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* in addition to the established memory protection. We examine the requirements of time protection, present a design and its implementation in the seL4 microkernel, and evaluate its efficacy as well as performance overhead on Arm and x86 processors.

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 its control on cross-domain communication.

Programs, however, can bypass OS protection by sending information over channels that are not intended for communication. Historically, such *covert channels* were explored within the context of military multi-level-secure systems [Department of Defence 1986]. Cloud computing, smartphone apps and server-provided JavaScript executed in browsers 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 of 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 can thus be considered a solved problem.

The same cannot be said about temporal isolation. *Timing channels*, and in particular *microarchitectural channels* [Ge et al. 2018b], which exploit timing variations due to shared use of the 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 [Irazoqui et al. 2015; Liu et al. 2015] through *side channels*, i.e. without the cooperation of the key owner.

In contrast, covert channels depend on insider help and are traditionally considered a less significant threat. However, in the recent Spectre attack [Kocher et al. 2019], an adversary uses a covert communication channel 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, covert channel mechanisms bear 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 *time protection* a primary OS duty, just as the established memory protection.[1] Such a design must eliminate, as far as possible, the sharing of hardware resources that is the underlying cause of timing channels. The ultimate aim should be to obtain temporal isolation guarantees comparable to the spatial isolation proofs of seL4, but for now we focus on a *design* that is suitable for a verifiable OS kernel, i.e. minimal, general and policy-free.

Specifically, we make the following contributions:

- We define the requirements for providing time protection, enabling confinement in the presence of microarchitectural timing channels (Section 3.2);
- we introduce a policy-free *kernel clone* operation that allows almost perfect partitioning of a system, i.e. almost completely removing sharing between security domains (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);

[1] Note that we use the term "OS" in a generic sense, referring to the most privileged software level that is responsible for security enforcement. This could refer to a hypervisor or the OS of a non-virtualised system.

- we show that the overhead imposed by these mechanisms is low (Section 5.4).

2 BACKGRN

2.1 Cbll

A covert channel is an information flow that uses a mechanism not intended for information transfer [Lampson 1973]. Covert channels therefore may violate the system's security policy, allowing communication between security domains that should be isolated. Note that our use of the term (security) domain is more general than the standard OS term protection domain (a specific set of access rights). A security domain consists of one or more protection domains.

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 recent progress on proving upper bounds for the cache side channels of cryptographic implementations [Doychev et al. 2013; Köpf et al. 2012], proofs of complete elimination of timing channels in a non-trivial system are beyond current formal approaches, and 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*, who, 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, who is being attacked by a malicious co-resident VM [İnci et al. 2016; Yarom and Falkner 2014].

2.2 M

Microarchitectural timing channels result from competition for capacity- or bandwidth-limited hardware features that are functionally transparent to software [Ge et al. 2018b].

Capacity-limited resources include the data and instruction caches, these can be used to establish high-bandwidth channels [Hu 1992; Liu et al. 2015; Maurice et al. 2017]. However, other microarchitectural state, such as the translation lookasid buffer (TLB), branch predictor (BP) or prefetcher state machines, can be used as well. Fundamentally, the cache channel works 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. Such attacks are possible where the resource is shared concurrently (cores or hardware threads sharing a cache) or time-multiplexed (time-sharing a core).

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 last-level cache (LLC) [Irazoqui et al. 2015; Liu et al. 2015], the TLB [Gras et al. 2018; Hund et al. 2013] and the BP [Acıiçmez et al. 2007].

Interconnects of limited bandwidth can also be used for covert channels: the sender encodes information into its bandwidth consumption, and the receiver senses the available bandwidth. So far, interconnects can only be exploited as a covert channel while the sender and receiver execute concurrently (on different cores). Also, if only bandwidth can be used for signalling, side channels are probably impossible to implement, none have been demonstrated to date.

2.3 Cn

The countermeasures must prevent interference resulting from resource competition while processing secret information. For bandwidth-limited interconnects, this would require time-multiplexing the interconnect or using some hardware mechanism to partition available bandwidth.[3]

The OS can prevent interference on stateful resources by flushing between accesses or by partitioning.[4]

Flushing is conceptually simple (although can be difficult in practice, as we will discuss in Section 4.3). It is only applicable for time-multiplexed hardware; flushing cannot prevent cross-core attacks through a shared cache. Flushing can also be very costly in the case of large caches (LLC), as we demonstrate in Section 5.2.

