Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning

Nils Wistoff, *Student Member, IEEE,* Moritz Schneider, *Student Member, IEEE,* Frank K. Gürkaynak, Gernot Heiser, *Fellow, IEEE,* and Luca Benini, *Fellow, IEEE*

Abstract—Microarchitectural timing channels enable unwanted information flow across security boundaries, violating fundamental security assumptions. They leverage timing variations of several state-holding microarchitectural components and have been demonstrated across instruction set architectures and hardware implementations. Analogously to memory protection, Ge et al. [1] have proposed *time protection* for preventing information leakage via timing channels. They also showed that time protection calls for hardware support. This work leverages the open and extensible RISC-V instruction set architecture (ISA) to introduce the temporal fence instruction `fence.t`, which provides the required mechanisms by clearing vulnerable microarchitectural state and guaranteeing a history-independent context-switch latency. We propose and discuss three different implementations of `fence.t` and implement them on an experimental version of the seL4 microkernel [2] and CVA6, an open-source, in-order, application class, 64-bit RISC-V core [3]. We find that a complete, systematic, ISA-supported erasure of all non-architectural core components is the most effective implementation while featuring a low implementation effort, a minimal performance overhead of less than 1 %, and negligible hardware costs.

Index Terms—timing channels, covert channels, security, computer architecture, microarchitecture.

1 Introduction

Computing systems are trusted with an increasing amount of sensitive information and a large number of safety- and security-critical tasks. Some examples include personal computing, industrial and military applications, and critical infrastructure. To prevent unauthorised and malicious access, computer architects have established a set of mechanisms that allow operating systems to isolate concurrent applications through *memory protection*, creating a security boundary.

As the Spectre attacks have prominently demonstrated [4] and several other attacks have confirmed since [5], [6], memory protection is insufficient for a complete isolation of applications. Concurrently running applications compete for shared hardware resources, which may result in timing variations of one application due to another application's execution. These timing differences can be leveraged to transfer information between applications, bypassing the security boundary. We refer to such information channels as *microarchitectural timing channels*.

Ge et al. have proposed supplementing existing memory protection with *time protection* to prevent such timing channels [1]. The key idea is to partition all shared hardware resources—either spatially or temporally—to prevent interference between concurrent applications. While they show that off-core resources (such as last-level caches) can be efficiently spatially partitioned from software, partitioning on-core resources requires hardware support beyond that specified in current instruction set architectures (ISAs).

In this work, we leverage the open and extensible RISC-V ISA to propose a streamlined and ultra-low overhead ISA extension and hardware implementation for on-core time protection. Specifically, we make the following contributions:

- We demonstrate the presence of serious timing channels even in an in-order RISC-V core and confirm previous claims [7] that the current ISA lacks mechanisms to close them.
- We propose the *temporal fence* instruction, `fence.t`, which allows software to partition on-core micrarchitectural resources.
- For implementing `fence.t`, we propose MicroReset, a systematic erasure of all non-architectural processor state. We compare MicroReset to two alternative implementations of `fence.t`: (i) A flush of all well-known vulnerable microarchitectural components, (ii) an exhaustive flush of an extended set of manually identified vulnerable state-holding components.
- We propose a context-switch with a history-independent latency, including said reset of microarchitectural state. This is of particular (but not exclusive) relevance for a write-back configuration of the L1 data cache.

We implement the proposed mechanisms and evaluate their efficacy and costs on CVA6, an open-source,

- *N. Wistoff, F. K. Gürkaynak, and L. Benini are with the Integrated Systems Laboratory (IIS), ETH Zürich, Switzerland. E-mail: {nwistoff,kgf,lbenini}@iis.ee.ethz.ch*
- *M. Schneider is with the Institute of Information Security, ETH Zürich, Switzerland. E-mail: moritz.schneider@inf.ethz.ch*
- *G. Heiser is with UNSW Sydney, Australia. E-mail: gernot@unsw.edu.au*
- *L. Benini also is with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, Bologna, Italy*

application-class, in-order, 64-bit RISC-V core [3] for different cache configurations, and on seL4, an open-source, secure microkernel with formal verification [2].[1]

Using benchmarks, we show that the temporal fence introduces a low performance overhead of less than 1 %. The hardware modifications do not impact the critical path and add a negligible area overhead of 0.13 %.

The remainder of this paper is structured as follows: Section 2 presents our threat model, the attack algorithm and the concept of *time protection* as a prevention methodology. We derive two security requirements that hardware needs to provide to enable time protection in Section 3. In Section 4, we propose different implementation approaches for time protection, which we evaluate in Section 5. We present the work related to this paper in Section 6 and conclude in Section 7.

