone the kernel. A Kernel_Image can be destroyed like any other object, and revoking a Kernel_Image capability destroys all kernels cloned from it.

We introduce another object type, Kernel_Memory, which represents physical memory that can be mapped to a kernel image, analogous to the existing Frame type, which represents memory that can be mapped into a user address space.

At boot time, the kernel creates a Kernel_Image master capability, which represents the present (and only) kernel and includes the `clone` right. It hands this capability, together with the size of the image, to the initial user process. That thread can then partition the system into security domains, by first partitioning its Untyped memory by colour. For each domain it clones a new kernel from the initial one, using some of the domain's memory pool, sets up an initial address space and thread in each of them, associates the threads with the respective kernels, and makes them runnable. The initial process can prevent other threads from cloning kernels by handing them only derived Kernel_Image capabilities with the `clone` right stripped.

Cloning consists of three steps. (1) The user process retypes some Untyped into an (uninitialised) Kernel_Image and Kernel_Memory of sufficient size, (2) it allocates an address space identifier (ASID) to the uninitialised Kernel_Image, (3) it invokes `Kernel_Clone` on the new Kernel_Image, passing an (existing) Kernel_Image capability with `clone` right and a Kernel_Memory capability as parameters, resulting in an initialised Kernel_Image.

Cloning copies the source kernel's code, read-only data (incl. interrupt vector table etc.) and stack. It also creates a new idle thread and a new kernel address space; the seL4 kernel has an address space that contains the kernel objects resulting from retype operations. This means that the Kernel_Image is represented as the root of the kernel's address space, plus an ASID. Any cloned Kernel_Image can independently handle any system calls, receive interrupts (Section 4.2) and system timer ticks, and run an idle thread when no user thread is runnable on a core. We add the capability of the kernel responsible for handling its system call to each thread's TCB.

Two kernels share only the minimum static data required for handing over the processor. On seL4, this is (numbers indicate size per core on x64, total of about 9.5 KiB):

1. the scheduler's array of head pointers to per-priority ready queues (4 KiB), as well as the bitmap used to find the highest-priority thread in constant time (32 B)
2. the current scheduling decision (8 B)
3. the tables of IRQ state interrupt handlers (2×1.1 KiB)
4. the interrupt currently being handled, if any (8 B)
5. the first-level hardware ASID table (1.1 KiB)
6. the IO port control table (2 KiB, x86 only)
7. the pointers for the current thread, its capability store (Cspace), the current kernel, idle thread, and the thread currently owning the floating point unit (40 B)
8. the kernel lock for SMP (8 B)
9. the barrier used for inter-processor interrupts (8 B).

We perform an audit of the shared data to ensure it cannot be used as a cross-core side channel. Specifically, we determine for all such data the circumstances (interrupt handling, context switch) under which the kernel will access it. We then establish that none of the cache lines involved contain or are accessed through private user information (such as address-space layout).

4.2 Partitioning interrupts

To support Requirement 5, we assign interrupt sources to a Kernel_Image. Interrupts (other than the kernel's preemption timer) are controlled by IRQ_Handler capabilities; the `Kernel_SetInt` system call allows associating an IRQ with a kernel. At any time, only the preemption timer and interrupts associated with the current Kernel_Image can be unmasked, this prevents kernels from triggering interrupts across partition boundaries.

Partitioning is policy, and the kernel will *not force* all IRQs to be partitioned. Associating an IRQ with multiple domains is valid but will leak; the kernel will only ensure that *partitioned* IRQs cannot leak.

4.3 Domain-switch actions

The running kernel is mostly unaware of domains; domain switches happen implicitly on a preemption interrupt. As the kernel is mapped at a fixed address in the virtual address space, the kernel (code and static data) switch happens implicitly when switching the page-directory pointer. Thus, the only explicit action needed for completing the kernel switch is switching the stack (after copying the present stack to the new one). The kernel detects the need for a stack switch by comparing the Kernel_Image reference in the destination thread's TCB with itself. In addition, the stack switch also implies actions for satisfying Requirements 1, 3, 4 and 5.

We flush all on-core microarchitectural state (Requirement 1) after switching stacks. The multicore version of seL4 presently uses a big lock for performance and verifiability [Peters et al. 2015]; we release the lock before flushing.

To reset on-core state on Arm, we flush the L1 caches (`DCCISW` and `ICIALLU`), TLBs (`TLBIALL`), and BP (`BPIALL`). On x86 we flush the TLBs (`invpcid`) and use the new *indirect branch control* (IBC) feature [Intel 2018b] for flushing the BP. Flushing the L1-D and -I caches presents a challenge on x86. While it has an instruction for flushing the complete cache hierarchy, `wbinvd`, it has no instruction for selectively flushing L1 caches. We therefore implement a "manual" flush: The kernel performs a load operation on one word per cache line of an L1-D-sized buffer. It flushes the L1-I cache by

following a sequence of jumps through a cache-sized buffer, which also indirectly flushes the branch target buffer (BTB).[6]

For addressing Requirement 4, an authorised thread (e.g. the cloner) may configure a switching latency. The kernel defers returning to user mode until the configured time is elapsed since the preemption interrupt. For policy-freedom we make this latency a user-controlled kernel-image attribute, as a safe value requires a worst-case execution time analysis, and the need for padding should be defined by the security policy. For example, with a hierarchical security policy such as Bell-LaPadula, flushing may not be needed when switching to a higher classification level. The padding latency is taken from the kernel active prior to the switch.

Satisfying Requirement 3 is much simplified by cloning, as the kernels share almost no data (Section 4.1). We achieve determinism by carefully prefetching all shared data before returning to userland, by touching each cache line. This will force the required data into the L1 cache, and ensure deterministic kernel exit. It is done just prior to the padding of the domain-switch latency, as the cost of loading these lines will depend their residency in lower-level caches. Prefetching is not needed for instructions, as kernel code is coloured.

To satisfy Requirement 5, we mask all interrupts before switching the kernel stack, and after switching unmask the ones associated with the new kernel. On x86, interrupts are controlled by a hierarchical interrupt routing structure, all the bottom-layer interrupts are eventually routed to the interrupt controllers on CPU cores. Because the kernel executes with interrupts disabled, there exists a race condition, where an interrupt is still accepted by the CPU just after the bottom-level IRQ source has been masked off. The kernel resolves this by probing any possible pending interrupts after masking, acknowledging them at the hardware level. Arm systems have a much simpler, single-level interrupt control mechanism, which avoids this race.

