# Towards Provable Timing-Channel Prevention

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

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

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

## ABSTRACT

We describe our ongoing research that aims to eliminate microarchitectural timing channels through *time protection*, which eliminates the root cause of these channels, competition for capacity-limited hardware resources. A proof-of-concept implementation of time protection demonstrated the approach can be effective and low overhead, but also that present hardware fails to support the approach in some aspects and that we need an improved hardware-software contract to achieve real security. We have demonstrated that these mechanisms are not hard to provide, and are working on their inclusion in the RISC-V ISA. Assuming compliant hardware, we outline how we think we can then formally prove that timing channels are eliminated.

## 1 INTRODUCTION

A covert channel is an information flow that uses a mechanism not intended for information transfer [Lampson 1973]. Traditionally, channels are classified as *storage channels* or *timing channels*, depending on whether time is used to exploit them [Wray 1991].

Storage channels can be considered a solved problem: Murray et al. [2013] proved that the seL4 microkernel [Klein et al. 2009] prevents covert storage channels. In contrast, timing channels are still an open problem, and the recent Spectre attacks [Kocher et al. 2019] clearly demonstrated the threat they pose to security. Spectre also demonstrated that the threat goes beyond *side channels*, which exploit accidental information leakage by innocent code (and are the main focus of the security community): The Spectre authors use *gadgets* in innocent code to construct a Trojan that deliberately leaks information.

The Spectre experience demonstrates the importance of solving the *confinement problem* [Lampson 1973]: It must be possible to encapsulate code that processes secrets so that this code cannot leak the secrets, including through timing channes. In other words, the operating system (OS) must provide mandatory confinement, and as a community we have been negligent by putting this problem into the "too hard" or the "not interesting" baskets.

Of particular importance are *microarchitectural channels*, which exploit implementation details of the hardware that are abstracted away by the hardware-software contract, i.e. the *instruction-set architecture* (ISA). Where such a resource has a capacity limit, competing accesses from multiple execution threads will result in a slowdown of execution, which can be exploited for signalling. The limitation can be bandwidth, such as for interconnects, where contention leads to an observable reduction of the apparent bandwidth; these channels only appear where true concurrency exists, i.e. across cores. The alternative are stateful resources, whose state that depends on execution history, which can be exploited by truely concurrent access or when time-multiplexing the hardware; these are generally caches and other accelerators such as branch predictors and prefetchers. Ge et al. [2018b] present a survey.

Our primary focus are such microarchitectural timing channels, as they are the hardest problem. The determinism we need to achieve to prevent these channels will also allow us to prevent algorithmic channels, where the length of the execution path of a computation depends on secret data.

We recently introduced *time protection* as a mechanism for the systematic prevention of timing channels, and evaluated a proof-of-concept implementation in an experimental version of seL4 [Ge et al. 2019]. Analogous to the well-established concept of memory protection, which prevents processes interfering with each other's memory state, time protection prevents interference that affects execution time, and thus prevents leakage through timing channels. Like memory protection, time protection is a mandatory OS mechanism. The combination of both mechanisms supports confinement of processes.

Our ultimate goal is a formal proof that time protection prevents timing channels, and a demonstration that it can solve real-world confinement problems. The path there presents a number of challenges, and this article will explain them and discuss our approach to overcoming them. Specifically:

- We find that contemporary hardware contains exploitable microarchitectural state that the OS cannot control, making time protection impossible to implement fully (Section 3).
- While we have demonstrated a proof-of-concept implementation of time protection and demonstrated that it is effective to the degree allowed by hardware, it is not yet integrated into a usable system model (Section 4).

- There are not yet suitable formal models of the relevant hardware features; we need to develop such models, prove relevant properties about them, and integrate them into the seL4 proof framework to then prove that confidentiality can be extended to timing (Section 5).

But first we will explain the concept of time protection and its implementation in more detail.

## 2 TIME PROTECTION

Time protection is defined 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.* It extends isolation to the time domain.

Note that the reference to "security domains" is intentional: Memory protection, as we know it, usually applies to process boundaries. Applying time protection to security domains instead gives the OS flexibility in applying the (possibly heavy-handed) time protection only where it is really needed. Security domains are defined by a systems's security policy, and in general consist of one or more processes.

As microarchitectural timing channels result from competing access to shared hardware resources, the only safe way to prevent them is to partition these resources between security domains. This partitioning can be spatial or temporal (a.k.a. time sharing).