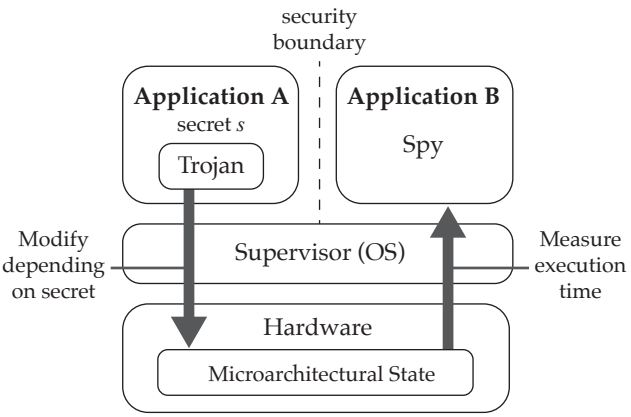


Figure 1: Threat model: a Trojan actively leaking data via shared hardware resources.

2 Background

2.1 Timing Channels

Information channels that let isolated applications communicate but are not intended for information transfer are called *covert channels* [8]. *Microarchitectural timing channels* are covert channels that leverage the timing behaviour of microarchitectural components to transfer information. They may generally occur whenever multiple applications compete for shared hardware resources [9]. Famous examples for exploitable hardware resources are data caches, where the latency of a memory access varies depending on how the cache was previously used [10], [11]. Attacks were also demonstrated for further components such as instruction caches [12], branch predictors [13], and translation lookaside buffers (TLBs) [14].

2.2 Threat Model

We examine covert-channel leakage under a *confinement* scenario [8]: An untrusted program possesses a secret, and the operating system (OS) encapsulates the program's execution in a security domain that only allows communication across defined channels to trusted components (e.g., an encryption service). The untrusted program contains a Trojan that is actively trying to leak the secret via a covert channel. Note that a Trojan could not only hide in malicious code, but can be constructed by control-flow hijacking of innocent code through exploiting bugs or speculatively executed gadgets as in a Spectre attack [4]. A second, unconfined, and also untrusted security domain contains a spy which is trying to read the secret leaked by the Trojan. This setup is illustrated in Figure 1.

The *intentional* leakage by the Trojan represents the worst case. If we can prevent this attack, we preclude any other leakage using the same mechanism including *side channels*, where leakage originates from an unwitting victim rather than a Trojan.

We assume that the Trojan and spy time-share one processor core, meaning cross-core leakage is out of the scope of this paper. [2] We only consider microarchitectural timing channels. Covert channels that abuse other characteristics, such as power draw, are not covered in this work.

2.3 Time Protection

Time protection is a principled approach to *preventing* timing channels [1]. While the established notion of *memory protection* prevents interference between security domains through unauthorised memory accesses, time protection aims to prevent interference that affects observable timing behaviour.

Time protection requires that all shared hardware resources, including non-architectural ones, must be partitioned between security domains, either temporally (secure time multiplexing) or spatially. Ge et al. show that (physically-addressed) off-core caches can be effectively partitioned through *cache colouring* [15], which leverages the associative cache lookup to force different partitions into disjoint subsets of the cache. They demonstrate that colouring is effective in preventing cache channels in both intra-core and cross-core attacks and comes with low overhead.

Spatial partitioning is generally impractical for on-core resources. For performance reasons, on-core resources are limited and are designed to be well utilised by a single program, so partitioning approaches usually result in unacceptable performance degradation. Furthermore, on-core resources are generally indexed by virtual addresses, which cannot be coloured by the OS. This leaves temporal partitioning as the only viable approach for on-core resources.

3 Security Requirements

Based on the findings of Ge et al. [1], we propose two security requirements for on-core time protection by temporal partitioning.

First, before handing a resource to a different domain, it must be brought to a state that is independent of execution history. Therefore, the OS must have the means to clear all microarchitectural state, which in practical terms requires an extension to the hardware-software contract to refer (in an abstract way) to such non-architectural state. Ge et at. specifically show that contemporary Intel and Arm processors lack the mechanisms required for implementing time protection [7].

1. We use an experimental version with time protection support that has not yet been formally verified.

2. The prevention of cross-core leakage is discussed further by Ge et al. [1].

Requirement 1. For temporal partitioning, hardware must provide a (set of) mechanism(s) that allow clearing all non-architectural state that depends on previous execution and may impact future timing.

A second requirement for time protection is a secret-independent context switch latency. This requirement is particularly (but not exclusively) relevant for processors featuring a write-back L1 cache: before we can reset this component, we need to write back all dirty cache lines. The number of dirty cache lines, and hence the latency of said reset and the whole context switch routine, directly depends on previous execution and can be probed to transfer information.

Requirement 2. The latency of the context switch routine, including the reset of the non-architectural state mentioned above, needs to be independent of the previous execution.

4 Implementing On-Core Time Protection

In the following, we will present different approaches to implement time protection for shared on-core hardware resources. We will use the RISC-V architecture for our analysis because of its openness, extensibility, and availability of modifiable open-source implementations such as CVA6 (see Section 5.1). Section 4.1 will describe a baseline system where time protection will be implemented using existing resources on unmodified hardware without additional architectural features. In Section 4.2, we will propose an ISA extension that provides software with the necessary means to partition shared on-core hardware resources temporally (Requirement 1 of on-core time protection), and we will discuss three different implementation approaches. Finally, Section 4.3 will address Requirement 2 of time protection, enabling a context-switch latency that does not depend on execution history.

