Can We Prove Time Protection?

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

Gerwin Klein
Data61, CSIRO
and UNSW Sydney
gerwin.klein@data61.csiro.au

Toby Murray
University of Melbourne
and Data61, CSIRO
toby.murray@unimelb.edu.au

Abstract

Timing channels are a significant and growing security threat in computer systems, with no established solution. We have recently argued that the OS must provide *time protection*, in analogy to the established memory protection, to protect applications from information leakage through timing channels. Based on a recently-proposed implementation of time protection in the seL4 microkernel, we investigate how such an implementation could be formally proved to prevent timing channels. We postulate that this should be possible by reasoning about a highly abstracted representation of the shared hardware resources that cause timing channels.

CCS Concepts • **Security and privacy → Trusted computing**; • **Software and its engineering → Operating systems**; **Software verification**.

Keywords timing channels, theorem proving, seL4

ACM Reference Format:
Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can We Prove Time Protection?. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321431

1 Introduction

Timing channels are a major threat to information security; they exist where the timing of a sequence of observable events depends on secret information [Wray 1991]. The observation might be of an externally visible event, such as the response time of a server, and might be exploitable over intercontinental distances [Cock et al. 2014]. Or it might only be locally observable, i.e. by a process or VM co-located on the same physical machine. Even the latter scenario may still enable remote attacks – if the observing process has access to the network and is controlled by a remote agent. The seriousness of the threat was recently highlighted by the Spectre attacks [Kocher et al. 2019], where speculatively executed gadgets leak information via a covert timing channel.

HotOS '19, May 13–15, 2019, Bertinoro, Italy

The secret-dependence of events may have *algorithmic* causes, e.g. crypto implementations with secret-dependent code paths. Or they may arise from interference resulting from competing access to limited hardware resources, such as caches; there exists a wide variety of such *microarchitectural channels* [Ge et al. 2018b].

Whether algorithmic or microarchitectural, those channels represent information flow across protection boundaries, i.e. *the boundaries are leaky*. Ensuring the security of these boundaries should be the job of the operating system (OS); yet, no contemporary, general-purpose OS seems to be capable of it. Clearly, this is not an acceptable situation, and we have recently called for OSes to provide *time protection* [Ge et al. 2018a] as the temporal equivalent of the well-established concept of memory protection.

Memory protection is a solved problem: the formal verification of seL4 proved, among others, that the kernel is able to enforce spatial integrity, availability and confidentiality [Klein et al. 2014]. This categorically rules out information leakage via *storage channels* (provided that the kernel is aware of the state that can be used for such channels). However, the approach taken in the seL4 verification has no concept of time, and therefore cannot make any claims about *timing channels*.

Our aim is to rule out timing-channel leakage just as categorically as information flow via storage. Put differently, *we aim to formally prove correct implementation of time protection*. This paper investigates the feasibility of, and prerequisites for, achieving such a proof. Obviously, we would not bother writing this paper if we were not convinced that it is feasible to achieve our aim, under certain conditions, which come down to hardware satisfying certain requirements. We have recently demonstrated that not all recent processors satisfy these requirements, resulting in a call for a new, security-oriented hardware-software contract [Ge et al. 2018a].

We claim that, for hardware that honours this contract, we will be able to achieve our aim of proving time protection, and thus eliminate microarchitectural timing channels. **The key insight behind this claim is that these channels result from competing accesses to shared hardware resources, and thus proving the absence of such competition precludes the timing channel.**

Note that other physical channels, such as power draw, temperature, or acoustic or electromagnetic emanation, are outside the scope of this work.

2 Threat Scenario

The basic problem we are concerned with is a secret held by one security domain, Hɪ, being leaked to another domain, Lo, which is not supposed to know it. The leaking might be intentional, by a bad actor (Trojan) inside Hɪ, constituting a *covert channel*. Or it can be unintentional, via a *side channel*. Note that Hɪ, Lo are relative to a particular secret, we do not assume a hierarchical security policy such as Bell and LaPadula [1976], and there may be other secrets for which the roles of the domains are reversed. It is the duty of the OS to prevent any unauthorised information flow, no matter what the system's specific security policy might be.

Our notion of a security domain refers to a subset of the system which is treated as an opaque unit by the system's security policy (i.e. intra-domain information flow is not restricted by the policy). In OS terms, a domain consists of one or more (cooperating) processes.