Resources that are accessed concurrently must obviously be spatially partitioned; this applies to shared caches and interconnects. For caches, page colouring by physical address is a well-known technique [Kessler and Hill 1992; Liedtke et al. 1997] that is effective and imposes low overhead [Ge et al. 2019].

But spatial partitioning is not available for most on-core resources, such as the L1 caches, TLBs, branch predictors and instruction and data prefetchers. These generally lack the hardware support needed to partition them; techniques such as page colouring require OS control over the addresses used to access the hardware, and for on-core resources that is the virtual address that is outside the control of the OS. Furthermore, L1 caches are small, and partitioning them would have a high performance impact.

Hence a combination is needed: temporal partitioning for on-core resources and spatial partitioning off-core. Ge et al. [2019] explore temporal partitioning in depth, and show that it must go beyond just partitioning user state. The OS must be partitioned, as must interrupts. They achieve this by furnishing each security partition with its own OS image, consisting of code, (most) data and stack. A small amount of shared state is needed to coordinate the images; care is taken that accesses to this state is fully deterministic in terms of execution latency. The core mechanisms are policy free, and based on a kernel *clone* operation that allows setting up partitions dynamically, based on the system's security policy.

The approach is shown to be effective (subject to hardware limitations discussed in Section 3) and efficient, adding of the order of 30 $\mu$s to the latency of a partition switch that happens at time-slice granularity (i.e. milliseconds).

## 3 HARDWARE LIMITATIONS

Complete time protection is only possible where the hardware provides sufficient support. Specifically, *all shared hardware resources must be spatially or temporally partitionable.*

Unfortunately, present hardware does not adhere to this requirement. There are two main issues: partitioning bandwidth and scrubbing.

### 3.1 Interconnect Bandwidth

When two security domains share a bandwidth-limited interconnect, such as a memory bus, a Trojan in one domain can trivially send information to a spy in the other by modulating its use of the available bandwidth. The spy can then monitor the apparent available bandwidth to receive the signal.

To prevent this channel, the bandwidth must be partitioned (a form of spatial partitioning where the "space" is the available bandwidth). Contemporary hardware does not support this. Intel recently introduced *memory bandwidth allocation* (MBA) technology, which imposes limits on the memory bandwidth available to a core [Intel Corporation 2016]. However, this limit is only enforced approximately and can therefore not be used for preventing timing channels.

Software-only bandwidth partitioning is possible in principle by having the OS impose a strict time-division multiple access (TDMA) discipline on the bus by enabling and disabling memory mappings for each TDMA slot. The MemGuard work demonstrated this approach for limiting interference though bus cross-talk in real-time systems [Yun et al. 2013]. However, making this precise enough to prevent covert channels would impose a very large overhead, offsetting most of the benefits from a multicore system, and we do not consider this a feasible approach for our purpose.

However, MBA and MenGuard create hope that low-overhead hardware support for interconnect partitioning may be feasible, and we plan to investigate this issue in the future.

For now, we have little choice but accept (temporary) defeat, and instead limit our ambitions to threat models that do not depend on bandwidth partitioning. Specifically we have to make one of three assumptions:

(1) the system operates single-core while sensitive code is executing
(2) all cores belong to the same security domain

(3) intentional leakage across cores is not an issue.

The first assumption is frequently made in military systems, but is clearly too restrictive for most systems. The second assumption can be satisfied by co-scheduling security domains across all cores, and time-slicing the whole multicore processor between multiple domains. This can work in some cases, but is not too realistic either.

The third maps to real-world scenarios. One of them is a cloud service of mutually-distrusting virtual machines (VMs), each restricted to using a single core at any one time. If such a VM contained a Trojan that wanted to leak information, it would not have to rely on an inter-core covert channel, but could leak directly to the outside work, for example by modulating network traffic.

While covering at least one useful scenario, these restrictions are unpalatable in general, and we really need to hope for better hardware support.

### 3.2 On-core microarchitectural state

As explained above, time-partitioned hardware, which includes most (if not all) on-core microarchitectural state, must be reset to a history-independent state before handing it from one security domain to the next. This seems straightforward, all it takes is a flush, which should be cheap, as this state is mostly read-only state (the L1 D-cache being the one exception).