4.1 Baseline Architecture

Ge et al. [1] report that neither the x86 nor the Arm architecture provides sufficient mechanisms for implementing time protection. Arm provides targeted L1 cache flushes but no mechanism for flushing other microarchitectural state. The x86 architecture provides *branch control mechanisms* for clearing the state of the branch predictors [16]. For partitioning the L1 cache on this architecture, the authors implemented software flushing by touching all cache lines, similar to the prime phase of the prime-and-probe attack. Such an approach is expensive and obviously brittle, as it must make assumptions on the replacement policy which may not hold in reality. Unsurprisingly, they find that this defence is incomplete, leaving residual channels that the OS is unable to close.

With RISC-V, the situation is presently worse, as the specification of cache management is still under discussion. While implementations generally support some cache management, this is not yet standardised. To explore this aspect, we implement a "software only" defence (in the following referred to as *Software*, SW), where the OS uses only mechanisms defined in the ISA as presently specified. This basically forces the OS to resort to the priming approach in an attempt to erase any microarchitectural state left by the Trojan's execution.

4.2 Temporal Fence Instruction

As we show in Section 5, this "software only" defence is insufficient since some microarchitectural timing channels remain. Moreover, it comes at a great performance overhead. Therefore, we propose the *temporal fence* instruction, `fence.t`, to let an OS access the hardware mechanisms required for time protection. We note that other semantics to realise the temporal fence are conceivable as well: for instance, a combination of multiple instructions and registers could be used. For our proof of concept, we stick to a single `fence.t` instruction that both triggers the reset of vulnerable microarchitectural state (Requirement 1) and guarantees a history-independent context switch latency (Requirement 2). Except for increased cycle and instruction counters, `fence.t` has no architectural effects. For evaluation purposes, we encode `fence.t` as a U-type RISC-V instruction with the opcode *custom-0*. `fence.t` optionally takes a 20-bit immediate value, which is a bitmap selecting the components that should be reset.

In the following, we will present three different implementation strategies for `fence.t`.

4.2.1 Basic Flush

In the first version of `fence.t`, we exclusively flush the principal components used by the prime-and-probe attack in Section 5.2.1: the L1 data and instruction caches, the TLBs, and the branch predictors (branch history table (BHT), branch target buffer (BTB)). If present, prefetchers are cleared and write-buffers are drained. We write back any dirty state (for write-back caches), invalidate the caches and TLBs, and purge the branch predictors. To preserve the computational correctness of in-flight instructions, we also flush the pipeline.

In the following, we will refer to this approach as the *Basic Flush* (FLUSH_1).

4.2.2 Full Flush

Motivated by our security analysis of the basic flush in Section 5.2, we identify several secondary, stateful components deeply embedded in CVA6 with a possible timing impact. In particular, these are

- a linear-feedback shift register (LFSR) per cache (L1 data, L1 instruction) that generates a pseudo-random number sequence for the cache-replacement policy,
- a pseudo-least-recently-used (pseudo-LRU) tree in each TLB that identifies a replacement candidate,
- a round-robin memory arbiter that arbitrates cache accesses between the load unit, the store unit, and the memory-management unit,
- two round-robin arbiters in the write buffer of the write-through L1 data cache, which choose an entry to serve (lookup or write back) next.

We extend `fence.t` to a *full flush* (FLUSH_2) by adding the support to clear the state of these components as well.

4.2.3 Microreset

Finally, we propose a principled and systematic approach to enforce complete temporal partitioning. The key idea is to clear *all* on-core state that is not architectural by

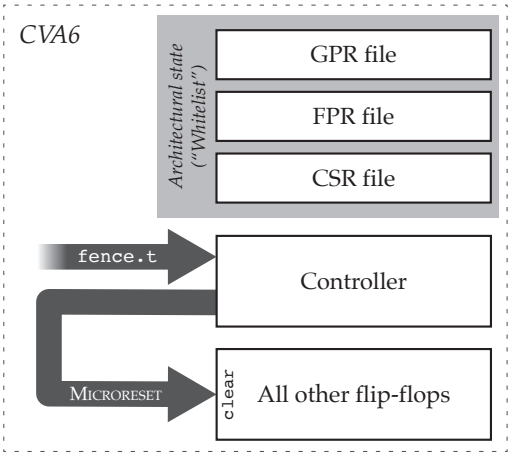


Figure 2: Illustration of the MICRORESET.

default, and explicitly exclude architectural state. We call this mechanism MICRORESET, as it exclusively resets non-architectural microarchitectural state.

All flip-flops in the design are extended by an additional `clear` input. By asserting this input on a `fence.t`, we can guarantee that all state on flip-flops in the design is set back to a predefined state. On-core state that is not resettable (such as SRAMs) must be cleared separately. To ensure computational correctness, architectural state needs to be retained, either by saving it before MICRORESET (e.g. write-back of an L1 cache) or by explicitly excluding it from the MICRORESET. Hence, we design the `fence.t` controller to proceed in the following six steps:

Step 1. Save the program counter.

To resume execution after MICRORESET from the correct location, we store the address of the instruction following `fence.t` in a register that is excluded from MICRORESET (we consider this architectural state).