[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.

[3] Intel recently introduced *memory bandwidth allocation* (MBA) technology, which imposes *approximate* limits on the memory bandwidth available to a core [Intel Corporation 2016]. This is a step towards bandwidth partitioning, but the approximate enforcement is not sufficient for preventing covert channels.

[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.

Partitioning by the OS is only possible where the OS has control over how domains access the shared infrastructure. This is the case in physically-indexed caches (generally the $L2 \cdots LLC$), as the OS controls the allocation of physical memory frames to domains, and thus the physical addresses. 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 part of the cache, referred to as the "colour" of the page. With a page size of P, a cache of size S and associativity w has S/wP colours. Therefore, the OS can partition the physically-indexed cache with coloured frames. By building domains with disjoint colours, the OS can prevent them competing for the same cache lines [Kessler and Hill 1992; Liedtke et al. 1997; Lynch et al. 1992].

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 the other on-core state, such as the TLB and BP. Hence, *if domains share a core, 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 against side channels, we believe the time protection, like memory protection, should be a *mandatory* (black-box) OS security enforcement mechanism, rather than depending on application cooperation. Only mandatory enforcement can support confinement.

2.4 L4

seL4 is a microkernel designed for use in security- and safety-critical systems. It features 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 many other security-oriented systems [Bomberger et al. 1992; Shapiro et al. 1999], seL4 uses capabilities [Dennis and Van Horn 1966] for access control: access to any object 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, dubbed the *root task*, 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 provided to the kernel by the usermode process which creates the need for such data. For example, if a process wants to create a new thread, it not only has to provide memory for that thread's stack, but it also 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 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 aids isolation. For example, the root task 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

3 ATTACKS AND DEFENCES

3.1

We aim to develop general time-protection mechanisms suitable for a wide range of use cases. To represent these, we pick threat scenarios from opposite ends of the spectrum (summarised in Figure 1). If we can satisfy both, we should be able to address many other cases as well.

3.1.1 Confinement. In this scenario, untrusted (malicious) code attempts to exfiltrate sensitive data which it processes.

This could represent untrusted code (such as a unverified library, third-party app or web browser plugin) which has access to sensitive personal data, or the gadget in a Spectre attack. An underlying assumption in these examples is that the code cannot be trusted not to leak information, hence the OS's time protection must prevent leakage through a microarchitectural channel.

In this scenario we 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

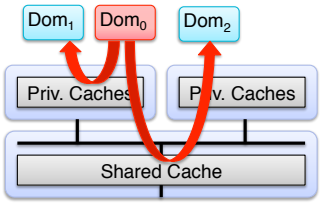


Fig 1: [illegible]

time only one domain executes. We require this restriction to prevent the use of the memory bus as a high-bandwidth channel [Hu 1991; Wu et al. 2012], addressing which is outside the scope of this work and likely impossible with current hardware (Intel MBA notwithstanding).

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 not of much relevance. Instead we aim to prevent side channels, where an attacking VM is trying to infer secrets held by a victim VM. Recent work demonstrated the feasibility of cross-core and cross-processor side channel attacks through the LLC [İnci et al. 2016; Irazoqui et al. 2015, 2016; Liu et al. 2015]. No side-channel attacks on the memory bus are known to date [Ge et al. 2018b] and they are probably infeasible.[5]

Multiple attacks exploiting concurrent execution within a core have been demonstrated [Acıiçmez and Seifert 2007; Percival 2005; Yarom et al. 2016] and hypervisor providers advise against sharing cores between VMs [Zhang et al. 2012]. The high level of resource sharing between hyperthreads prevents spatial partitioning. We therefore assume that hyperthreading is either disabled or that all hyperthreads of a core belong to the same VM. 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.

[5]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.

3.2 [illegible]

To address the threats above, we propose a combination of techniques for spatially partitioning concurrently shared resources, and for flushing 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.

> **[illegible]: Flush**
> When time-sharing a core, the OS must flush on-core microarchitectural state on partition switch.

Other core-private caches, such as the (physically addressed) L2 in Intel processors, could be flushed or partitioned. Hardware resources shared between cores, in particular the LLC, must be partitioned by the OS, as flushing introduces too much overhead (see Section 5.2) and cannot prevent timing channels in our cloud scenario.