We assume that the OS provides strong, verified memory protection, and is free of storage channels, seL4 being an example. Our primary concern is microarchitectural channels, i.e. channels that exploit competition for finite hardware resources that are abstracted away by the instruction-set architecture (ISA), the classic hardware-software contract. This means that algorithmic channels are not our primary concern, but we will discuss in Sect. 4.3 how time protection can be employed to remove such channels (within limits).

Like memory protection, time protection is a black-box OS mechanism, that provides *mandatory security enforcement* without relying on application cooperation.

For realism, i.e. to ensure that contemporary hardware is at least close to satisfying the requirements of time protection (and can fully satisfy them with minor enhancements) we limit our scope in one important way: we do not (yet) attempt to prevent *covert channels* through stateless interconnects. Such channels, exploiting the finite bandwidth of interconnects through concurrent competing access, are trivial to implement [Hu 1991; Wu et al. 2012]: a Trojan running on one core signals by modulating its use of interconnect bandwidth, and a spy running on a different core measures the remaining bandwidth by trying to saturate the shared interconnect. Such channels can only be prevented with hardware support that is not available on any contemporary mainstream hardware.[1] We will be able to extend time protection in a fairly straightforward way, should such hardware support (or at least an accepted model for it) become available.

An obvious example of the excluded scenario would be a *covert channel* between two virtual machines (VMs) concurrently executing on different cores of the same processor on a public cloud. Such a covert channel is not a particular concern: the Trojan in the victim VM does not need the co-located spy, as it can communicate by other means, e.g. by modulating its network communication. Shared-cache *side channels* are a real concern in the cloud scenario [Irazoqui et al. 2015; Liu et al. 2015] and our threat scenario includes this case (i.e. requires their prevention). But stateless interconnects reveal no address information, and no interconnect side channels have been demonstrated to date [Ge et al. 2018b]; they are most likely impossible.

There are covert-channel scenarios we would like to address but are unable to do so with present hardware. For example, server-provided JavaScript code running in a browser may contain attack code that uses a covert channel to communicate with a (supposedly confined) Trojan inside one of the same user's untrusted apps. If attacker and Trojan are executing concurrently on different cores, the interconnect channel can be used by the Trojan to leak to the attacker, which then relays the secrets back to the server. While such an attack would not be easy to execute, it is clearly possible and should be prevented. However, this is impossible on contemporary commodity hardware.

In summary, our threat scenario assumes that on multicore processors, security domains are co-scheduled across all cores (allowing full protection against *covert channels*) or, if multiple domains are executing concurrently, only *side-channel* attacks are prevented. Where multiple domains time-share a core, full isolation is provided (including covert channels).

For now we exclude timing channels resulting from hardware beyond the main processor, such as DRAM row buffers, peripheral devices, disks. We note that it should be possible to fit those into the same model, e.g. DRAM buffers are a form of physically-addressed cache, but we leave their detailed treatment to future work.

3 Timing-Channel Mechanisms

There are two ways in which Lo may learn Hɪ's secret: by timing observable actions of Hɪ, or by Lo observing how its own execution speed is influenced by Hɪ's execution.

3.1 Timing Lo progress

This channel utilises the performance impact of interference between processes resulting from competition for shared hardware resources, especially stateful resources such as caches, TLBs, branch predictors and pre-fetcher state machines. For example, Lo's rate of progress (performance) is affected by cache misses. If Lo shares a cache with Hɪ (either time-sharing a core-private cache or concurrently sharing a cache with Hɪ's core), then the miss rate will depend on Hɪ's cache usage. If the cache is set-associative (which they inevitably are nowadays), then the pattern of cache misses will also reveal address information from Hɪ. Such address

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

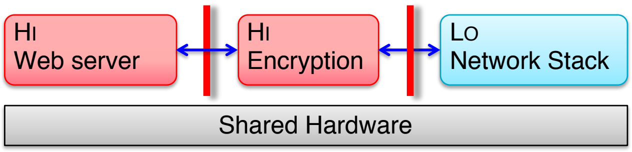

Figure 1. Encryption engine as a downgrader.

information supports the implementation of side channels with potentially high bandwidth, e.g. where the secret is used to index a table [Ge et al. 2018b].

An effective exploitation of such a channel is the *prime-and-probe* technique [Osvik et al. 2006; Percival 2005]. Here Lo fills the cache by traversing a buffer large enough to cover the cache (prime phase). After, or while, Hi is executing, Lo traverses the buffer again, monitoring the time taken for each access (probe phase); a long latency indicates a conflict miss with Hi's cache footprint. The address of the missing access reveals the index bits of Hi's access.