Step 2. Save locally modified (dirty) architectural state.

In particular, this concerns components such as write-back L1 caches: to preserve the contents of dirty cache lines, we need to write them back before clearing the cache. As we will discuss later, this is the most costly step of `fence.t`.

Step 3. Drain pending transactions.

Next, we need to wait for all pending external transactions to complete without issuing or accepting new ones. This way, we prevent violating any handshake protocols or losing data that we began to write back in the previous step.

Step 4. Clear components that are not cleared on reset.

Not all components can be fully reset to a predefined state. For instance, SRAMs such as those found in caches are generally not resettable. To temporally partition these components, they need to be cleared separately, e.g. by a finite-state machine (FSM) that overwrites their contents line by line.

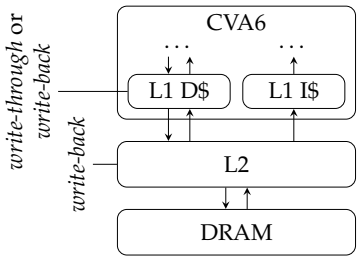


Figure 3: Hardware platform.

Step 5. Assert MICRORESET.

We now clear all flip-flops containing non-architectural state. For this purpose, we assert the `clear` input of all flip-flops in the design. We explicitly exclude flip-flops that hold architectural state—the state of these components is preserved during MICRORESET and saved/restored explicitly by the OS during a context switch. This approach removes the risk of omitting state that could create a timing channel.

While identifying the architectural state is potentially one of the biggest challenges of this approach, for CVA6, it turned out relatively straight-forward: we explicitly exclude the integer and floating-point register files, the control and status register (CSR) file, and the controller, which is driving MICRORESET. Figure 2 shows the resulting setup.

In other designs, such as out-of-order cores with merged register files, identifying the architectural state might be more challenging but should, in general, still be feasible with reasonable effort.

Step 6. Continue execution from saved program counter.

Finally, we de-assert the MICRORESET and continue fetching from the next program counter.

4.3 Time Padding

When introducing time protection in Section 3, we asserted that the context switch latency needs to be independent of previous execution (Requirement 2).

In RISC-V, the context switch routine is initiated by a timer interrupt of the core-local interrupt controller (CLINT). While this interrupt is generated at a fixed period independently of the microarchitectural state, any machine-mode or kernel code following it until the end of `fence.t` may be delayed by cache misses, mispredictions etc. Hence, to prevent a dependency of the context switch latency on previous execution, we pad the interval between the CLINT's timer interrupt and the completion of `fence.t` to a worst-case latency.

We implement this mechanism by adding a custom `cspad` CSR that takes a 32-bit value. We stall the completion of `fence.t` until `cspad` cycles after the CLINT timer interrupt. Padding can be disabled by setting `cspad` to 0.

5 EVALUATION

5.1 Hardware Platform

We evaluate the channels and defences on CVA6, an open-source, RV64GC, 6-stage RISC-V core developed at ETH Zürich and currently maintained by OpenHW Group [3].[3]

3. CVA6 is formerly known as ARIANE.

It is implemented in SystemVerilog and publicly available on GitHub [17]. It features three privilege levels and address translation, and thus supports full-fledged operating systems. Its configurability, simplicity, and openness make it a good candidate for architectural exploration.

Setup: We instantiate the CVA6 core on a Xilinx Kintex-7 FPGA (Digilent Genesys II), running at 50 MHz. We configure two versions of CVA6: one with a *write-through* L1 data cache and one with a *write-back* cache. Both versions feature an 8-way, 32 KiB L1 data cache and a 4-way, 16 KiB L1 instruction cache. The caches use 16-byte lines and a pseudo-random replacement strategy driven by an 8-bit LFSR. The L1 data cache is accessed by the load-, store-, and memory-management units, with concurrent accesses arbitrated with a round-robin policy. The branch predictor has a 64-entry BHT and a 16-entry BTB. There are two single-level, fully associative data and instruction TLBs, with 16 entries each, using a pseudo-LRU replacement policy. Our system-on-chip features a 512-KiB write-back L2 cache [18] that is connected to DRAM. Figure 3 shows the memory architecture.

We partition the L2 cache by colouring [15], which precludes channels in the memory backend and allows us to focus on channels resulting from on-core state.

5.2 Security Analysis

5.2.1 Prime and Probe

Techniques for exploiting covert channels are well established; for our scenario of intentional leakage, the *prime-and-probe* attack [11] is simple and effective. We stress that our proposed mechanism addresses the root cause of covert channels and therefore expect an equal efficacy for other attacks such as *evict-and-time* [19] and *evict-and-reload* [20].

In a prime-and-probe attack, the spy first forces the exploited hardware resource into a known state (*prime*). For the data cache it traverses a large buffer (in cache-line-sized strides for efficiency); for the instruction cache it executes a series of linked jumps. The TLBs are similarly primed by accessing or jumping with page-size strides.[4] The branch predictors are primed by a series of conditional branches (BHT) or by executing multiple indirect jumps (BTB). With a correctly-sized priming buffer, this leaves the hardware resources in a state where further accesses by the spy within the same address range are fast. This state is illustrated in the second element of Figure 4 where all entries in the buffer have been primed by the spy.

