Two interfaces for seL4 kernel verification: Time Protection and the Kernel–Userland Gap (Extended Abstract)

Rob Sison

UNSW Sydney
Sydney, Australia
r.sison@unsw.edu.au

Abstract. This talk will present two case studies on verification interfaces at both ends of the seL4 operating system kernel's bulk of formal specifications in Isabelle/HOL. Reaching deep into the concrete depths, our efforts to prove that seL4 enforces *time protection*, the absence of timing leaks through microarchitectural state, rely entirely on a new hardware–software contract. Meanwhile, up where verified user-level programs would need to rely formally on a suitable abstraction of seL4's system call behaviour, there stands a long-unbridged *kernel–userland gap*: what developers would expect from reading seL4's reference manual is not quite yet what's been proved. We will examine how the urgent need to clarify both interfaces arose and what we are doing about it at UNSW, offered as data points for discussion on what it can take to provide a trustworthy body of software through formal methods and mechanised verification in Isabelle.

1 Time protection verification

Timing attacks like Spectre [13], Prime+Probe [18], Flush+Reload [29], Evict+Time [1], and many others [8] use computing hardware's microarchitectural features to exfiltrate secret data between separate user-land processes, in violation of any access control policy the OS kernel is meant to enforce. These attacks exploit that, if the OS is not careful to isolate the use of the many hardware features that support the CPU's performance, then a Trojan process T can signal a Spy process S even when T's and S's physical memory regions are supposedly disjoint. In these situations, solely by varying its own actions, T can impact how long S takes to execute its own code or access its own data.

To understand why, we must examine these hardware features. For each CPU core is a host of on-core features like branch target buffers (BTB) for speculative execution, translation lookaside buffers (TLB) for faster address translation, and specialised on-core (L1) memory caches for data and code, typically supported also by larger but slower off-core (L2 and L3) memory caches – all to improve runtime performance. If the OS pays no regard for the contents of these caches when context switching from T to S, then clearly T's own mere accesses to its own memory can impact S's running times by modifying what is or is not loaded into these caches. But while unfortunate, this is understandable because by definition, such hardware features *are* microarchitectural in the sense that they exist below the traditional hardware–software application binary interface (ABI) that has always been agnostic to how fast the hardware gets things done.

Given this state of affairs, is OS-enforced isolation of access to microarchitectural hardware features and their resulting timing effects possible? In proposing, developing for seL4, and empirically evaluating the effectiveness of a new OS feature of *time protection* to enforce such isolation, Ge et al. [6, 7, 9] found the answer is yes – and no. Yes, they found that OS-enforced time protection is in practice possible and measurably effective, to the extent that the hardware provides some basic support; but no, in that modern hardware is largely missing at least some of the needed supporting features. Thus, they argued for a new *hardware–software contract* that includes these features so as to make timing isolation possible. What is this contract?

The general principle identified by Ge et al. [7] is that, to achieve time protection, all parts of the microarchitectural state need to be either *spatially partitionable* – given only to T or to S with exclusive access – or *temporally partitionable*: the OS needs to be able to make deterministic all the effects of T's actions before context switching to S. Note the latter includes both *flushing* the relevant state contents and *padding* (waiting) until the worst-case execution time (WCET) of the context switch to wake S up.

How does modern hardware stack up on this rubric? In short, the work at our research group from Ge et al. [9] onwards has found that most of the problems with modern microarchitectures stem from (1) inflexibility when it comes to support for spatial partitioning, and (2) an inability to temporally partition the state when spatial partitioning is not possible. First, which microarchitectural elements are spatially partitionable?