Timer-interrupt handling may be delayed due to another interrupt or system call occurring just before the preemption timer. We prevent this from delaying the domain switch by making the padding time long enough to allow for the worst-case handling time of such a system call or interrupt. A more sophisticated implementation would, if preemption is immediate, defer handling of the system call or interrupt until the next time slice of the same domain.

The steps performed by the kernel when handling a preemption tick are (bold steps are kernel-switch only):

1. acquire the kernel lock
2. process the timer tick normally
3. **mask interrupts**
4. **switch the kernel stack**
5. switch thread context (and implicitly the kernel image)
6. release the kernel lock
7. **unmask interrupts of the new kernel**
8. **flush on-core microarchitectural state**
9. **pre-fetch shared kernel data**
10. **poll the cycle counter for the configured latency**
11. reprogram the timer interrupt
12. restore the user stack pointer and return.

[6]This "manual" flush is dependent on assumptions on the hardware's (undocumented) line replacement policy, making it a brittle and potentially incomplete mechanism. Intel recently added support for flushing the L1-D cache [Intel 2018a]. However, we cannot use this feature, as a microcode update is yet to be available for our machine, and there is still no L1-I flush.

4.4 Kernel destruction

Destroying a kernel in a multicore system creates a race condition, as the kernel that is being destroyed may be active on other cores. For safe destruction, we first suspend all threads belonging to the target kernel. We support this with a per-kernel bitmap that indicates the cores on which it is presently running, it is updated during kernel switch.

During Kernel_Image destruction, the kernel first invalidates the target kernel capability (turning the kernel into a "zombie" object). It then triggers a `system_stall` event, which sends IPIs to all cores where the zombie is presently running; this is analogous to TLB shoot-down. The cores then schedule the idle thread belonging to the default Kernel_Image (created at boot time). Similarly, the kernel sends a `TLB_invalidate` IPI to all the cores that the target kernel had been running on. Lastly, the initial core completes the destruction and cleanup of the zombie.

Destroying active Kernel_Memory also invalidates the kernel, resulting in the same sequence of actions. Destroying either object invalidates the kernel, allowing the remaining object to be destroyed without complications.

The existence of an always runnable idle thread is a core invariant of seL4; we must maintain this in the face of dynamic kernel creation and destruction. We ensure the initial kernel image (and idle thread) remain, by not providing the initial kernel's Kernel_Memory capability to userland. This guarantees that there is still a kernel with an idle thread, even if userland destroys the last Kernel_Image. Such a system will have no user-level threads, and will do nothing more than acknowledging timer ticks.

A more sophisticated solution might allow reusing the initial kernel's memory, where the intention is to keep the system partitioned for its lifetime. This is hardly worthwhile, as the amount of dead memory is tiny: on x86 it is 216 KiB on a single core or 300 KiB on a 4-core machine, including the buffers for flushing the L1 caches. Corresponding Arm sizes are 120 KiB and 168 KiB.

5 Evaluation

We evaluate our approach in terms of its ability to close timing channels, as well as its effect on system performance.

5.1 Methodology

We quantify leakage using *mutual information* (MI) [Shannon 1948] as a measure of the size of a channel. We model the channel as a pipe into which the sender places *inputs*, drawn from some input set I (the secret values), and the receiver obtains *outputs* from some set O (the attacker's time measurements). In the case of a cache attack, the input could be the number of cache sets the sender accesses and the output is the time it takes the receiver to access a previously-cached buffer. MI indicates the average number of bits of information that a computationally unbounded receiver can learn from each input by observing the output.

We model the time measurements as a probability density function, meaning that we calculate the MI between discrete inputs and continuous outputs. If we treated the output time measurements as purely discrete then we would be treating all values as unordered and equivalent, e.g. a collection of unique particularly high values would not be treated differently from a collection of unique uniformly distributed values, therefore we might miss a leak. Furthermore, for a uniform input distribution, if continuous MI is zero then it implies that other similar measures, such as discrete capacity [Shannon 1948], are also zero. As it is an average function, rather than a maximum, MI is also easier to reliably estimate, making it an effective metric to see if a leak exists or not.

We send a large number of inputs and collect the corresponding outputs. From this we use kernel density estimation [Silverman 1986] to estimate the probability density function of outputs for each input. We then use the rectangle method (see e.g. [Hughes-Hallet et al. 2005] p. 340) to estimate the MI between a uniform distribution on inputs and the observed outputs, which we write as $\mathcal{M}$.

Sampling introduces noise, which will result in an apparent non-zero MI even when no channel exists. Sampled data can never prove that a leak does not exist, so instead we ask if the data collected contains any evidence of an information leak. Our present tool has a resolution of about 1 millibit, so cannot give conclusive evidence if $\mathcal{M} < 1$ mb, but such channels can be considered negligible anyway. If the estimated leakage is higher than this we use the following test [Chothia and Guha 2011; Chothia et al. 2013] to distinguish noise in the sampling process from a significant leak.

We simulate the measurement noise of a zero-leakage channel by shuffling the outputs in our dataset to randomly chosen inputs. This produces a dataset with the same range of values, but the random assignment ensures that there is no relation between the inputs and outputs (i.e., zero leakage). We calculate the MI from this new dataset and repeat 100 times, giving us 100 estimations from channels that are guaranteed to have zero leakage. From this we calculate the mean and standard deviation of these results, and then calculate the exact 95% confidence interval for an estimate to be compatible with zero leakage, which we write as $\mathcal{M}_0$.

If the estimate of MI from the original dataset is outside the 95% confidence interval, i.e. $\mathcal{M} > \mathcal{M}_0$, the observations are inconsistent with the MI being zero, and so there is a leak (the strict inequality is important here, because for very uniform data with no leakage $\mathcal{M}$ may equal $\mathcal{M}_0$). Otherwise we conclude that the dataset does not contain evidence of an information leak.

Unlike some prior work [Liu et al. 2015; Maurice et al. 2017], our aim is not to construct high capacity channels. Instead, we aim to demonstrate the existence or absence of a channel, so we usually choose unsophisticated encodings.

5.2 Hardware platforms

We conduct our experiments on representatives of the x86 and Arm architectures; Table 1 gives the details. Our Arm platform is somewhat dated, but we have not yet ported our time protection implementation to the Arm v8 architecture, which is used by the more recent cores. Furthermore, our earlier work demonstrated that recent (out-of-order) Arm cores contain microarchitectural state that cannot be scrubbed by architected means, and thus contain uncloseable high-bandwidth channels [Ge et al. 2018a].