Ge et al. [2018a] show that the reality is less rosy. For one, the Intel architecture lacks even a dedicated flush of the L1 caches, all it provides is a flush of the complete cache hierarchy. For the purpose of time protection, this is overkill, as the physically-addressed lower-level caches can be spatially partitioned, and for the shared last-level cache (LLC), the flush adds nothing to security. Furthermore, the full flush is prohibitively expensive, of the order of milliseconds. While Intel has recently added an L1-D flush through a microcode update [Intel 2018a], there is still no I-cache flush, and the D-cache flush is not available for all processors.

Arm processors are better in this respect, they provide instructions for flushing the L1 caches. But on all processors we investigated, there is other state that cannot be flushed completely. A major culprit are branch predictors, which cache recent branch targets and whether conditional branches have been taken; these provide high-bandwidth channels. Intel has recently mitigated this as part of their Spectre defences with a feature called indirect branch control (IBC) [Intel 2018b]. However, this proves insufficient for removing branch-predictor channels completely. We also find that the instruction- and data-prefetchers provide channels that cannot be closed [Ge et al. 2019, 2018a].

This is a depressing situation: Full time protection is unimplementable on present hardware, irrespective of processor architecture. However, there is an upside. Our experiments with an open-source RISC-V processor show that implementing a complete flush of on-core state is straightforward, cheap in terms of circuitry as well as latency [Wistoff et al. 2020].

### 3.3 Improved hardware-software contract

The fundamental problem here is that the ISA as the hardware-software contract is designed for enabling functional correctness, but intentionally hides anything to do with timing. It is simply insufficient for ensuring security.

The inescapable conclusion is that the hardware-software contract must be augmented [Ge et al. 2018a]. In particularly, the contract must ensure that hardware meets our requirement for time protection: all microarchitectural resources must be (spatially or temporally) partitionable by software. In the case of on-core state, our experience shows that this is straightforward and mostly a matter of finding the right abstraction. We are engaged with the RISC-V community to ensure that appropriate mechanisms are included in the RISC-V ISA spec.

Bandwidth-limited resources are a more difficult problem that requires more research and reluctantly we leave them out of scope for now.

## 4 TIME PROTECTION SYSTEM MODEL

Our present implementation of time protection in an experimental version of seL4 [Ge et al. 2019] is a proof of concept that implements the basic mechanisms but leaves many issues unresolved. In particular, it only supports single-threaded security domains, which severely restricts its practical use. Clearly there needs to be a notion of hierarchical scheduling, where security domains are scheduled with fixed time slices, but internally consist of multiple multi-threaded proceses that are scheduled within the domain time slice.

The obvious approach here is to use strict time partitioning for domains, as in seL4's existing domain scheduler [Murray et al. 2013], using the standard seL4 scheduler inside domains. This approach, which is widely used in avionics [ARINC 2012], is highly restrictive. Among others it implies that the latency of interrupt handling is of the order of the domain period, typically tens to hundreds of milliseconds – far too long for most real-world systems. Strict partitioning also implies poor resource utilisation, as idle cycles in one domain cannot be used elsewhere.

A better approach would be to donate idle cycles to slack domains that can run in any statically scheduled domain's time slice, but are trusted (ideally through formal verification) not to leak. An interesting challenge is to integrate time protection with the periodic budget enforcement model which we recently added to seL4 for ensuring the temporal integrity of mixed-criticality real-time systems [Lyons et al.

2018]. This could form the basis of a more flexible notion of domain time slices.

A proper system model that retains the power and flexibility of seL4 will also support security boundaries of configurable strength. Eliminating all possible cross-partition information flow, while needed in some cases, is overkill in others. The system's security policy must determine for which boundaries covert channels matter, and whether they matter only in one direction or all, and the kernel must allow providing the degree of security (and associated overheads) that is needed but not more.

In the end we want to be able to provide the right flavour of time protection for everyday use cases. A good one to aim for is browser security: Untrusted code running in a tab should not be able to communicate with other tabs, or use timing side-channels to infer host data. Achieving this will require a flexible isolation model.

# 5 FORMALISING HARDWARE AND PROVING ISOLATION

Proving time protection means proving that seL4's kernel mechanisms, when deployed correctly, properly isolate the timing of one security domain from another. Naturally, two domains that share a storage channel also share a timing channel, so timing isolation makes sense to prove (at least at first) for domains that share no storage channels, as defined by seL4's existing information flow theorem. Once proved, time protection will extend that statement of isolation to also cover timing channels.

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].