Colouring rules out sharing of physical frames between partitions, 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, we will demonstrate this in Section 5.3.1.

> **[illegible]**
> Each domain must have its private copy of kernel text, stack and (as much as possible) global data.

As discussed in Section 2.4, partitioning most kernel data is straightforward in seL4: partitioning all user memory automatically partitions all dynamically allocated (i.e. user-provided) kernel memory as well. Hence, colouring user memory will colour all dynamically allocated kernel data structures. Significantly more work would be required to implement such partitioning in other systems, but there is no fundamental reason why it could not be done. This leaves a (in seL4 small) amount of global kernel data uncoloured.

> **[illegible]: Det[illegible]**
> Access to remaining shared kernel data must be deterministic enough to prevent its use for timing channels.

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:

[illegible]
The kernel must pad cache flushing to its worst-case latency.

Interrupts could also be used for a covert channel (although the bandwidth would be low, as this could not signal more than a few bits per partition switch). They are irrelevant to the cloud scenario, as there is no evidence that interrupts could be used as side channels.

[illegible]
When sharing a core, the kernel 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 [illegible]

Requirement 2 demands per-partition copies of the kernel. It would certainly be possible to structure a system at boot-image configuration time, such that each partition is given a separate kernel text segment, as in some NUMA systems [Concurrent Real Time 2012]. The partitions 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.

The drawback of this approach is that it would lead to a completely static partitioning, where the configuration of partitions, and thus the system's security policy, is baked into the kernel boot image. As changes of policy would require changes to the kernel itself, this reduces the degree of assurance (or increases its cost). Especially in the case of seL4, the initialisation code would have to be re-verified for each configuration, or assurance lost.

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, and the security policy is defined by the initial user process, just as with the present seL4 kernel.

We introduce 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 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 partition, clones a kernel for each partition into memory from the partition's pool, starts a child process in each pool, and associates the child with the corresponding kernel image, and then commits suicide, resulting in a completely and permanently coloured system.

The existing mechanisms of seL4 are sufficient to guarantee that such a system will remain coloured for its lifetime. Furthermore, the process can be repeated: a partition can further sub-divide itself with new kernel clones, as long as it has sufficient Untyped memory and more than one page colour left. Partitioning can also be reverted (assuming that the process that created it remains runnable).

4 [illegible] seL4

4.1 [illegible]

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 a second new object type, Kernel_Memory, which represents physical memory that can be mapped to a kernel image. This is 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 thread. That thread can then partition the system by first partitioning its Untyped memory by colour. For each partition it clones a new kernel from the initial one, using some of the partition's memory, 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 thread 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 thread 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 Kernel_Image, passing a Kernel_Image with `clone` right and Kernel_Memory capabilities 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. Hence, 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.

The new kernel shares only the following static data with the source kernel:

- the scheduler's array of head pointers to per-priority ready queues, as well as the bitmap used to find the highest-priority thread in constant time
- the current scheduling decision
- the IRQ state table and capabilities for IRQ endpoints (i.e. references to interrupt handlers)
- the interrupt currently being handled (if any)
- the first-level hardware ASID table
- the IO port control table (x86)
- the pointers for the current thread, its capability store (Cspace), the current kernel, idle thread, and the thread currently owning the floating point unit (FPU)
- the kernel lock (for SMP)
- the barrier used for inter processor interrupts (SMP).

We perform an audit of the shared data to ensure it cannot be used as a cross-core side channel.

We add to each TCB the Kernel_Image capability of the kernel that handles that thread's system calls. The creator of a TCB can use the `TCB_Config` system call associate the thread with a specific Kernel_Image .

4.2 [illegible]

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, thus preventing kernels from triggering interrupts across partition boundaries, as long as all interrupts are partitioned. Note that policy-freedom implies that the system will not *enforce* IRQ partitioning.

4.3 [illegible]

The running kernel is mostly unaware of partitioning. 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, 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 a properly partitioned system, the stack switch only happens on a preemption-timer interrupt. 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 recently added *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 the L1 caches. We therefore have to implement a "manual" flush: The kernel sequentially traverses a buffer the size of the L1-D cache, performing a load operation on one word per cache line. Similarly, the kernel 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.

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 is done just prior to the padding of the domain-switch latency. As the kernel image and stack are already switched, and the kernel exit code path is deterministic, this prevents the latency of the exit code from depending on the previous domain's execution (via lower-level caches).