At the end of its time slice, the OS preempts the spy and switches to the application that contains the Trojan, which accesses a subset of the hardware resource to encode the secret. Given a cache of n lines, the Trojan can transmit a secret $s \leq n$, the *input signal*, by touching s cache lines, thereby replacing the spy's content. The resulting state is illustrated in the third element of Figure 4. Obviously, more complex encodings are possible to increase the amount of data transferred in a time slice (the channel capacity), but for our purposes, the simple encoding is sufficient, as we want to prevent *any* leakage.

4. This is a somewhat simplified description—in general, it is necessary to randomise the access order to prevent interference from prefetching, but that is not an issue on our processor.

When execution switches back to the spy, it again traverses (*probes*) the whole buffer, observing its execution time. Each entry replaced by the Trojan's execution leads to a cache miss, and results in an increase in probe time. If the latency of a hit is t_{hit} and that of a miss is $t_{\text{miss}} > t_{\text{hit}}$, the total latency increase is $s \cdot (t_{\text{miss}} - t_{\text{hit}})$. For our simple encoding scheme, the *output signal* is the total probe time, which is linearly correlated to the input signal. A more sophisticated encoding scheme could, for example, exploit the time measurements of each individual access and thus extract more information.

5.2.2 Measuring Leakage

We adopt the approach of Ge et al. [7] for quantifying and evaluating leakage and prevention strategies. For attack i, the Trojan encodes as input value a randomly chosen secret, s_i, and the spy subsequently measures as the output value its probe latency, t_i. s and t can be regarded as samples of the random variables S and T. A covert channel exploits the correlation of the two random variables: if the output t is correlated with the input s, there is a covert channel that transfers information from the Trojan to the spy.

We use a *sample size* (number of repeated attacks) of 1 million. For leakage we use a combination of two indicators: The *channel matrix* for visualisation and the *discrete mutual information* $\mathcal{M}$ as a quantitative metric.

5.2.2.1 Channel Matrix: The channel matrix represents the conditional probability of observing a particular output value, t, given input value s. The conditional probability distribution $p(t \mid s)$ can be computed directly from the measured sample pairs $\{(s_1, t_1), \ldots, (s_N, t_N)\}$.

We represent the channel matrix as a heat map: inputs vary horizontally and outputs vertically, and bright colours indicate high, dark colours low probability. A variation of colour along any horizontal line through the graph indicates a dependence of the output on the input, and thus a channel. For example, Figure 6a shows a clear diagonal pattern indicating a channel: if the spy observes a probe time of 85,000 cycles, it can infer with high confidence that the Trojan has encoded a value between *170* and *180*. Repeating the experiment can increase the spy's confidence.

5.2.2.2 Mutual Information: For quantifying channel capacity we use *continuous mutual information* $\mathcal{M}$, the amount of information gained about a random variable by observing another, possibly correlated random variable [21].

Intuitively, mutual information is the difference of the information gained by observing the random variable T *without* and *with* knowledge of the second random variable S. If both random variables are highly correlated (i.e., there exists a covert channel), the information gained by observing S is low and $\mathcal{M}$ high. Conversely, if both random variables are uncorrelated, $\mathcal{M} = 0$. The unit for $\mathcal{M}$ is bits; as most of our channel capacities are small, we use millibits ($1\,\text{mb} = 10^{-3}\,\text{b}$) in our measurements.

5.2.2.3 Zero Leakage Upper Bound $\mathcal{M}_0$: Since all measurements are affected by noise, $\mathcal{M}$ will mostly not be zero, even if there is no channel. We use a Monte Carlo simulation for estimating the apparent channel produced by this noise. Specifically, we pick uniformly random pairs of input and output values, and thus remove any correlation between them, while retaining their original value ranges

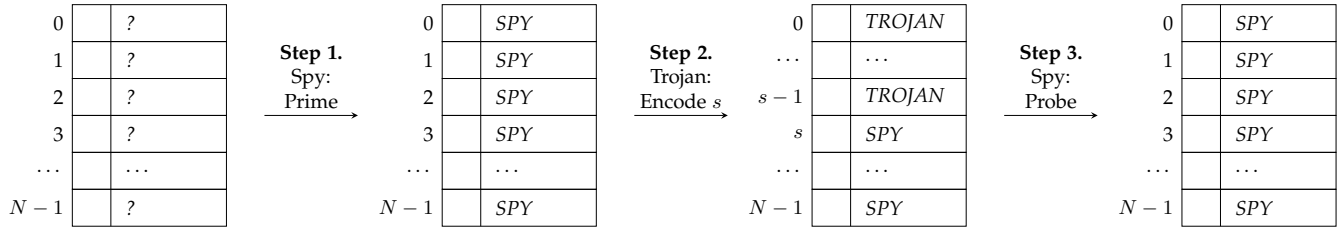


Figure 4: A prime-and-probe attack on a cache.