Prime-and-probe can be used as a high-bandwidth covert channel, where Hi explicitly encodes information into the memory addresses accessed, or as a side channel, where the encoding is implicit in Hi's normal execution (e.g. via a secret-derived array index). It can be used for time-shared (core-private) caches as well as caches shared between cores.

3.2 Timing Hi events

This attack is based on observing the exact timing of Lo's interactions. It may be observing external interactions, such as timing response latencies of a server, or internal events, such as the timing of messages Lo exchanges with other components.

On a first glance this might seem like a silly case, why worry about a covert channel if there is an overt one, such as message passing? In fact, this is a common scenario: Hi might be a *downgrader*, an entity trusted to handle secrets and decide which can be safely declassified. A common example is a crypto component, which encrypts secrets, e.g. from a web server, and publishes the encrypted text, by handing them to a network unit; this is shown in Figure 1.

In this case, the leakage might be resulting from an algorithmic channel (e.g. a crypto implementation with secret-dependent execution), a Trojan modulating the speed of the encryption (possibly via microarchitectural interference), or the server itself leaking through the timing of messages to the crypto component.

Time protection here must make execution time deterministic, meaning that message passing or context switching happen at pre-determined times. Obviously, the OS can only provide the mechanism here (deterministic switch/delivery time), not the policy (the time of the switch). This must be set by the system designer or security officer, taking into account issues like the worst-case execution time (WCET) of the encryption.

Cock et al. [2014] proposed a possible model: a synchronous IPC channel switches to the receiver only once the sender domain has executed for a pre-determined minimum amount of time. It is then left to the system designer to determine a safe time threshold.

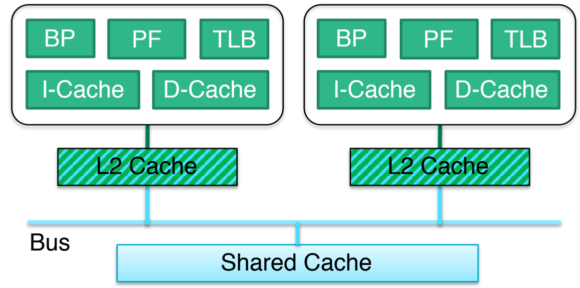

Figure 2. Partitioning hardware resources. Resources shown (dark) green must be temporally partitioned, those in (light) blue spatially, hatched ones may be either.

4 Closing the Leaks: Time Protection

Our principled defence against unauthorised information flow through timing channels is *time protection*, just as memory protection stops unauthorised access to storage. We define time protection as *a collection of OS mechanisms which jointly prevent interference between security domains that would make execution speed in one domain dependent on the activities of another* [Ge et al. 2019].

4.1 Spatially and temporally partitioning hardware

As microarchitectural timing channels result from competition for (non-architected) hardware resources, eliminating them requires removing the competition. This means that on order to provide time protection, *the OS must partition those resources between security domains, either spatially or temporally (by time sharing).* With temporal partitioning, any history dependence must be erased when re-assigning the hardware to a different domain, by resetting any state to a defined value (eg. flushing).

Figure 2 shows how resources are partitioned. Temporal partitioning is obviously incompatible with concurrent access, meaning it can only work for resources that are private to an execution stream. In the absence of hyperthreading, this applies to core-local resources, such as the L1 caches, private L2 caches (on Intel hardware), TLBs, branch predictors, and core-local prefetchers.

Resources that are accessed concurrently, especially caches shared between cores, must therefore be spatially partitioned. This would also be the only option for core-local state when hyperthreading is enabled. However, no mainstream hardware supports partitioning of hardware resources between hyperthreads, and such partitioning would seem fundamentally at odds with the concept of hyperthreading, which is based on improving hardware utilisation by sharing. Consequently, there are a plethora of side-channel attacks between hyperthreads [Ge et al. 2018b]. We have to conclude that *hyperthreading is fundamentally insecure, and multiple hardware threads must never be allocated to different security domains* (multi-threading a single domain is not a security issue). This is well understood [Zhang et al. 2012], and hypervisor vendors advise against enabling hyperthreading [Marshall et al. 2010].

Spatial partitioning of shared (physically-addressed) caches is possible without extra hardware support by using page colouring [Kessler and Hill 1992; Liedtke et al. 1997; Lynch et al. 1992]. Colouring makes use of the fact that the associative lookup of a large cache forces a page into a specific subset, so only pages mapping to the same subset, said to have the same colour, can compete for cache space. By ensuring that different security domains are allocated physical page frames of disjoint colours, the OS can partition the cache between domains. Modern last-level caches have at least 64 colours.