We evaluate leakage in three scenarios: **raw** refers to the unmitigated channel while **protected** refers to our implementation of time protection, using two coloured domains with cloned kernels, each is allocated 50% of available colours unless stated otherwise.

For intra-core channels we additionally evaluate **full flush**, which performs a maximal architecture-supported reset of microarchitectural state. On Arm, this adds flushing the L2 cache to the flush operations used for time protection (as described in Section 4.3), and we also disable the BP and prefetcher for minimising uncontrollable microarchitecture state. On x86 the full flush scenario omits the "manual" L1 cache flush and instead flushes the whole cache hierarchy (`wbinvd`), and disables the data prefetcher by updating `MSR 0x1A4` [Viswanathan 2014].

Table 1. Hardware platforms.

System	Haswell (x86)	Sabre (Arm v7)
Microarchitecture	Haswell	Cortex A9
Processor/SoC	Core i7-4770	i.MX 6Q
Cores × threads	4 × 2	4 × 1
Clock	3.4 GHz	0.8 GHz
Cache line size	64 B	32 B
L1-D/L1-I cache	32 KiB, 8-way	32 KiB, 4-way
L2 cache	256 KiB, 8-way	1 MiB, 16-way
L3 cache	8 MiB, 16-way	N/A
I-TLB	64, 8-way	32, 1-way
D-TLB	64, 4-way	32, 1-way
L2-TLB	1024, 8-way	128, 2-way
RAM	16 GiB	1 GiB

Table 2. Worst-case cost of cache flushes (μs). The direct cost of the x86 L1 flush (bold) would be about 1 μs with a hardware-supported L1 I-cache flush.

Cache	x86			Arm		
	dir	ind	total	dir	ind	total
L1 only	**26**	1	27	20	25	45
Full flush	270	250	520	380	770	1,150

As a base line we measure the worst-case direct and indirect costs of flushing the (uncolourable) L1-I/D caches vs. the complete cache hierarchy. The direct cost is the combined latency of the flush instructions when all D-cache lines are dirty (or the cost of the "manual flush" on x86). We measure the indirect cost as the one-off slowdown experienced by an application whose working set is the size of the cache.

Table 2 shows results. The surprisingly high L1-flush cost on x86 is a result of our "manual" flush: the L1-D flush is <0.5 μs, the rest is for the L1-I, where each of the chained jumps (Section 4.3) is mis-predicted. Actual flush instructions should reduce the overall L1 flush cost to well below 1μs.

To put these figures into context, consider that cache flushes would only be required on a timer tick, which is typically in the order of 10–100 ms. Furthermore, the indirect cost of an L1 flush is irrelevant in practice: It would be highly unusual for a process to find any hot data in the L1 after another domain has been executing for a full time slice. We see from these results that flushing the L1 can be expected to add well below 1% overhead, while flushing the whole cache hierarchy will add substantial overheads.

5.3 Timing channel mitigation efficacy

To cover the attack scenarios listed in Section 3.1, we demonstrate a covert timing channel with a shared kernel image (Section 5.3.1), intra-core (Section 5.3.2) and inter-core (Section 5.3.3) timing channel benchmarks that exploit conflicts on all levels of caches, and a timing channel based on domain switching latency (Section 5.3.4).

5.3.1 Timing channel via a shared kernel image

As discussed in Section 2.4, partitioning user space automatically partitions dynamic kernel data (and will defeat e.g. page-table side-channel attacks [van Schaik et al. 2018]). We now demonstrate that it is insufficient for eliminating covert channels.

We implement an LLC covert channel between coloured user-space processes. The sender signals by triggering system calls, while the receiver, sharing the same core with a time slice of 1 ms, monitors the cache misses on the cache set that a kernel uses for serving the system calls.

The receiver firstly builds a probe buffer with the prime& probe technique [Liu et al. 2015; Osvik et al. 2006; Percival 2005]: it compares the cache misses on the probed cache sets before and after executing the system call, then marks a cache set as an attack set if the number of cache misses increase after the system returns.

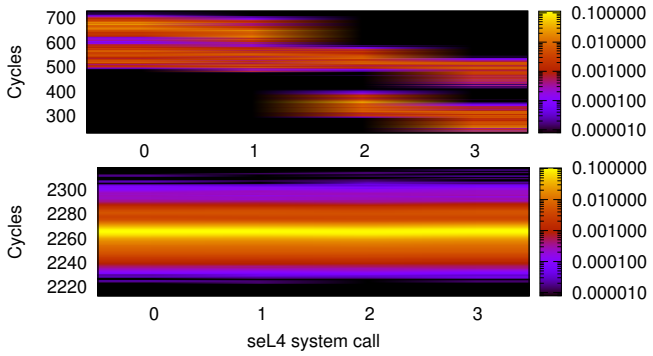

Figure 3. Kernel timing-channel matrix on x86, showing the conditional probability of output symbols (cycles) given an input symbol (system call). The top graph shows coloured userland only, where MI observed from a sample size of 255,790 is 0.79 bits, which we write as $\mathcal{M} = 0.79$ b, $n = 255,790$. The bottom graph is with full time protection, $\mathcal{M} = 0.6$ mb, $\mathcal{M}_0 = 0.1$ mb, $n = 255,040$ (1 mb = 10^{-3} bits).

The sender encodes a random sequence of symbols from the set $I = 0, 1, 2, 3$ by using three system calls: `Signal` for 0, `TCB_SetPriority` for 1, `Poll` for 2, and idling for 3. Figure 3 (top) shows the resulting *channel matrix*, i.e. the conditional probability of observing an output symbol given a particular input symbol, shown as a heat map. A channel is indicated by output symbols (cycles) being correlated with input symbols (system calls), i.e. variations of probability (colour) along horizontal lines. `Signal` and `TCB_SetPriority` lead to 500–700 cycles probing time, while `Poll` and idle result in 200–600 cycles, a clear channel. The MI of $\mathcal{M}$=0.79 bit per iteration (2 ms) means the channel can transmit 395 b/s.

With cloned kernels the channel disappears (bottom of Figure 3). The remaining channel is measured as $\mathcal{M} = 0.6$ millibits (mb), which is below the resolution of our tool and negligible. We implement a similar channel on the Arm, observing a non-trivial MI $\mathcal{M} = 20$ mb, which reduces to $\mathcal{M} = 0.0$ mb with time protection.