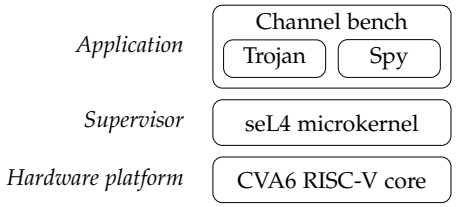


Figure 5: HW/SW stack of the evaluation framework.

and spreads. Any mutual information that is measured from this data can only be due to noise. We repeat this process 1000 times and then compute the 95%-confidence interval $\mathcal{M}_0$ for an experiment without a channel. Notably, $\mathcal{M}_0$ can differ strongly between experiments, as it depends on the range and distribution of the measured values. We conclude that a channel is present if $\mathcal{M} > \mathcal{M}_0$, otherwise, the result is consistent with no channel.

We use the leakiEst tool [22] to compute mutual information $\mathcal{M}$ and zero leakage upper bounds $\mathcal{M}_0$.

5.2.2.4 Testbench: Ge's CHANNEL BENCH [23], [24] provides a minimal OS and data collection infrastructure; we port it to RISC-V and adapt it to CVA6. CHANNEL BENCH uses attack implementations from the MASTIK toolkit [25], running on an experimental version of seL4 [2] that supports time protection. The resulting stack of our evaluation framework is shown in Figure 5.

5.2.3 L1 Data Cache

Figure 6 shows the result of CHANNEL BENCH for the write-through L1 data cache for different implementation approaches of time protection. We use the write-through L1 data cache as an example for an in-depth security analysis and comparison of the proposed mechanisms. Most of the following observations also hold for the other microarchitectural components.

5.2.3.1 Unmitigated: As a baseline, we use the original, unmodified CVA6 core and run our testbench without any further on-core time protection in seL4. Figure 6a shows the resulting channel matrix. A clear correlation between the Trojan's secret and the spy's execution time is visible, indicating the presence of a covert channel. This is confirmed by the mutual information $\mathcal{M}$, which is clearly above the zero-leakage upper bound $\mathcal{M}_0$ at more than 1.6 bit per iteration. To illustrate this channel's bandwidth, let us assume a 256-bit AES key, two concurrently running applications, and a time slice of 1 ms. The AES key could be leaked in less than 320 ms. More efficient encodings could achieve even higher throughput.

5.2.3.2 Mitigation Using Existing Architecture: We next evaluate the approach of 4.1, using only existing instructions to mitigate the timing channel. As the results in Figure 6b and Table 1 show, this decreases the channel's capacity without fully closing it. One explanation for this behaviour lies within the replacement policy of the data cache. CVA6 pseudo-randomly selects a cache entry for eviction in case of a collision. As a result, the OS cannot reliably evict all data cache entries on a context switch. It is possible to re-iterate the prime sequence, but the security guarantees remain limited, and the performance costs increase rapidly, as shown in Section 5.3.1. We conclude that the current architecture does not provide the OS sufficient means to enforce time protection, and hardware support is needed.

5.2.3.3 Basic Flush (FLUSH_1): The channel matrix for the basic flush, presented in Section 4.2.1, is shown in Figure 6c. While the overall appearance of the channel matrix is flat, some patterns along the x-axis remain. Additionally, the mutual information is clearly above the zero-leakage upper bound, confirming a residual channel. A closer analysis reveals that the timing of memory accesses is not only determined by the state of the cache itself, but also by that of further stateful components, such as the LFSR providing a pseudo-random index sequence for the cache replacement policy, and the round-robin memory arbiters of the core. Concurrently to our work, Vila et al. [26] made similar observations on an Intel core.

5.2.3.4 Full Flush (FLUSH_2): The full flush (Section 4.2.2) clears these *secondary* components as well. While we close most channels with this approach, sporadically, a binary channel such as the one shown in Figure 6d reappears in the write-through L1 data cache. The channel does not exist consistently for each measurement. We observe that it appears depending on the initial hardware state.

Running CHANNEL BENCH on CVA6 in RTL simulation, we find that a single-cycle flush of the targeted components is insufficient. As CVA6 is a pipelined design, the flush signal may reach the various components at different points in time. If the components are not reset synchronously, information can flow from one component that is not yet reset to another component that has already been reset and thus persist. A possible approach to solving this issue is to apply the flush signal for multiple cycles to ensure it propagates through the whole design before being deasserted, as we do for MICRORESET. We do not explore this path further for the full flush.