## 5.1 How to Prove Time Protection

So how can we prove time protection if we cannot hope to build formal models of the precise timing behaviour of modern hardware? We argue that reasoning about the exact latency of executions is unnecessary. The key insight [Heiser et al. 2019] is to make use of the fact that microarchitectural timing channels result from sharing of capacity-limited hardware resources. Therefore, if we can prove that these resources are not shared (spatial partitioning), or if they are time-shared (which only applies to stateful resources) any history dependence in their state is removed, these channels are eliminated.

This means that we need formal models that represent microarchitectural state in our proofs. However, these models can be kept quite abstract, providing only 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 that its execution time depends on state that must be partitioned or flushed for achieving isolation.

The model does not have to distinguish between different stateful resources that need flushing, such as L1 caches, TLBs, or branch predictors. They can be represented as abstract "blobs", as long as there is a defined sequence of instructions that resets all the state. Refining the model into different components may enable some optimisation, though.

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

For state that requires flushing, correct application of the flush is also a functional property, provided that that the flushing operation itself is implemented in a constant-time manner (we will return to this assumption shortly).

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.

Once timing-channel reasoning is reduced to the verification of functional properties, it can then be integrated into seL4's existing invariants and thus extend its existing proof of storage-channel freedom [Murray et al. 2013] to also cover timing channels.

Returning now to our assumption that the flush instructions themselves are constant time – this is obviously a dangerous assumption, at least for flushing the L1-D cache; this latency will depend on the number of dirty lines. This means that we will have to pad the flush to its worst-case latency. This latency must also be provided by the enhanced hardware-software contract. Assuming it is known, 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.

A final complication is that not all OS-level state can be partitioned (as we observed in Section 2): the per-domain OS images need some data structure to coordinate. We will need to prove that accesses to those data structure are not history

dependent (where the latency variance cannot be lumped into the padding).

We can remove the history dependence of the access latencies by pre-fetching all such data into the L1 cache *inside* the time-padded code, thus ensuring determinism of the kernel exit latency. Proving this correct will require a degree of reasoning outside the standard verification framework, and some deeper reasoning about the data cache. This part likely presents the biggest challenge in verifying time protection.

### 5.2 Required 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 time model, which captures how far time advances on each execution step, is defined as a *deterministic yet unspecified* function of the microarchitectural state.

This construction neatly reflects the basic assumptions that underpin the augmented hardware-software contract (Section 3.3): (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 Achieving Verified Time Protection

With these models in hand, we can then prove 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 implemented to take a constant amount of time (e.g. via padding). These proofs can then be integrated with existing storage-channel freedom proofs to derive the absence of timing channels, effectively extending the existing arguments about storage channels to also cover the microarchitectural state on which timing channels depend.

## 6 RELATED WORK

Page colouring is a well established technique for partitioning physically-addressed caches [Liedtke et al. 1997; Lynch et al. 1992] and Ge et al. [2019] demonstrated that its performance impact is low. Liu et al. [2016] used Intel's cache allocation technology (CAT), which partitions by cache ways, to provide side-channel-protected "safe" memory, but this is a discretionary mechanism that does not protect against Trojans.

Flushing of stateful hardware is also well established [Godfrey and Zulkernine 2013; Guanciale et al. 2016; Osvik et al. 2006; Zhang and Reiter 2013]. Proposals to extend partitioning to on-core state [Domnister et al. 2012; Wang and Lee 2007] have not gained traction, presumably because the cost in terms of hardware utilisation is too high. Tiwari et al. [2009] suggested a leasing approach to share hardware resources between threads, which guarantees bounds on resource usage and side effects. Tiwari et al. [2011] later proposed a solution for top-to-bottom information flow guarantees, including a Star-CPU, a microkernel, and an I/O protocol. This is a fairly radical departure from mainstream computing architectures and manufacturers of commodity hardware will be hard to convince to go down this route.

We expect to extend formalisations of the ISA such as the L3 model of the Arm architecture by Fox and Myreen [2010]. Recently, the Sail formalisation of Armstrong et al. [2019] was adopted as the primary specification of the RISC-V architecture. These ISA models by definition omit any microarchitectural details. Syeda and Klein [2018] developed a formal model of the Arm TLB which is more detailled than what our approach requires, and thus indicates the feasibility of the formalisations we need. While prior work has proved notions of information flow control for operating system designs whose formal models include a notion of execution time [Verbeek et al. 2015], such reasoning has so far been performed only in the abstract, in the absence of formal models of the microarchitectural state on which execution time depends and without application to a verified kernel implementation.