5.3.2 Intra-core timing channels

We investigate the full set of channels exploitable by processes time-sharing a core, targeting the L1-I, L1-D and L2 caches, the TLB, the BTB, and the branch history buffer (BHB). We use a prime&probe attack, where the receiver measures the timing on probing a defined number of cache sets or entries.

We use the Mastik [Yarom 2017] implementation of the **L1-D** cache channel, the output symbol is the time to perform the attack on every cache set. The **L2** channel is the same, with a probing set large enough to cover that cache. We

Table 3. Mutual information capacity $\mathcal{M}$ (in mb) of unmitigated (raw) intra-core timing channels, mitigated with full cache flush (full flush) and time protection (protected). $\mathcal{M}_0$ is the 95% confidence bound for a zero channel. Bold values represent a definite channel ($\mathcal{M} > \mathcal{M}_0$), others are consistent with no channel or below the 1 mb tool resolution.

Platform	Cache	Raw	Full flush		Protected	
		$\mathcal{M}$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$
x86	L1-D	**4,000**	0.5	0.5	0.6	0.6
	L1-I	**300**	0.7	0.8	0.8	0.5
	TLB	**2,300**	0.5	0.5	16.8	23.9
	BTB	**1,500**	0.8	0.8	0.4	0.4
	BHB	**1,000**	0.5	0.0	0.0	0.0
	L2	**2,700**	2.3	2.6	**50.5**	**3.7**
Arm	L1-D	**2,000**	1	1	30.2	39.7
	L1-I	**2,500**	1.3	1.3	4.9	5.2
	TLB	**600**	0.5	0.5	1.9	2.2
	BTB	**7.5**	4.1	4.4	62.2	73.5
	BHB	**1,000**	0	0.5	0.2	54.4

build the **L1-I** channel by having the sender probe with jump instructions that map to corresponding cache sets [Acıiçmez 2007; Acıiçmez et al. 2010]. For the **TLB** channel, the sender probes the TLB entries by reading an integer from a number of consecutive pages. We use chained branch instructions as the probing buffer for the **BTB** channel, the sender probing 3584–3712 branch instructions on Haswell, 0–512 on Sabre. Our **BHB** channel is the same as the residual state-based covert channel [Evtyushkin et al. 2016], where the sender sends information by either taking or skipping a conditional jump instruction. The receiver measures the latency on a similar conditional jump instruction, sensing any speculative execution caused by the sender's history.

Table 3 summarises results for the three scenarios defined in Section 5.2. The *raw* scenario shows a large channel in each case, except for the Arm BTB. On the Sabre we find that all channels are effectively closed by the *full flush* as well as the *protected scenario*.

On Haswell, the picture is the same, except for a residual L2 channel. While the full flush closes it, our implementation of time protection (which colours the L2) leaves a sizeable channel of 50 mb. Further investigation shows that the channel is decreased to $\mathcal{M} = 6.4$ mb ($\mathcal{M}_0 = 4.1$ mb) if we disable the aggressive data prefetcher – obviously this state machine is not reset by the mechanisms we have at our disposal for implementing time protection (manual L1-I and L1-D flush plus IBC). The remaining small channel likely results from the instruction prefetcher, which cannot be disabled.

This result is strong evidence for our previously-argued need of a better hardware-software contract for controlling any hidden microarchitecture state [Ge et al. 2018a].

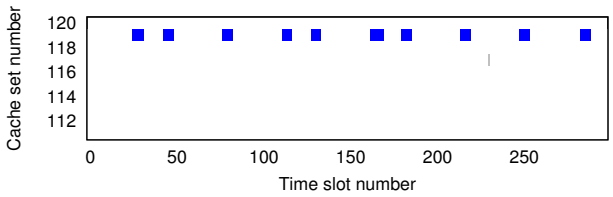

Figure 4. Unmitigated concurrent LLC side-channel attack on x86. The pattern at set 119 shows the victim's cache footprint detected by the spy.

5.3.3 Cross-core LLC channel

Recall that for cross-core attacks, our threat scenario only considers side channels. The only medium for these on both our evaluation platforms is the LLC. We evaluate mitigation for LLC-based cross-core side channels by reproducing the attack of Liu et al. [2015] on GnuPG version 1.4.13. The attack targets the square-and-multiply implementation of modular exponentiation used as part of the ElGamal decryption.

We use two processes, executing concurrently on separate cores on the Haswell. The victim repeatedly decrypts a file, whereas the spy uses the Mastik implementation of the LLC prime&probe attack to capture the victim's cache activity, searching for patterns that correspond to the use of the square function. The cache activity learned by the spy is shown in Figure 4. On cache set number 119, we see a sequence of dots separated by intervals of varying lengths. Each of these dots is an invocation of the square function and the secret key is encoded in the length of the intervals between the dots, with long intervals encoding ones and short intervals zeros. We find that time protection closes the channel (in this case by colouring the LLC), the spy can no longer detect any cache activity of the victim.

5.3.4 Cache-flush channel

To demonstrate the cache-flush channel we create a receiver that observes its progress by monitoring a cycle counter, waiting for a large jump in the counter value that indicates a preemption. *Online time* measures the uninterrupted period, while *offline time* is the length of the jump.

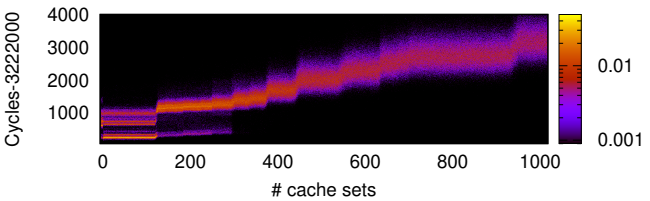

Figure 5. Unmitigated cache-flush channel, showing receiver-observed offline time vs. the sender's cache footprint on Arm. $\mathcal{M} = 1.4$ b, $n = 1828$

Table 4. Channel resulting from cache-flush latency (mb) without and with time protection.