Another channel that was identified through this analysis is the miss handler of the L1 data cache. When it receives a new request just before `fence.t` is executed, it waits for the cache's write-back and flush procedure to complete

	Write-Through L1										Write-Back L1									
	None		SW		FLUSH$_1$		FLUSH$_2$		MICRORESET		None		SW		FLUSH$_1$		FLUSH$_2$		MICRORESET	
	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$
L1D	**1629**	0.5	**1165**	0.5	**10.7**	1.6	**248.4**	10.0	21.9	27.8	**1620**	0.5	**770**	1	36.0	36.3	34.4	36.8	32.6	37.7
L1I	**1891**	0.5	n/a	n/a	**9.5**	1.4	33.4	42.0	42.0	42.5	**1893**	0.5	n/a	n/a	**9.1**	3.0	43.0	48.7	13.5	13.4
DTLB	**378**	0.1	n/a	n/a	1.7	4.4	4.8	5.7	4.3	7.9	**363**	0.1	n/a	n/a	69.0	90.0	37.6	91.4	60.9	91.6
BTB	**3611**	0.1	n/a	n/a	54.8	134.4	84.2	156.5	137.6	161.7	**3690**	0.1	n/a	n/a	85.2	158.2	92.3	181.3	83.1	162.7
BHT	**3933**	0.4	n/a	n/a	137.4	160.5	**161.0**	160.0	0.0	0.0	**4147**	0.2	n/a	n/a	118.8	167.2	99.8	162.2	0.0	0.0

Table 1: Timing channel capacities and their corresponding zero-leakage upper bounds for the unmitigated design and the discussed mitigation mechanisms in millibit [mb]. Leaking channels are highlighted.

before serving the request. The response is discarded later on, but it still leaves a trace on state that was already reset by `fence.t`. This trace depends on the request that was issued before `fence.t`, and therefore previous execution.

Although these channels might appear very small and impractical at first sight, they become more prominent as additional sources of noise are removed with the reset of other components. We see this as the main reason for sporadically higher channel capacities of FLUSH$_2$ compared to FLUSH$_1$ shown in Table 1. Also, it is important to highlight that our L1 data cache attack shown in Figure 6 does not target these channels specifically—they are only visible as a side effect of the L1 data cache attack. An attack directly targeting the presented channels would, presumably, achieve much higher capacity.

5.2.3.5 Microreset: Finally, MICRORESET from Section 4.2.3 yields the expected result, as demonstrated in Figure 6e. The channel is consistently closed across configuration and attacks, as supported by Table 1.

5.2.4 Further Components

Besides the L1 data cache, we analyse prime-and-probe attacks on the L1 instruction cache, the data TLB, the BTB, and the BHT, see Table 1. They confirm our findings from the L1 data cache: the unmodified design leaks significant amounts of data (e.g. more than 4 bit per iteration for the BHT), while executing `fence.t` using MICRORESET during a context switch reliably closes all channels.

5.2.5 Context-Switch Latency

To evaluate leakage through the context-switch latency, we configure the spy to measure the time span during which it is evicted, as shown in Figure 7. In this interval, the Trojan application runs for a fixed time-slice before the OS performs a context switch back to the spy.

Figure 8a shows the secret number of L1 data cache lines that the Trojan writes on the horizontal axis and the corresponding eviction duration measured by the spy on the vertical axis for the unmitigated case. A clear correlation between both is visible, indicating a covert channel. When the Trojan writes to cache lines, it evicts any kernel data stored at the same location. Thus, the context switch routine causes cache misses, increasing the context switch latency.

Resetting the microarchitectural state by executing `fence.t` on a context switch *without* accounting for the latency worsens the situation: Figure 8b shows a covert channel close to its theoretical upper capacity limit of 8 bit. As `fence.t` needs to write back the L1 data cache's dirty cache lines sequentially, its latency directly depends on the secret number of previously written cache lines.

We proposed padding for the worst-case execution latency in Section 4.3. Hence, we measure the latency for a fully dirty write-back cache: it takes $3\,077\,(\pm 7)$ cycles from the CLINT timer interrupt until the start of the execution of `fence.t` execution.[5] It then takes $18\,646\,(\pm 4)$ cycles for writing back and invalidating the dirty L1 data cache. Finally, $16\,(\pm 0)$ cycles are spent draining pending transactions, and MICRORESET is asserted for another $16\,(\pm 0)$ cycles, resulting in a total of $21\,755\,(\pm 8)$ cycles from the CLINT timer interrupt until the end of `fence.t`. We conservatively round up and set `cspad` $= 22\,000$. Similarly, we determine a worst-case upper bound of 3 700 cycles for the write-through L1 data cache. As shown in Figure 8c, this removes any dependence of the context switch latency on previous execution and microarchitectural state.