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

[6] This "manual" flush is obviously dependent on assumptions on the (undocumented) line replacement policy implemented by the hardware, 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.

systems have a much simpler, single-level interrupt control mechanism, which avoids this race.

Another race is caused by timer interrupt handling being delayed due to another interrupt occurring just before the preemption timer. We handle this by adding a margin for interrupt handling to the padding time.

In summary, the kernel executes the following steps when handling a preemption tick; steps in bold are only performed on a kernel switch.

(1) acquire the kernel lock
(2) process the timer tick normally
(3) [illegible]
(4) [illegible]
(5) conduct the domain switch by switching the user thread (and thus the kernel image)
(6) release the kernel lock
(7) [illegible]
(8) [illegible]
(9) [illegible]
(10) [illegible]
(11) [illegible]
(12) restore the user stack pointer and return.

4.4 [illegible]

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 bitmap in each kernel that indicates the cores on which the kernel is presently running, the bitmap is updated during each 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 invariant in the face of kernels being dynamically created and destroyed. To always keep the initial kernel, we prevent the destruction of its Kernel_Memory capability by not providing it to userland. That way, even if userland destroys the last Kernel_Image, we guarantee that there is still a kernel and an idle thread. Such a system will have no user-level threads, and will do nothing more than acknowledging timer ticks.

One could think of more sophisticated schemes that allow reusing the initial kernel's memory where the intention is to have a system that is partitioned for its lifetime. For now we accept a small amount of dead memory. On x86, where the kernel image includes 64 KiB of buffers used to flush the L1 caches, this presently amounts to about 216 KiB of waste on a single core or 300 KiB on a 4-core machine. Corresponding Arm sizes are 120 KiB and 168 KiB.

5 [illegible]

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

5.1 [illegible]

For quantitative analysis of timing channels, we use *mutual information* (MI), defined in Shannon information theory [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 receives *outputs* from some set O (the publicly observable 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 output time measurements as a probability density function, meaning that we are calculating 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. Then, we 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. If $\mathcal{M}$ is very small, e.g., less that 1 millibit, we can safely say that any channel is closed or negligible. 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 $\mathcal{M}_0$ (we note that the 95th highest result from the tests would only approximate the 95% confidence interval, not give it exactly).

If the estimate of MI from the original dataset is outside the 95% confidence interval for zero leakage (i.e., $\mathcal{M} > \mathcal{M}_0$) we say that 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 $\mathcal{M}_0$). If the estimated MI is within, or equal to, the 95% confidence interval we conclude that the dataset does not contain evidence of an information leak.

5.2 Hardware platforms

We conduct our experiments on representatives of the x86 and Arm architectures; Table 1 gives the details. 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. (This scenario makes no sense on inter-core channels due to the concurrent access.) 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 prohibiting any 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].

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 equals the size of the L1-D or LLC. Note that for the L1 caches, the indirect cost is somewhat academic: It would be highly unusual for a process to find any hot data in the L1 after another partition has been executing for a full time slice (i.e. many milliseconds).

Table 2 shows results. The surprisingly high L1-flush cost on x86 is a result of our "manual" flush: less than 0.5 μs is for the L1-D flush, the rest is for the L1-I, where each of the chained jumps 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. Flushing the L1 can be expected to add well below 1% overhead, while flushing the whole cache hierarchy will add substantial overheads.

System	Haswell (x86)	Sabre (ARMv7)
Microarchitecture	Haswell	Cortex A9
Processor/SoC	Core i7-4700	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

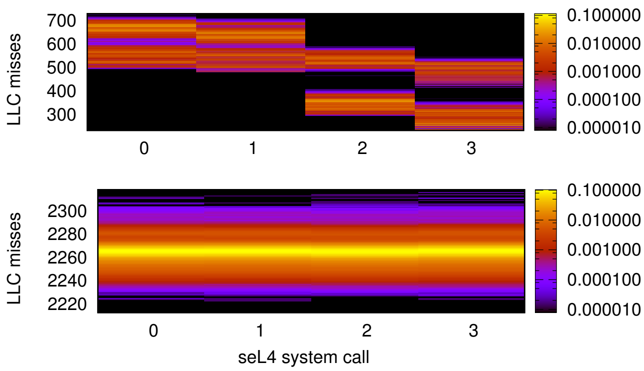
Table 1: Hardware platforms

5.3 Timing channels

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 and inter-core timing channel benchmarks that exploit conflicts on all levels of caches (Section 5.3.2), and a timing channel based on domain switching latency (Section 5.3.4).