Platform	Timing	No pad	Protected	
		$\mathcal{M}$	$\mathcal{M}$	$\mathcal{M}_0$
x86	Online	**8.4**	0.5	0.5
pad = 58.8 μs	Offline	**8.3**	0.6	0.6
Arm	Online	**1,400**	16.3	24.6
pad = 62.5 μs	Offline	**1,400**	210	237.2

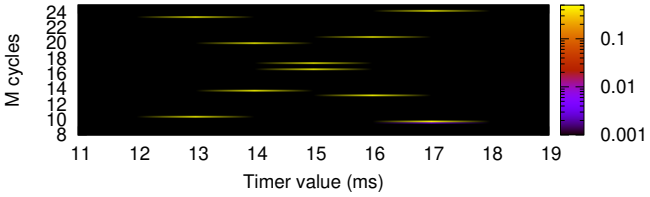

Figure 6. Unmitigated interrupt channel, showing receiver-observed online time vs. the timer interrupt configured by the Trojan, $\mathcal{M} = 902$ mb, $n = 10,860$. Benchmarked on the Haswell platform.

The sender, running on the same core, varies the number of cache sets it accesses in each time slice, manipulating the cost of the kernel's L1 cache flushes, and thus the receiver's online or offline time.

Figure 5 shows that the sender effectively modulates the offline time. Table 4 shows that the channel exists on both architectures, but is effectively closed with time padding.

5.3.5 Interrupt channel

We evaluate the interrupt partitioning with a timing channel based on a timer interrupt. The Trojan and spy execute on the same core, with a 10 ms system tick. For sending information, the Trojan programs the timer to fire after 13–17 ms and then sleeps for the rest of its time slice; this ensures that the timer fires approximately 3–7 ms into the spy's time slice. Figure 6 shows that the spy, which is identical to the one of Section 5.3.4, experiences two on-line periods per time slice, before and after the interrupt, resulting in a strong channel of 0.9 b per time slice. IRQ partitioning results in an uninterrupted time slice for the spy, and a closed channel ($\mathcal{M} = 0.5$ mb, $\mathcal{M}_0 = 0.7$ mb, $n = 11,029$).

5.4 Performance

5.4.1 IPC microbenchmarks

We evaluate the performance impact of time protection by measuring the cost of the most important (and highly optimised) microkernel operation, cross-address-space message-passing IPC. Table 5 summarises the results,[7] where *colour-ready* refers to a kernel supporting time protection without using it, intra-colour measures IPC that does not cross domains (kernels), while inter-colour does. The last is an artificial case that does not use a fixed time slice or time padding (which would defer IPC delivery to the partition switch). We use this to examine the baseline cost of our mechanisms. Standard deviations from 30 runs are less than 1%.

[7] Note that we are not using the mainline kernel, the cycle counts here are not comparable to what can be found on the seL4 web site.

Table 5. IPC microbenchmark performance and slowdown.

	x86		Arm	
Version	**Cycles**	**Slowd.**	**Cycles**	**Slowd.**
original	381	-	344	-
colour-ready	386	1%	391	14%
intra-colour	380	0%	395	15%
inter-colour	378	-1%	389	13%

We find that time protection adds negligible overhead on x86. On Arm, in contrast, there is a significant baseline cost for supporting the kernel clone mechanism. This is due to the fact that the baseline kernel uses global mappings to map the kernel's virtual address space. With multiple kernels, this is no longer possible. As the L2 TLB of the Cortex A9 processor, on which the Sabre is based, is only 2-way associative, these additional kernel mappings result in a significant increase of conflict misses on the cross-address-space IPC. There is no further overhead from using cloning.

Note that Arm v8 cores have 4-way associativity, so we expect this overhead to be significantly reduced on more recent cores.

5.4.2 Domain switch cost

In Table 2 we measured the worst-case cache-flush costs. We expect those to dominate the cost time protection adds to domain switches. We test this hypothesis by measuring the domain-switch latency (without padding) for a number of our attack workloads; specifically the time taken to switch from the receiver of a prime&probe attack to an idle domain. We report the mean for 320 runs, all standard deviations are less than 1% (Arm) or 3% (x86). An exception is the LLC test, where original seL4 times have a bimodal distribution and we report median values (standard deviation: 25% for Arm, 18% for x86).

Table 6. Absolute cost (μs) with no padding of switching away from a domain running various receivers from Section 5.3.2.

Platf.	Mode	Idle	L1-D	L1-I	L2	L3
x86	Raw	0.18	0.19	0.22	0.23	0.5
	Full flush	271	271	271	271	271
	Protected	30	30	30	30	30
Arm	Raw	0.7	0.8	1.2	1.6	N/A
	Full flush	414	414	414	414	N/A
	Protected	27	27	27	31	N/A

Table 6 shows the results for our three defence scenarios. We observe first that the workload dependence of the latency evident in the raw system has mostly vanished from the defended systems, even without padding. These benchmarks establish a lower bound on the safe padding time. We secondly notice that, as expected, the *full flush* latencies match the flush costs of Table 2. With time protection, the switch latency is slightly higher than the direct L1-flush cost of Table 2, confirming our hypothesis that this is the dominant cost, and also supporting the comment in Section 5.2 that indirect flush cost are of little relevance for L1 caches.

Most importantly, the results show that our implementation of time protection imposes significantly less overhead than the full flush, despite being as effective in removing timing channels (except for the issues resulting from the lack of targeted cache flushes discussed in Section 5.3.2). Looking at these numbers in the context of a 10 ms time slice, we can see that the relative overhead of a full flush would be about 3% in x86 and 4% on the Arm, while for our implementation of time protection it is only about 0.3% on both processors.

5.4.3 Kernel cloning and destruction cost

Table 7 shows the cost of cloning and destroying kernel images. We can see that the clone cost is a fraction of that of creating a process in Linux on the same hardware, while destroying a kernel is 1–2 orders of magnitude faster still.

5.4.4 The cost of cache colouring

Cache partitioning (through colouring) as we use it for implementing time protection replaces the dynamic partitioning done by hardware by a static (although OS-changeable) partitioning. This can be expected to lead to somewhat less optimal use of the cache and thus a performance cost, a well-understood tradeoff. However, static partitioning also leads to more predictable performance, which was the original motivation for it [Kessler and Hill 1992; Liedtke et al. 1997; Lynch et al. 1992]. More recently, cache colouring has also been proposed as a way for *improving* performance [Han et al. 2018; Noll et al. 2018].

Nevertheless, using it as part of mandatory security enforcement will in average lead to some performance degradation, particularly if the cache is shared between an application with a large and one with a small footprint. Here we will try to get an idea for the size of this effect, by running a single benchmark with a reduced (to 75% or 50%) cache size.