## 7 CONCLUSIONS

We assert that it should be possible to eliminate microarchitectural timing channels completely, and prove it. Getting there requires overcoming a number of challenges. Firstly, our proof-of-concept implementation of time protection has shown that the approach is effective where hardware supports (spatially) partitioning or scrubbing such state; but it has also shown that present hardware contains unpartitionable state that cannot be flushed. This calls for an improved hardware-software contract that supports timing security, and we are working with the RISC-V community to develop such a contract and conforming hardware.

Secondly, a bunch of mechanisms do not yet make a usable operating system, and further work is required on developing a model that supports time security for real-world use.

Thirdly, more work is required for formalising suitable abstractions of the microarchitecture, and feed those into a verification framework such as the one that was used to prove storage-freedom of seL4. The key here is to transform timing channels into storage channels by linking them to the underlying microarchitectural state.

We are confident that this is all achievable and are actively working on it.

## ACKNOWLEDGMENTS

This work is supported by the Australian Research Council (ARC) through grant DP190103743, as well as the US Air Force Asian Office for Aerospace Research and Development (AOARD).

## REFERENCES

ARINC 2012. *Avionics Application Software Standard Interface*. ARINC. ARINC Standard 653.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2019), 71:1–71:31.

Leonid Domnister, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012).

Anthony Fox and Magnus Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proceedings of the 1st International Conference on Interactive Theorem Proving (Lecture Notes in Computer Science)*, Matt Kaufmann and Lawrence C. Paulson (Eds.), Vol. 6172. Springer, Edinburgh, UK, 243–258.

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: the Missing OS Abstraction. In *EuroSys Conference*. ACM, Dresden, Germany, 17. https://ts.data61.csiro.au/publications/csiro_full_text//Ge_YCH_19.pdf

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. https://ts.data61.csiro.au/publications/csiro_full_text//Ge_YCH_18.pdf

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. https://ts.data61.csiro.au/publications/csiro_full_text//Ge_YH_18.pdf

Michael Godfrey and Mohammad Zulkernine. 2013. A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud. In *Proceedings of the 6th IEEEInternational Conference on Cloud Computing*. Santa Clara, CA, US.

Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. San Jose, CA, US, 38–55.

Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can We Prove Time Protection?. In *Workshop on Hot Topics in Operating Systems (HotOS)*. ACM, Bertinoro, Italy, 23–29. https://ts.data61.csiro.au/publications/csiro_full_text//Heiser_KM_19.pdf

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.

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, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, USA, 207–220. https://ts.data61.csiro.au/publications/nicta_full_text/1852.pdf

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. https://ts.data61.csiro.au/publications/nicta_full_text/4631.pdf

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. https://ts.data61.csiro.au/publications/csiro_full_text//Kocher_HFGGHHLMPSY_19.pdf

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

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*. IEEE, Barcelona, Spain, 406–418. https://ts.data61.csiro.au/publications/nicta_full_text/8984.pdf

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

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-Context Capabilities: A Principled, Light-Weight OS Mechanism for Managing Time. In *EuroSys Conference*. ACM, Porto, Portugal, 14. https://ts.data61.csiro.au/publications/csiro_full_text//Lyons_MAH_18.pdf

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. https://ts.data61.csiro.au/publications/nicta_full_text/6464.pdf

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.

Hira Syeda and Gerwin Klein. 2018. Program Verification in the Presence of Cached Address Translation. In *International Conference on Interactive Theorem Proving*, Vol. 10895. Lecture Notes in Computer Science, Oxford, UK, 542–559. https://ts.data61.csiro.au/publications/csiro_full_text//Syeda_Klein_18.pdf

Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. 2009. Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference. In *Proceedings of the 42nd ACM/IEE International Symposium on Microarchitecture*. New York, NY, US.

Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th International Symposium on Computer Architecture*. San Jose, CA, US.

Freek Verbeek, Oto Havle, Julien Schmaltz, Sergey Tverdyshev, Holger Blasum, Bruno Langenstein, Werner Stephan, Burkhart Wolff, and Yakoub Nemouchi. 2015. Formal API specification of the PikeOS separation kernel. In *NASA Formal Methods Symposium*. Springer, 375–389.

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

Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Luca Benini, and Gernot Heiser. 2020. Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. ACM, Valencia, Spain, 7. https://ts.data61.csiro.au/publications/csiro_full_text//Wistoff_SGBH_20.pdf

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.

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

Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*. Berlin, DE, 827–838.