- Yes, but...: Off-core memory caches. The L2 and L3 caches on modern hardware are typically large enough to be spatially partitionable using a technique called *cache colouring* [2, 11, 15] – being indexed by physical address, the OS can be careful to assign disjoint address regions to S and T serviced by disjoint sets of cache lines.
  However: Each given cache architecture dictates the maximum number of colours, which can only divide it into regions of equal size in fixed positions, leading to poor memory utilisation. It is costly to set colours aside for *shared kernel data* or other regions shared between processes to avoid expensive copying between colours [21], and copied or not, all the affected cache lines need to be flushed on context switch.
- No: On-core caches and buffers. Unlike the L2 and L3, the L1 caches and other on-core features (BTB, TLB, etc.) are not spatially partitionable, due to their small size and being indexed by virtual addresses that freely overlap between S and T.

Consequently, the OS needs to be able to flush *all* the on-core state on context switch from T to S, and needs a reliable method to wait until a deterministic time before waking S up. However, Ge [6] found that x86 and both Arm platforms evaluated were missing any primitive to flush many of the on-core state elements, and in some cases provided no means to disable the corresponding feature, leaving some timing channels impossible to close. Furthermore, x86 had no instruction to flush the L1 cache alone without also flushing every level of cache, an overly costly measure. Finally, though Ge [6] fortunately found software-based time padding to be measurably effective, no platform provided a hardware-supported means of waiting until a deterministic time.

Since Ge et al.'s pioneering investigations, we have been hard at work formalising the above requirements [3, 22] and integrating them [4, 14, 17] into the seL4 kernel's existing correctness, security and refinement proofs [20] – all in Isabelle/HOL – so we

can prove that seL4 enforces time protection, provided it has access to hardware primitives that meet them. This process has forced us to clarify these requirements both to make provable in Isabelle a new security property for seL4 that includes time protection, as well as to accommodate new time protection features in the pipeline – notably, recent support under development for *time-protected cross-domain communication* [21] that prevents backwards flows against a one-way "data diode" policy.

In general, we have found that dedicated hardware primitives have made the formal verification of time protection *feasible* compared to verifying the corresponding software-based approaches devised for legacy hardware. For example, the software-based alternative to hardware-based flushing is *prefetching*, as used in places by Ge [6], which we have found infeasible to verify as effective without an accurate internal model of all caches' replacement policy. Arguably, this would be a rather intrusive and interface-breaking ask for a manufacturer to provide, compared to a flush primitive. A consequence of this is that, beyond requiring flush primitives for all on-core state, we need more flexible flush primitives for selected partitions of the off-core caches. This is crucial for shared kernel data handling and to enable time-protected shared memory as prototyped by Sethu [21], but without prefetching so it is verifiable.

Meanwhile, thanks to our group's collaboration with ETH Zürich on the RISC-V Cheshire platform, based on CVA6 (formerly Ariane) developed by Wistoff et al. [25–27], we now have all three of the above-mentioned missing hardware features: a flush primitive `fence.t` for *all* on-core state that *also* pads until a configured deterministic time, as well as a more flexible *dynamically partitionable last-level cache* (DPLLC) that allows configuring partitions of varying sizes and a primitive to selectively flush them individually. Our hope is that with basic hardware support of a new hardware–software contract for time protection thus demonstrated as feasible, effective and performant, the value-add in terms of both measurable and verifiable time protection will be clear.

2 Kernel–userland verification gap

We now turn to the question of whether the seL4 kernel itself provides the interfaces needed to support the verification of userland programs running on top of it.

The answer, again is yes – and no, though in slightly different senses to last time. Yes, in the sense that, as verification by Paturel et al. [19] (further documented by Weibel et al. [24]) of the user-level seL4 Microkit [23] library's main message handler loop showed, such verification is possible assuming seL4 satisfies certain *Hoare triple* [10] (precondition–postcondition) requirements across the system calls it provides. The wrinkle in the story is that seL4 has never been verified to satisfy these Hoare-triple assumptions. Although famously the world's first formally verified OS kernel [12], seL4 has been proved functionally correct in the sense that there exists an abstract specification of the kernel's correct behaviour including that it does not crash, and a refinement proof showing seL4's C code faithfully implements that specification. What is missing, however, is any formal proof that seL4's abstract specification *of its correctness* actually meets the Hoare triples implied by the English-language descriptions of its system call behaviour that can be found in the seL4 reference manual [5].