	x86			Arm		
Cache	dir	ind	total	dir	ind	total
L1 (μs)	25.52	1.08	26.59	20	24.53	44.53
all (ms)	0.27	0.25	0.52	0.38	0.77	1.15

Table 2: Cost of full flush



Fig[illegible]2: K[illegible]6, [illegible] [illegible] [illegible]mFo [illegible]255,790 [illegible]0.79 [illegible] $\mathcal{M} = 0.79$ [illegible] $n = 255,790$. **[illegible]** $\mathcal{M} = 0.6$ [illegible]m $\mathcal{M}_0 =$ **0.1** [illegible]m $n = 255,040$ (1 [illegible]m 10^{-3}[illegible]

5.3.1 Timing channel via a shared kernel image. We demonstrate that partitioning only user space is insufficient for mitigating covert channels, even though on seL4 this automatically partitions dynamic kernel data (Section 2.4). Among others, this setup already defeats the attack of van Schaik et al. [2018], as page tables are automatically coloured.

We implement an LLC covert channel between coloured user-space processes. The sender sends information 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 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 call is returned.

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 2 (top) shows the resulting *channel matrix*, which represents 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 (cache misses) being correlated with input symbols (system calls), i.e. variations of probability (colour) along horizontal lines. `Signal` and `TCB_SetPriority` lead to 500–700 misses, while `Poll` and idle result in 200–600 misses, a clear channel. Calculating the MI gives an estimated information leak of $\mathcal{M}$=0.79 bit per iteration (2 ms), transmitting 395 b/s. While the channel could be made more efficient with more complicated encoding schemes, this is not the main focus of our work.

With cloned kernels the channels disappear (bottom of Figure 2). The remaining channel is measured as $\mathcal{M} = 0.6$ millibits (mb), therefore closed or 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 a 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 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 a integer from a number of consecutive pages. We use a 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 skiping 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.

[illegible]	C[illegible]	[illegible]	F[illegible]	[illegible]
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)
	L2	1,900	21 (22)	1.4 (1.4)

[illegible]3: [illegible] $\mathcal{M}_0$.

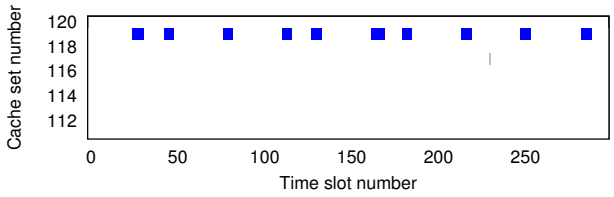


Fig 3: LLC

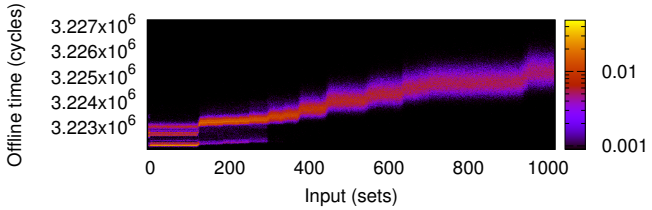


Fig 4:

1.4 b $n = 1828$ $\mathcal{M} =$

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

On Haswell, the picture is the same except for the L1-I and L2 channels. The L1-I channel does not seem quite closed with time protection, although the residual capacity is negligible, and likely results from our imperfect "manual" flush. While the full flush closes the L2 channel, our implementation of time protection (which colours the L2) fails to do this, leaving a sizeable channel of 50 mb. We believe that the remaining channel is due to the aggressive data prefetcher, as the channel is decreased to $\mathcal{M} = 6.4$ mb ($\mathcal{M}_0 = 4.1$ mb) with the data prefetcher disabled. This is strong evidence for the need of a better software-hardware contract for controlling any hidden microarchitecture state [Ge et al. 2018a].