Table 7. Cost of cloning (μs) vs. Linux process creation.

	seL4		Linux
Arch	**clone**	**destroy**	**fork+exec**
x86	79	10	257
Arm	608	116	4,300

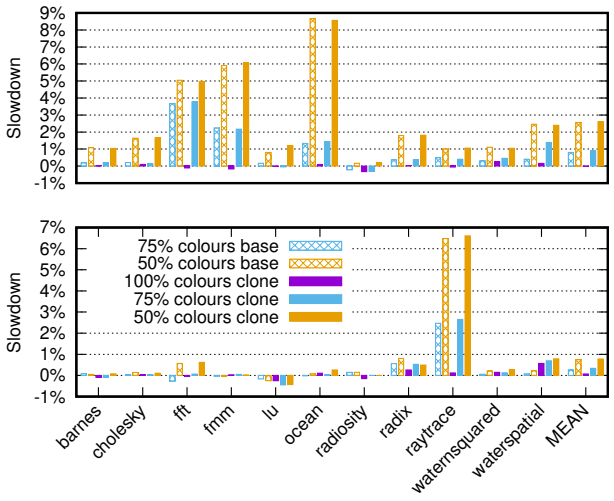

Figure 7. Slowdowns of Splash-2 benchmarks against baseline kernel without partitioning for x86 (top) and Arm and geometric mean. Benchmarks are run as the only process on the system. The "base" cases use the standard kernel with reduced cache, the "cloned" cases run the benchmark on a cloned kernel with the "100% colours" case using an unpartitioned cache like the baseline.

Our seL4 kernel with time protection is a research prototype, which lacks a Posix-like environment that is expected by most benchmarks, such as SPEC. As an easily portable benchmark suite we use Splash-2 [Woo et al. 1995]. These are obviously quite dated, but for our purposes all we need is something that exercises the LLC. We set the running parameters to consume 220 MiB of heap and 1 MiB of stack. We omit the *volrend* program due to its Linux dependencies.

Figure 7 shows the slowdown resulting from cache colouring and kernel cloning. We report the mean of 10 repeated single-threaded runs (standard deviations are below 3%), as well as the geometric mean across the suite. The benchmarking thread is the only user thread in the system.

On the Arm, cache colouring introduces less than 1% slowdown for benchmarks, except *raytrace*, which shows a 6.5% slowdown when executing with 50% of the cache, as this benchmark has a large cache working set. However, given a 75% cache share, the slowdown drops to 2.5%. On top of this, running on a cloned kernel adds almost no performance penalty, except on *waterspatial*, where it is still below 0.5%.

On Haswell, we observe slightly larger performance overheads, as we partition based on colours of the relatively small (256 KiB) L2 cache (which implicitly colours the LLC). The alternative would be to only colour the LLC and flush the L2, at the expense of increased domain-switching cost. With no targeted L2 flush supported by the architecture, this seems not worthwhile. Still, the majority of the Splash-2 tests only slow down by less than 2%. Increasing cache share to 75%

Table 8. Performance impact on Splash-2 of time protection with 50% colours, including the increased context-switch latency in a time-shared setup, with and without padding. On x86, overhead is highest (Max) on ocean and lowest (Min) on raytrace, on Arm Max is Raytrace, Min is Radix.

	x86			Arm		
Pad	**Max**	**Min**	**Mean**	**Max**	**Min**	**Mean**
no	10.96%	0.26%	2.76%	6.73%	-2.88%	0.75%
yes	11.06%	0.86%	3.38%	7.11%	-2.55%	1.09%

limits the overhead to below 3.5%. As for Arm, kernel cloning adds close to zero overhead.

5.4.5 The impact of domain switches

The above evaluation of partitioning and cloned kernel images does not show the effect of the increased context-switching cost resulting from flushing on-core state. To evaluate we rerun the Splash-2 benchmarks, now time-sharing the processor with an idle domain. This measures the effective reduction of CPU bandwidth from the increased context-switch latency.

Table 8 shows the result for both architectures. Specifically we show the benchmarks with the highest and lowest overhead as well as the geometric mean across the suite. Without padding, the additional context-switch cost is minimal, indicated by the mean being only slightly higher than in Figure 7. Padding adds very little on top of that, 0.5% on x86 and 0.3% on Arm. This is consistent with our expectations, see the discussion of Table 2 in Section 5.2.

6 Discussion

6.1 Strengths and limitations of time protection

The evaluation shows that our implementation of time protection in seL4 is generally highly effective, and low-cost. In particular, there is negligible cost of running on a cloned kernel (except on our Arm v7 processor with its low TLB associativity), and so are memory overheads, of the order of 100 KiB per kernel image and core. Kernel image creation and destruction is fully dynamic and cheap, compared to the cost of creating Linux processes (leave alone whole VMs).

Also, time protection consists of a suite of mechanisms that remove interference on different classes of resources; the threat model determines which are needed. E.g. in a Cloud scenario, where side channels but not covert channels are an issue, the most expensive operation, padding of domain-switches to their worst-case latency, may not be needed.

Just as with a multikernel, the separate kernel images do not prevent sharing of user-level state, if allowed by the security policy. E.g. shared memory can be set up with a dedicated colour; the resulting channel would need to be prevented by ensuring deterministic user-level access.

Re-allocating memory between security domains is possible in principle, and could be supported by ballooning [Waldspurger 2002], but the granularity would have to be that of a full cache colour, making it fairly expensive. This is a inevitable consequence of lack of hardware support for more fine-grained cache partitioning; if better support becomes available, time protection can make use of it.

Time protection is obviously at the mercy of hardware, and not all hardware provides sufficient support for full temporal isolation. We have seen this with the x86 L2 channel in Table 3, which we could not close. We traced this to the prefetcher, which retains state we cannot flush without a (prohibitively expensive) flush of the full cache hierarchy. Prior to Intel adding the IBC feature in a recent microcode update, the situation was much worse [Ge et al. 2018a], which indicates that Intel could easily do more. Our earlier work also shows that recent Arm processors have similar problems, but as the Arm ISA is not microcoded, it may be impossible to fix those security holes on existing processors.

The results reinforce the need for a new, security-oriented hardware-software contract. We have specified the requirement on such a contract in detail [Ge et al. 2018a], but it can be summarised as requiring that:

- the OS must be able to partition or flush any shared hardware resource
- concurrently-accessed resources must be partitioned
- virtually-addressed state must be flushed.