Our investigations have found the problem seems to go further than just the seL4 kernel: apparently, no OS kernel verification effort in the literature to date has yet attempted to verify such Hoare triples *for system calls that have to block waiting for another process* before returning to their original caller. A prime example of this is the blocking `seL4_Recv` system call, which only returns to the Microkit's main handler loop once an entirely different process calls a `seL4_Signal` system call that unblocks it. In comparison, although the Verve project [28] demonstrated a proof composability outcome of the kind we ultimately desire across the kernel–userland gap – though in their case, across the divide between its Nucleus and the rest of the Verve kernel – its Nucleus exported only nonblocking functions. The only such case for a blocking system call of any kind appears to be from the CertiKOS project [16], which verified a system call that blocks waiting on I/O, but not on the actions of another user process.

In solving this problem, there are two tasks at hand: Apart from proving new facts (as Hoare triple lemmas) about the seL4 kernel's existing abstract specification in Isabelle that would be expected by the userland processes, we also need to compose them into a single Hoare triple when a system call can span multiple entries into the kernel.

First, by "new facts" I mean that, for example, to prove the needed Hoare triple for `seL4_Recv` – even for its nonblocking case, where there *is* already a signal ready to receive – we found we needed to prove new lemmas that are largely more specific than the existing ones when it comes to state relevant to the system call. Furthermore, these proofs must also establish that, when handling `seL4_Recv`, the seL4 kernel does *not* modify any other state the caller can see but is expecting the system call not to change.

Then, in addition to proving new facts about individual entries into the kernel, we also need to compose them when system calls block waiting for the actions of another process. Taking for example `seL4_Recv` as called when there is *no* signal ready to receive, this involves at least three kinds of kernel entries: first, we have (1) the initial `seL4_Recv` that blocks the calling process and (2) the final `seL4_Signal` by another process that unblocks it. Then, between the two, we must account for their composition with (3) any finite sequence of system calls, interrupts, or other kernel entry events that have nothing to do with the original system call. For composing the initial and final call with these arbitrary sequences, we expect new facts proved about individual kernel entries *not* modifying any other state irrelevant to the task at hand to play a crucial role.

With these proofs almost complete for some motivating cases, we hope to show that, despite previously being unsolved, it is eminently feasible to provide such proofs meeting the kinds of formal requirements on system calls that userland verification efforts would expect of OS kernels. Ongoing work will extend the proofs to more of seL4's system calls, initially prioritising those used by the seL4 Microkit libraries [23] as relied on by its formal verification [19, 24]. In the long run, further such verification can proceed to formalise the informal descriptions provided in seL4's reference manual [5].

Acknowledgements

This abstract describes ongoing work funded by Cyberagentur via the EVIT program and also funded by the Foresight Institute. Thank you also to Julia Vassiliki and Gernot Heiser for feedback on earlier drafts of this abstract.

Bibliography

[1] Acıiçmez, O., Koç, c.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: Crytographers' track at the RSA Conference on Topics in Cryptology. pp. 225–242. Springer (2007)

[2] Bershad, B.N., Lee, D., Romer, T.H., Chen, J.B.: Avoiding conflict misses dynamically in large direct-mapped caches. In: International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 158–170. ACM, San Jose, CA, US (Oct 1994)

[3] Buckley, S., Sison, R., Klein, G.: An Isabelle/HOL formalisation of microarchitectural timing channel prevention by operating systems - VM artifact and proof release (Nov 2022), https://doi.org/10.5281/zenodo.7343912

[4] Buckley, S., Sison, R., Wistoff, N., Millar, C., Murray, T., Klein, G., Heiser, G.: Proving the absence of microarchitectural timing channels. arXiv preprint arXiv:2310.17046 (2023)

[5] seL4 Foundation: seL4 Reference Manual, Version 15.0.0 (Mar 2026)