5.3.3 Side channel on the LLC. To test the side channel mitigation for LLC-based attacks, we reproduce 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 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 on Figure 3. On cache set number 119, we see a sequence of blue 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 measures two properties: *online time* is the receiver's uninterrupted execution time (i.e. the observed length of its time slice) while *offline time* as the time between receiver's executions. The receiver repeatedly checks the cycle counter, waiting on a large jump that indicates a preemption. The length of the jump is the offline time, whereas the time between consecutive intervals is the online time.

The sender varies the number of L2 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 4 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 interrupt partitioning with a sender which sends "no" by doing nothing or "yes" by programming a timer to fire every millisecond, the receiver is as in Section 5.3.4. Without interrupt partitioning, if the interrupt fires while the receiver is executing, in average 0.5 ms into its time slice, the kernel will be invoked, resulting in the receiver recording a short on-line time. As the kernel has masked the IRQ, it will not fire again until the sender

x86	On-line	8.4	0.5 (0.5)
pad = 58.8 μs	Off-line	8.3	0.6 (0.6)
Arm	On-line	1,400	16.3 (24.6)
pad = 62.5 μs	Off-line	1,400	210 (237.2)

4: Ch

x86	no	10 (0)	10 (0)
	yes	5 (5)	10 (0)
Arm	no	10 (0)	10 (0)
	yes	5 (5)	10 (0)

5: I

	x86		Arm	
	Cycles	O/H	Cycles	O/H
original	381	-	344	-
colour-ready	386	1%	391	14%
intra-colour	380	0%	395	15%
inter-colour	378	-1%	389	13%

Table 6: IPC [illegible]

acknowledges it, and the receiver will not be interrupted a second time, and thus record a long on-line time, in average 9.5 ms for a 10m ms time slice. This bi-modal distribution is an effective channel and is reflected in the large, 5 ms standard deviation in Table 5.

With interrupt partitioning, the receiver is not preempted during its time slice, resulting in a deterministic on-line time, and thus a closed channel.

5.4 [illegible]

5.4.1 IPC microbenchmarks. We evaluate the impact of time protection by measuring the cost of the most important (and highly optimised) microkernel operation, cross-address-space message-passing IPC. Table 6 summarises the results, 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) examining baseline cost of our mechanisms. Standard deviations from 30 runs are less than 1%.

We find that the time-protection mechanisms add negligible overhead on x86. On Arm, in contrast, there is a significant baseline cost to supporting the kernel clone mechanism, resulting from the fact that with multiple kernels, we can no longer use global mappings with large entries to map the kernel's virtual address space. The Sabre's A9 core has a 2-way L2-TLB, resulting in increased conflict misses on the cross-address-space IPC test. There is no further overhead from using cloning.

[illegible]	[illegible]	Idle	L1-D	L1-I	L2	LLC
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	N/A	1.6
	Full flush	414	414	414	N/A	414
	Protected	27	27	27	N/A	31

Table 7: Cost of [illegible] (μs) [illegible] 5.3.2.

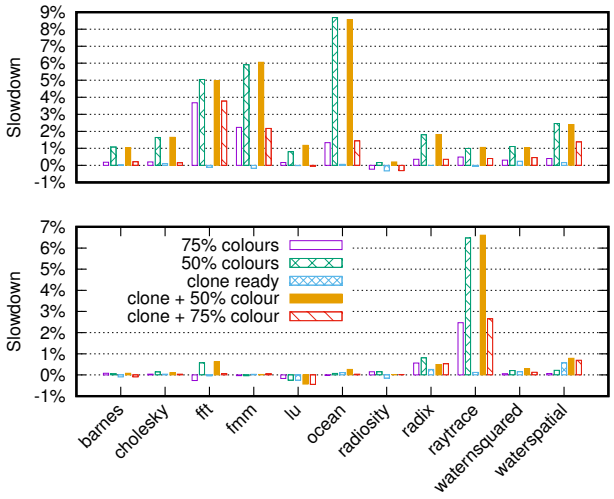


Figure 5: [illegible] x86 (p[illegible] Arm

5.4.2 Domain switching cost. In Table 2 we measured the worst-case cache-flush costs. We expect those to dominate the cost added to domain switches by time protection. To verify we evaluate the domain-switch latency (without padding) for a number of our attack workloads. Specifically we measure 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 7 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. We second 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).