The resource for which contemporary hardware most obviously fails to satisfy this contract is the interconnect (buses), which cannot be partitioned – this is the reason why we had to omit cross-core covert channels from our threat model. While we have argued that these are not relevant in some important scenarios (see Section 3.1.2), if would clearly be desirable to apply time protection more generally. For example requiring single-core execution when confining JavaScript code in a browser is very restrictive. Alas, we are powerless without appropriate hardware support.

The number of available colours is a potential bottleneck, especially in a Cloud scenario. Note that in this case only the LLC needs to be coloured, which has more colours than the private L2 (32 vs. 8 colours on our Haswell). Furthermore, the hashing scheme used on the distributed shared LLC on recent Intel processors increases the number of colours over that resulting from cache associativity alone [Yarom et al. 2015]. But there is potential for further hardware support.

6.2 Time protection in other systems

There is no reason time protection cannot be implemented in other systems, although seL4's design-for-isolation approach simplifies many things, in particular partitioning kernel data. Systems like Linux have a kernel heap and far more static global data, all of which must be partitioned or deterministic access enforced. While this is probably challenging to do,

there is no fundamental reason why it could not be done. Cloning a large kernel, such as Linux, will also be more expensive, yet another argument in favor of a microkernel design. A small-ish (although still large compared to seL4) hypervisor such as Xen is probably an easier target.

7 Related Work

Deterministic systems eliminate timing channels by providing only virtual time; Determinator [Aviram et al. 2010] is an example aimed at clouds. Ford [2012] extends this model with scheduled IO. Stopwatch [Li et al. 2013] virtualises time by running three replicas of a system, then only announces externally-visible timing events at the median of the times determined by the replicas. The system is effective but at a heavy performance penalty.

Kessler and Hill [1992] and Bershad et al. [1994] proposed page colouring for performance isolation. Liedtke et al. [1997] proposed the same for improved real-time predictability, while Shi et al. [2011] proposed dynamic page colouring for mitigating attacks against cryptographic algorithms in the hypervisor. StealthMem [Kim et al. 2012] uses colouring to provide some safe storage with controlled cache residency. CATalyst [Liu et al. 2016] uses Intel's CAT technology for LLC partitioning for a similar purpose.

Percival [2005] proposed hardware-supported partitioning of the L1 cache, while Wang and Lee [2007] suggested hardware mechanisms for locking cache lines, called a partition-locked cache (PLcache).

Spectre, Meltdown and Foreshadow (L1TF) [Kocher et al. 2019; Lipp et al. 2018; Van Bulck et al. 2018; Weisse et al. 2018] exploit covert channels to exfiltrate information from speculatively executed instructions. Among the countermeasures for these attacks, Intel introduced instructions for flushing the branch predictor [Intel 2018b] and the L1-D cache [Intel 2018a]; we use these in our implementation. Other countermeasures for these attacks are mostly orthogonal to our work. Our design is effective at preventing cross-security-domain Spectre attacks, the other attacks can only be prevented with (forthcoming) improvements to the hardware.

The idea of using multiple kernel images on a single system has been proposed in the past for supporting multi-core platforms. Corey [Boyd-Wickizer et al. 2008] enhances many-core scalability by letting the application control the degree of sharing of kernel data structures. Helios [Nightingale et al. 2009] uses satellite kernels with a common API for seamlessly supporting heterogeneous computing elements. Multikernels [Baumann et al. 2009] run per-core, shared-nothing kernel images. Barrellfish/DS [Zellweger et al. 2014] separates OS kernel images from physical CPU cores, to support hot-plugging and energy management.

8 Conclusions

We proposed, implemented and evaluated *time protection*, a mandatory, black-box kernel mechanism for preventing microarchitectural timing channels. It employs a combination of partitioning and flushing of shared hardware. We eliminate channels through a shared kernel image by a policy-free *kernel clone* mechanism that allows almost complete partitioning of the kernel. It allows constructing a system that runs a separate kernel for each security domain, and also supports partitioning of interrupts, to completely prevent any cross-domain temporal interference.

Our evaluation shows that the mechanisms are effective in closing all examined timing channels, while imposing small to negligible overhead. However, we also observe that present hardware has significant shortcomings in its support for preventing interference. This finding strongly supports our earlier claim that the ISA is an insufficient hardware-software contract for providing true security, and that we need an improved, security-oriented contract [Ge et al. 2018a]. Ideally this contract will also support partitioning interconnect channels (see Section 2.3), allowing time protection to prevent inter-core covert channels (see Section 3.1).

Despite those hardware-inflicted limitations, we claim that the concept of time protection is general and overdue. Implementations will be able to adapt as hardware improves, and provide better security to a growing class of use cases.

Time protection comprises a suite of kernel mechanisms. Proper integration into the seL4 API is future work, especially combining it with the recently added temporal integrity mechanisms [Lyons et al. 2018]. Our ultimate aim is a verified seL4 with time protection. We have some ideas on how to achieve this [Heiser et al. 2019] but these are rather speculative at this point.

Acknowledgments

This research was supported in part by Intel Corporation, by a Google Faculty Award, a Google PhD Fellowship and by the Australian Department of Defence's Next Generation Technology Fund.

Availability

The raw datasets as well as the toolchain for calculating $\mathcal{M}$ are available for download, see https://ts.data61.csiro.au/projects/TS/timingchannels/ for details.

References

Onur Acıiçmez. 2007. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*. ACM, Fairfax, VA, US, 11–18.

Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Workshop on Cryptographic Hardware and Embedded Systems*. Springer, Santa Barbara, CA, US, 110–124.

Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *11th IMA International Conference on Cryptography and Coding*. Springer, Cirencester, UK, 185–203.

Onur Acıiçmez and Jean-Pierre Seifert. 2007. Cheap Hardware Parallelism Implies Cheap Security. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Vienna, AT, 80–91.

ARM Ltd. 2008. *ARM Architecture Reference Manual, ARM v7-A and ARM v7-R*. ARM Ltd. ARM DDI 0406B.

Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. 2010. Determinating timing channels in compute clouds. In *ACM Workshop on Cloud Computing Security*. IEEE, Chicago, IL, US, 103–108.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, US, 29–44.

Brian N. Bershad, D. Lee, Theodore H. Romer, and J. Bradley Chen. 1994. 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*. ACM, San Jose, CA, US, 158–170.

Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. 1992. The KeyKOS Nanokernel Architecture. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*. USENIX Association, Seattle, WA, US, 95–112.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, San Diego, CA, US, 43–57.

Tom Chothia and Apratim Guha. 2011. A Statistical Test for Information Leaks Using Continuous Mutual Information. In *IEEE Computer Security Foundations Symposium*. IEEE, Cernay-la-Ville, FR, 177–190.