Note that TLBs are virtually-indexed caches that cannot in general be spatially partitioned (ASIDs notwithstanding) and must be flushed on a partition switch. This is orthogonal to the need for shooting down remote TLBs when removing mappings. TLB shoot-down is a requirement for functional correctness irrespective of timing channels, and as such assumed to be verified already.

To summarise, **microarchitectural timing channels can be prevented if the OS can partition all shared hardware either spatially or temporally, with temporal partitioning implying the ability to reset any relevant hardware state.** Furthermore, spatial partitioning is the only option for hardware that is concurrently accessed. Together with a few other conditions outlined by Ge et al. [2018a], these form part of a security-oriented hardware-software contract, which we call the *augmented ISA* (aISA). The aISA allows the OS to prevent timing channels, while the ISA alone is an insufficient contract for ensuring security [Heiser 2018; Hill 2018].

4.2 Implementing time protection

We have recently proposed an implementation of time protection in seL4, for hardware that conforms to a security-oriented aISA [Ge et al. 2019]. It uses cache colouring to spatially partition shared caches. As even read-only sharing of code is sufficient for creating a channel [Gullasch et al. 2011; Yarom and Falkner 2014], we also colour the kernel image. This is achieved by a policy-free *kernel clone* mechanism, which allows setting up a domain-private kernel image in coloured memory, with only a small amount of static data shared between images.

We flush temporally partitioned microarchitectural state on a security-domain switch (but not on intra-domain context switches). For writable microarchitectural state (e.g. the L1 data cache), the latency of the flush is itself dependent on execution history (number of dirty lines), which would create a channel. We avoid this channel by padding the domain-switch latency to a fixed value.

Padding is performed at the very end of the domain-switch operation, just before the kernel returns to user mode. To ensure deterministic timing of the few instructions required to return, we force all required shared kernel data into the L1 cache prior to padding. This guarantees a defined hardware state prior to performing the exit from kernel mode, and thus a fixed latency for those instructions. Note that pre-fetching instructions is not needed, as the kernel text is coloured.

The overall effect is that the user-visible time taken by the kernel to switch partitions, from when the timer interrupt is raised until return to user mode in the destination domain, is constant and thus cannot leak information.

The padding time must obviously be at least the worst-case latency of the flush, but also needs to account for any delay of the handling of the preemption-timer interrupt by other kernel entries. The domain switch could be delayed if user code invokes a system call, or an interrupt happens, just before the domain-switch timer interrupt. Such a delay would also constitute a channel. The padding time must be long enough to allow the system to handle the kernel entry before performing the domain switch. To avoid very long domain-switch latencies, this should be done by determining whether there is sufficient time left in the present time slice, and if not, configure the kernel state to perform the operation during the present domain's next time slice, and then progress directly to the domain switch. This is similar to the way seL4 restarts preempted operation at preemption points [Klein et al. 2014].

For generality (see Sect. 4.3) we make determination of the padding time not the job of the OS, but an attribute of the switched-from security domain, controlled by the system designer: the next domain will not start executing earlier than the previous domain's time slice plus the padding time.

Use of the correct padding time requires some reasoning outside the formal model, as it needs to use the output of the WCET analysis and the experimentally determined worst-case flush time. In terms of assurance, this is similar to other cases where the formal world must be connected to the physical world.

Finally, interrupts could also be used as a channel, if Hi triggers an I/O such that its completion interrupt fires during Lo's execution. We prevent this by partitioning interrupts (other than the preemption timer) between domains, and keep all interrupts masked that are not associated with the presently executing domain [Ge et al. 2019].

4.3 Preventing algorithmic channels

Padding is a general mechanism that can also be used to prevent algorithmic channels. In the scenario of Figure 1, we can pad the execution time of the downgrader's domain to a safe value (an upper bound of its execution time). This means that the kernel will not switch away from the domain before the padding time has passed. If the downgrader's domain has no other runnable threads, the system would have to idle until the switch time.

5 Proving Time Protection

At first glance, one might expect that proving time protection is a hopeless exercise. After all, the precise interaction between microarchitectural state and execution latency is unspecified for modern hardware platforms, and the latency of some instructions may vary by orders of magnitude depending on hardware state. Formally reasoning about precise execution latencies is therefore infeasible [Klein et al. 2011].