5.4.3 The cost of cache colouring. To evaluate the cost of cache colouring, we port the Splash-2 benchmark [Woo et al. 1995] to the seL4 system (except volrend due to its Linux dependencies), with running parameters that consume 220 MiB

of heap and 1 MiB of stack. Figure 5 shows the overhead of cache colouring with and without the kernel clone mechanism. We report the mean of 10 repeated single-threaded runs (standard deviations are below 3%). The benchmarking thread is the only user thread in the system.

On Sabre, cache colouring introduces less than 1% slowdown for most of the benchmarks. The only exception is *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, but 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% limits the overhead to below 3.5%. As for Arm, the kernel clone mechanism has close to zero overhead.

5.4.4 The impact of domain switches. For evaluating the full impact of time protection, we select from Splash-2 the benchmarks suffering the highest and lowest cache-partitioning overheads according to Figure 5. We simulate a timing-channel defence scenario, with the Splash benchmark sharing a core with an attacking thread. The latter is continuously probing the L1-I and the LLC caches. We use full time protection with a 10 ms time slice. We give the Splash program 50% or 75% of the cache and use the padding times of Table 4. Note these are well above the worst-case L1 flush costs of Table 2 and could be optimised. We report in Table 8 averages of 10 runs (standard deviations below 0.1%).

On Haswell (x86), cache partitioning actually improves performance of the *radiocity* benchmark: it provides performance isolation, removing the frequent conflict misses resulting from the attack thread in the unprotected system. This performance gain offsets the increased context-switch latency from time padding; we see the same effect on *ocean* when it gets a 75% share of the cache. The overhead resulting from padding is about 0.7%.

The performance isolation effect is even more pronounced on the Arm: *raytrace* consistently performs better with a partitioned cache, and the performance of *lu* is practically unaffected by partitioning. Padding only introduces 0.2%–0.4% performance overhead.

	x86		Arm	
50% colour	4.8%	-0.5%	0.03%	-2.4%
50% + padding	5.5%	0.1%	0.3%	-2.0%
75% colour	-0.3%	-0.5%	-0.02%	-6%
75% + padding	0.4%	0.1%	0.2%	-5.8%

6 RLATED

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] visualizes 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.

Page colouring for partitioning caches goes back to Bershad et al. [1994]; Kessler and Hill [1992], who proposed it 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). Ge et al. [2018a] investigate shortcomings in architectural support for preventing timing channels and propose an extended hardware-software contract.

Multikernels [Baumann et al. 2009] consist of multiple, shared-nothing kernel images on the same hardware platform, although on separate cores, for improved many-core scalability. Barrelfish/DS [Zellweger et al. 2014] separates OS kernel images from physical CPU cores, to support hotplugging and energy management.

7 CNLSN

We proposed, implemented and evaluated *time protection*, a mandatory, black-box kernel mechanism for preventing microarchitectural timing channels. Time protection employs a combination of cache partitioning through colouring and flushing of non-partitionable hardware state. It leverages a policy-free *kernel clone* mechanism to almost perfectly partition the kernel itself, resulting in a per-partition kernel image on each core, with interrupts being partitioned as well. Our evaluation shows that the mechanisms are effective for closing all studied timing channels, while imposing

small to negligible performance overhead. While present x86 hardware has some shortcomings that prevent perfect time protection, it would be easy for manufacturers to address this by supporting more targeted flush operations for microarchitectural state.

REFERENCES

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

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

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-Verlag, 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*. 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*. 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.

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*. 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.

Tom Chothia and Apratim Guha. 2011. A Statistical Test for Information Leaks Using Continuous Mutual Information. In *IEEE Computer Security Foundations Symposium*.

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *International Conference on Computer Aided Verification*.

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*. Istambul, TK.

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*.

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.

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*. 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*. Oakland, CA, US, 490–505.

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.

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

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *Asia Conference on Computer and Communication Security (ASIA CCS)*. Xi'an, CN, 353–364.

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, 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*. 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)*. 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.

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*. 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*. 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*. 222–225.

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)*. San Diego, CA, US.

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*. San Francisco, CA, 415–429.

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*.

Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCon 2005*. Ottawa, CA.

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)*. Tokyo, JP.

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* (1948). 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)*. HK, 194–199.

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

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*. 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.

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*. 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*. San Diego, CA, US.

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*. 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*. Bellevue, WA, US.

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*. San Diego, CA, US, 719–732.

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)*. Santa Barbara, CA, US, 346–367.

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*. 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*. Raleigh, NC, US, 305–316.