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *International Conference on Computer Aided Verification*. ACM, Saint Petersburg, RU, 690–695.

Patrick J. Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Istambul, TK, 177–189.

Concurrent Real Time 2012. *An Overview of Kernel Text Page Replication in RedHawk Linux 6.3*. Concurrent Real Time.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9 (1966), 143–155.

Department of Defence 1986. *Trusted Computer System Evaluation Criteria*. Department of Defence. DoD 5200.28-STD.

Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. USENIX, Washington, DC, US, 431–446.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and Mitigating Covert Channels Through Branch Predictors. *ACM Transactions on Architecture and Code Optimization* 13, 1 (April 2016), 10.

Bryan Ford. 2012. Plugging side-channel leaks with timing information flow control. In *Proceedings of the 4th USENIX Workschop on Hot Topics in Cloud Computing*. USENIX, Boston, MA, USA, 1–5.

Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018b. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* 8 (April 2018), 1–27.

Qian Ge, Yuval Yarom, and Gernot Heiser. 2018a. No Security Without Time Protection: We Need a New Hardware-Software Contract. In *Asia-Pacific Workshop on Systems (APSys)*. ACM SIGOPS, Korea, 9.

Ben Gras, Kaveh Razavi, Herbert Bos, and Christiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of the 27th USENIX Security Symposium*. USENIX, Baltimore, MD, US, 955–972.

David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, US, 490–505.

Myeonggyun Han, Seongdae Yu, and Woongki Baek. 2018. Secure and Dynamic Core and Cache Partitioning for Safe and Efficient Server Consolidation. In *International Symposium on Cluster Computing and the Grid*. IEEE, Washington, DC, US, 311–320.

Gernot Heiser, Toby Murray, and Gerwin Klein. 2019. Can We Prove Time Protection? https://arxiv.org/pdf/1901.08338.pdf. *arXiv preprint arXiv:1901.08338* (Jan. 2019), 6.

Wei-Ming Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, Oakland, CA, US, 8–20.

Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, US, 52–61.

Deborah Hughes-Hallet, Andrew M. Gleason, Guadalupe I. Lonzano, et al. 2005. *Calculus: Single and Multivariable* (4 ed.). Wiley, New York, NY, US.

Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, 191–205.

Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Workshop on Cryptographic Hardware and Embedded Systems*. Springer, Santa Barbara, CA, US, 368–390.

Intel. 2018a. Deep Dive: Intel Analysis of L1 Terminal Fault. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault

Intel. 2018b. Speculative Execution Side Channel Mitigations. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

Intel Corporation 2016. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*. Intel Corporation. http://www.intel.com.au/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. 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*. IEEE, San Jose, CA, US, 591–604.

R. E. Kessler and Mark D. Hill. 1992. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems* 10 (1992), 338–359.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, Bellevue, WA, US, 189–204.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Haburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, 19–37.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*. Springer, Berkeley, CA, US, 564–580.

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16 (1973), 613–615.

Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. 1975. Policy/Mechanism Separation in HYDRA. In *ACM Symposium on Operating Systems Principles*. ACM, Austin, TX, US, 132–140.

Peng Li, Debin Gao, and Michael K Reiter. 2013. Mitigating access-driven timing channels in clouds using StopWatch. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*. IEEE, Budapest, HU, 1–12.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Montreal, CA, 213–223.

Moritz Lipp, Michael Schwartz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*. USENIX, Baltimore, MD, USA, –.

Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE Symposium on High-Performance Computer Architecture*. IEEE, Barcelona, Spain, 406–418.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, US, 605–622.

William L. Lynch, Brian K. Bray, and M. J. Flynn. 1992. The effect of page allocation on caches. In *ACM/IEE International Symposium on Microarchitecture*. IEEE, Portland, OR, US, 222–225.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-Context Capabilities: A Principled, Light-Weight OS Mechanism for Managing Time. In *EuroSys Conference*. ACM, Porto, Portugal, 14.

Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. 2010. Security best practices for developing Windows Azure applications. https://docs.microsoft.com/en-us/azure/security/security-best-practices-and-patterns

Clémentine Maurice, Manuel Weber, Michael Schwartz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*. USENIX, San Diego, CA, US, 15.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, 415–429.

Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, US, 221–234.

Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2018. Accelerating Concurrent Workloads with CPU Cache Partitioning. In *International Conference on Data Engineering*. IEEE, Paris, FR, 437–448.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*. Springer, San Jose, CA, US, 1–20.

Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan 2005*. Ottawa, CA, 13. http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Asia-Pacific Workshop on Systems (APSys)*. ACM, Tokyo, JP, 7.

Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. 1977. Program Confinement in KVM/370. In *Proceedings of the Annual ACM Conference*. ACM, Atlanta, GA, US, 404–410.

Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *ACM Symposium on Operating Systems Principles*. ACM, Charleston, SC, USA, 170–185.

Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, HK, 194–199.

Bernard W. Silverman. 1986. *Density estimation for statistics and data analysis*. Chapman & Hall, London, UK.

Jo Van Bulck, Marina Minkin, Ofir Weiss, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Stracks. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*. USENIX, Baltimore, 991–1008.

Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *Proceedings of the 27th USENIX Security Symposium*. USENIX, Baltimore, MD, US, 937–954.

Vish Viswanathan. 2014. Disclosure of H/W Prefetcher Control on some Intel Processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

VMware Knowledge Base. 2014. 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.

Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Boston, MA, US, 181–194.

Yao Wang and G Edward Suh. 2012. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th ACM/IEEE International Symposium on Networks on Chip*. IEEE, Lyngby, Denmark, 142–151.

Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*. ACM, San Diego, CA, US, 494–505.

Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*. ACM, S. Margherita Ligure, IT, 24–36.

John C. Wray. 1991. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, Oakland, CA, US, 2–7.

Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, Bellevue, WA, US, 159–173.

Yuval Yarom. 2017. Mastik: A Micro-Architectural Side-Channel Toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf.

Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, San Diego, CA, US, 719–732.

Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. http://eprint.iacr.org/.

Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Conference on Cryptographic Hardware and Embedded Systems 2016 (CHES 2016)*. Springer, Santa Barbara, CA, US, 346–367.

Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Philadelphia, PA, US, 55–64.

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. 2014. Decoupling Cores, Kernels, and Operating Systems. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Broomfield, CO, US, 17–31.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*. ACM, Raleigh, NC, US, 305–316.