[6] Ge, Q.: Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection. Ph.D. thesis, UNSW, Sydney, Australia (Oct 2019)

[7] Ge, Q., Yarom, Y., Chothia, T., Heiser, G.: Time protection: the missing OS abstraction. In: EuroSys Conference. ACM, Dresden, Germany (Mar 2019)

[8] Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**, 1–27 (Apr 2018)

[9] Ge, Q., Yarom, Y., Heiser, G.: No security without time protection: We need a new hardware-software contract. In: Asia-Pacific Workshop on Systems (APSys). ACM SIGOPS, Korea (Aug 2018)

[10] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**, 576–580 (1969)

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

[12] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles. pp. 207–220. ACM, Big Sky, MT, USA (Oct 2009)

[13] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Haburg, M., Lipp, M., Mangard, S., Prescher, T., Schwartz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: IEEE Symposium on Security and Privacy. pp. 19–37. San Francisco, CA, US (2019)

[14] Liang, T.: Refining seL4's Accounting of Touched Addresses for Time Protection. BSc(Hons) thesis, School of Computer Science and Engineering, Sydney, Australia (Nov 2025)

[15] Liedtke, J., Härtig, H., Hohmuth, M.: OS-controlled cache predictability for real-time systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 213–223. IEEE, Montreal, CA (Jun 1997)
[16] Mansky, W., Honoré, W., Appel, A.W.: Connecting higher-order separation logic to a first-order outside world. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 12075, pp. 428–455. Springer (2020), https://doi.org/10.1007/978-3-030-44914-8_16
[17] Nair, S.: Updating L4v Invariants to Aid Time Protection Proofs. BSc(Hons) thesis, School of Computer Science and Engineering, Sydney, Australia (Nov 2025)
[18] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Crytographers' track at the RSA Conference on Topics in Cryptology. pp. 1–20. Springer, San Jose, CA, US (2006)
[19] Paturel, M., Subasinghe, I., Heiser, G.: First steps in verifying the seL4 Core Platform. In: Asia-Pacific Workshop on Systems (APSys). ACM, Seoul, KR (Aug 2023)
[20] seL4 microkernel code and proofs, https://github.com/seL4/
[21] Sethu, V.: A Usable System Model for Time Protection. BE thesis, School of Computer Science and Engineering, Sydney, Australia (Nov 2025)
[22] Sison, R., Buckley, S., Murray, T., Klein, G., Heiser, G.: Formalising the prevention of microarchitectural timing channels by operating systems. In: International Symposium on Formal Methods (FM). Springer, Lübeck, DE (Mar 2023)
[23] Velickovic, I.: The seL4 Microkit (Sep 2023), https://trustworthy.systems/publications/papers/Velickovic_23:sel4s.abstract.pml, talk at the 5th seL4 Summit
[24] Weibel, T., Kocsis, Z.A., Paturel, M., Sison, R., Subasinghe, I., Heiser, G.: Verifying the seL4 Microkit. https://trustworthy.systems/publications/papers/Weibel_KPSSH_24.pdf (Jun 2024)
[25] Wistoff, N., Heiser, G., Benini, L.: fence.t.s: Closing timing channels in high-performance out-of-order cores through ISA-supported temporal partitioning. In: International Conference on Applications in Electronics Pervading Industry, Environment and Society (ApplePies). Springer, Turin, IT (Sep 2024)
[26] Wistoff, N., Schneider, M., Gürkaynak, F., Benini, L., Heiser, G.: Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core. In: Design, Automation and Test in Europe (DATE). IEEE, virtual (Feb 2021)
[27] Wistoff, N., Schneider, M., Gürkaynak, F., Heiser, G., Benini, L.: Systematic prevention of on-core timing channels by full temporal partitioning. IEEE Transactions on Computers **72**(5), 1420–1430 (2023)
[28] Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 99–110. ACM, Toronto, Ont, CA (Jun 2010)
[29] Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium. pp. 719–732. USENIX, San Diego, CA, US (2014)