However, we argue that reasoning about the exact latency of executions is unnecessary. **The key insight is that these channels are effected by shared hardware resources, and if we can prove that no sharing happens, there can be no timing channels.** Consequently, proving temporal isolation requires formal models of microarchitectural state, but these can be kept abstract. They only require enough detail to identify resources that need to be spatially partitioned (and how such partitioning is performed), and state that must be reset at switch time (and how to reset it). That is, we do not need to know how long an instruction will take to execute, only which microarchitectural state its execution time depends on and how this state behaves wrt. partitioning and flushing.

5.1 Reasoning about hardware state

For spatially partitionable state, temporal isolation becomes a *functional* property (namely an invariant about correct partitioning) that can be verified without any reference to time, existing verification techniques therefore apply.

For state that requires flushing, correct application of the flush is also a functional property. As mentioned in Sect. 4.2, the latency of flushing operations themselves needs to be hidden by the OS, by padding its execution latency. As the padding time is an explicit property of the previously executing domain, correct padding can be verified with a relatively simple formalisation of hardware clocks, which allows verifying padding time by simply comparing time stamps, reducing this to a functional property as well.

Once timing-channel reasoning is reduced to the verification of functional properties, it should be possible to integrate it into existing proof frameworks of storage-channel freedom, such as seL4's information flow proofs [Murray et al. 2013].

Indeed, under this approach **timing-channel reasoning is transmuted into reasoning about storage channels**, reducing it to a solved problem, and also enabling reasoning about timing-channels without reference to precise execution time. This possibility may seem surprising, but it is known that the distinction between storage and timing channels is not fundamental, but refers to the mechanisms used for exploitation [Wray 1991]. In our case we transform the temporal interference problem into a spatial one, by reasoning about the shared hardware resources which the channels exploit.

5.2 Hardware formalisation

Carrying out these proofs requires a model of the shared hardware resources (the *microarchitectural model*) that influence execution latencies, as well as a simple model of a hardware clock (the *time model*) to allow reasoning about elapsed time intervals. Naturally these models are interrelated: how much an execution step advances the hardware clock naturally depends on the microarchitectural state that influences execution time.

Crucially, a precise description of this interaction is not necessary. Instead, the interaction can be faithfully yet feasibly modelled as follows. Firstly, the microarchitectural model must delineate the spatially partitionable state from the flushable state, and all microarchitectural state must be spatially partitionable or flushable (Sect. 4.1). Secondly, the time model, which captures how far time advances on each execution step, is defined as a *deterministic yet unspecified* function of the microarchitectural state. Then, when the microarchitectural state is properly partitioned and flushed, one can prove that a security domain's execution time cannot be influenced by other domains (see Sect. 5.3 below).

This construction neatly reflects the basic assumptions that (i) the hardware provides sufficient mechanisms to spatially partition or flush microarchitectural state between security domains, that (ii) such mechanisms work correctly, and that (iii) these account for all microarchitectural state that influences execution time.

5.3 Information-flow proofs

With these models in hand, time protection can then be proved by showing that there is no way in which the execution of one domain can affect the execution timing of another domain.

Specifically the proofs must show that all resource partitioning and flushing is applied at all times and not bypassable, and that domain switches (including flushing) are correctly padded to a constant amount of time (under the assumption that the specified padding value, obtained by a separate analysis, is sufficiently large). These proofs can then be integrated with existing proofs of storage-channel freedom to derive the absence of timing channels as follows.

Without loss of generality, fix some domain (Lo) and consider one of its execution steps for which we want to show its timing cannot be influenced by another domain (Hi). There are two possibilities: (Case 1) either it is an ordinary user-mode instruction, or (Case 2) it is a trap (a system call, exception, or interrupt arrival). For Case 1, the execution time given by the time model will be affected by the shared hardware resources in the microarchitectural model. Recall that this effect can be modelled by an unspecified deterministic function from the state of the microarchitectural model to an elapsed (symbolic) time value. For an individual instruction this function will examine the state of the instruction cache, namely the cache set identified by the program counter, and the state of the data cache for any memory address accessed by that instruction. Since the access does not fault (otherwise it would be a trap), all such memory accesses must lie within the physical memory of the current domain and thus within areas of the cache that cannot be affected by other partitions (due to correct cache partitioning by the kernel, or correct flushing, e.g. for the on-core L1 cache).

In the above, "instruction cache" refers to any microarchitectural state that affects the latency of fetching and executing an instruction, including pipeline state, branch predictor, pre-fetcher and TLB. The hardware model does not even have to distinguish these resources, they can be lumped into the same abstraction, with the flush operation potentially consisting of multiple steps. Similarly, for load or store instructions, "data cache" includes any state that affects the time to load or store a register. The upshot is that the resulting execution time cannot be affected by other partitions.

For Case 2, we distinguish two sub-cases: The trap is either (Case 2a) a system call, an exception or a partitioned interrupt, or (Case 2b) it is the arrival of a timer interrupt signalling a switch to the next domain. For Case 2a, the execution time depends on the state of the instruction cache (in the above generalised meaning) wrt. the kernel instructions executed, plus the data cache for any data accessed. However, in a partitioned system with the kernel correctly cloned as in Sect. 4.2, the former cannot be affected by other partitions and the latter accesses only data of the current domain. The only remaining state that might be accessed is global kernel data, for which we will prove that it is accessed deterministically, and whose cache state after a domain switch is independent of prior Hi activity (due to correct flushing). A similar, if naturally more involved, argument applies as to the user mode case (Case 1). For Case 2b, we invoke the proof of the constant-time domain switch property. □

Expressing elapsed time as a value in the state of the time model (updated by an unspecified function of the microarchitectural model) achieves the above discussed reduction of timing-channel reasoning to storage-channel reasoning. As a result, time protection itself can be phrased and proved akin to storage-channel freedom via a suitable noninterference property [Murray et al. 2012].

5.4 TLB

The TLB is an example where the principles of partitioning and flushing can already be observed in a formal model for pure functional correctness: while not yet suitable for reasoning about timing, Syeda and Klein [2018] provided a logic for functional correctness under an ARM-style TLB. For instance, it is easy to show in this model that page table modifications under one address space identifier (ASID) do not affect TLB consistency for any other ASID. This is the kind of partitioning theorem we will use for timing-relevant spatially-partitionable state (i.e. shared caches).

For time protection, the TLB must be temporally partitioned (flushed), so this model is not directly applicable, but it points the way towards a model for spatially-partitionable caches. The model is a high-level abstraction of the TLB proved sound with respect to a low-level model that would be infeasible to reason about directly. We propose the same for timing behaviour. Instead of reasoning about a detailed low-level architecture model with precise timing information, we only record the information needed for timing-independence.

6 Conclusions

We conclude that proving time protection should be possible with established formal methods, thanks to the key insight that they result from spatial-type microarchitectural resources, and can thus be treated as storage channels. This requires some reasoning about those hardware resources, but we expect to get away with very high-level abstractions.

The key challenge is to achieve agreement on a hardware-software contract [Ge et al. 2018a] that makes it at least *possible* to remove timing channels. Verifying time protection implies formalising (at least aspects of) this contract, which will also enable verifying hardware implementations against the contract.

In terms of getting agreement on such a contract, and honouring it into the future, we are clearly at the mercy of processor manufacturers. The RISC-V community is presently taking the lead in incorporating the security contract into the processor/platform specification.

7 Acknowledgments

The authors thank the Australian Research Council for enabling this work through grant DP190103743.

References

D.E. Bell and L.J. LaPadula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997. MITRE Corp.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Some Timing Channels on seL4. In *ACM*

Conference on Computer and Communications Security. ACM, Scottsdale, AZ, USA, 570–581.

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: the Missing OS Abstraction. In *EuroSys Conference*. ACM, Dresden, Germany, 17.

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.

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.

Gernot Heiser. 2018. For Safety's Sake: We Need a New Hardware-Software Contract! *IEEE Design and Test* 35 (March 2018), 27–30.

Mark D. Hill. 2018. A Primer on the Meltdown & Spectre Hardware Security Design Flaws and their Important Implications. https://www.sigarch.org/a-primer-on-the-meltdown-spectre-hardware-security-design-flaws-and-their-important-implications/

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.

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.

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.

Gerwin Klein, Toby Murray, Peter Gammie, Thomas Sewell, and Simon Winwood. 2011. Provable Security: How feasible is it?. In *Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX, Napa, USA, 5.

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.

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, 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.

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

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.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. 2012. Noninterference for Operating System Kernels. In *International Conference on Certified Programs and Proofs*. Springer, Kyoto, Japan, 126–142.

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

Hira Taqdees Syeda and Gerwin Klein. 2018. Program Verification in the Presence of Cached Address Translation. In *Interactive Theorem Proving, ITP*, Vol. 10895. Springer's LNCS series, Oxford, UK, 542–559.

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 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.

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.