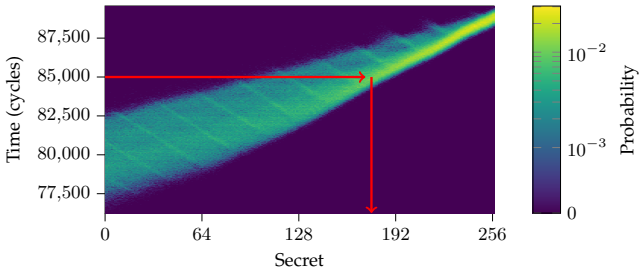
5.3 Costs

5.3.1 Context-Switch Latency

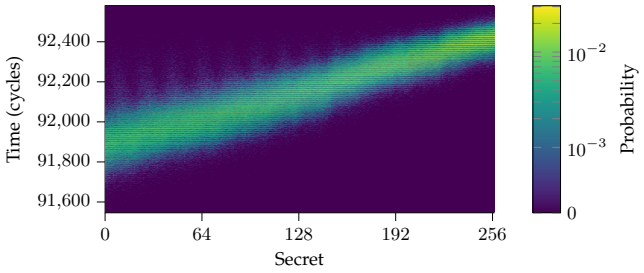
For evaluating the context switch latency, we use the inter-address-space IPC benchmark from sel4bench [27]. Analogue to Section 4.3, we set `cspad` $= 22\,000$. However, as the benchmark uses the process-initiated *fastpath* context switch routine, in contrast to Section 4.3, the beginning of the context switch routine is not defined by a CLINT timer interrupt. Hence, for our performance evaluation, we use the privilege level switch from U-mode as the start of the pad interval. This event approximates the CLINT timer interrupt for full context switches.

Table 2 shows the context-switch latencies for different configurations. *Hot* assumes that the core very recently executed a context switch; hence the caches contain kernel data, the branch predictors are trained, etc. No on-core mitigations against timing channels are in place. On the other hand, *Cold/Dirty* is the context-switch latency for an untrained and dirty microarchitecture. For instance, this is the case after a resource-intensive application has evicted all kernel information during its time slice. We assume that this is the more common case in practice. We emulate this behaviour by resetting the microarchitecture and, if configured as write-back, polluting the L1 data cache from user space between context switches. Finally, *SW* and `fence.t` are the resulting context-switch latencies after configuring seL4

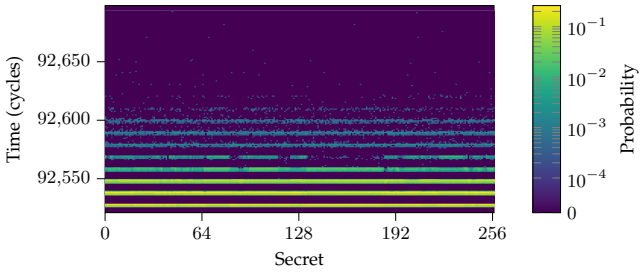
5. The majority of this latency (approximately 1800 cycles) are spent for reconfiguring the CLINT. Scheduling takes around 800 cycles and switching to the new thread (e.g. changing the address space) takes another 320 cycles.



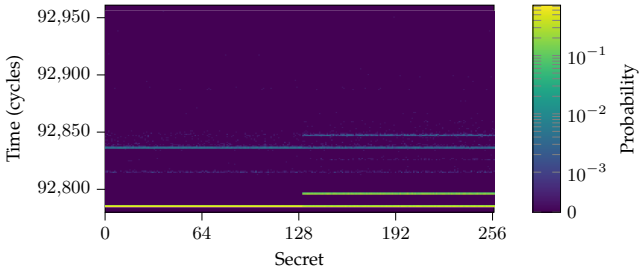
a: Unmitigated. (1629/0.5)



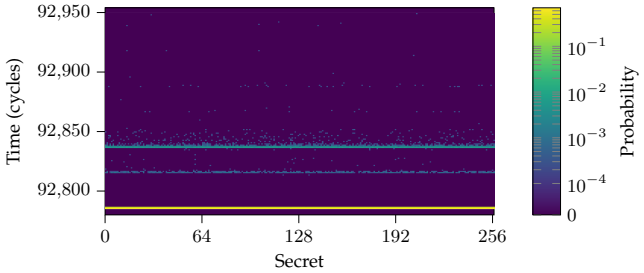
b: Software. (1165/0.5)



c: FLUSH$_1$. (10.7/1.6)



d: FLUSH$_2$. (248.4/0.1)



e: MICRORESET. (21.9/27.8)

Figure 6: Channel matrices and corresponding mutual information ($\mathcal{M}$[mb]/$\mathcal{M}_0$[mb]) for the write-through L1D.

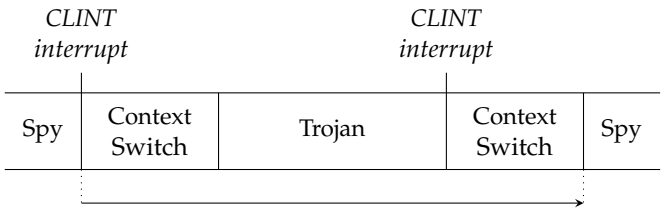


Figure 7: Time span measured by the spy in the context-switch latency channel.

Mitigation		Write-Through L1	Write-Back L1
		Mean ± SD	Mean ± SD
None	Hot	514 ± 0	423 ± 0
	Cold/Dirty	1243 ± 2	1827 ± 217
SW		40 650 ± 2	41 152 ± 15
`fence.t`	FLUSH$_1$	4073 ± 1	22 402 ± 5
	FLUSH$_2$	4067 ± 1	22 402 ± 5
	MICRORESET	4125 ± 0	22 450 ± 0

Table 2: seL4 fastpath context switch latency in cycles without mitigation, mitigated using existing architecture (SW), and with `fence.t`. `fence.t` is padded for a worst-case slowpath context switch.

to mitigate on-core timing channels using the approaches described in Section 4.

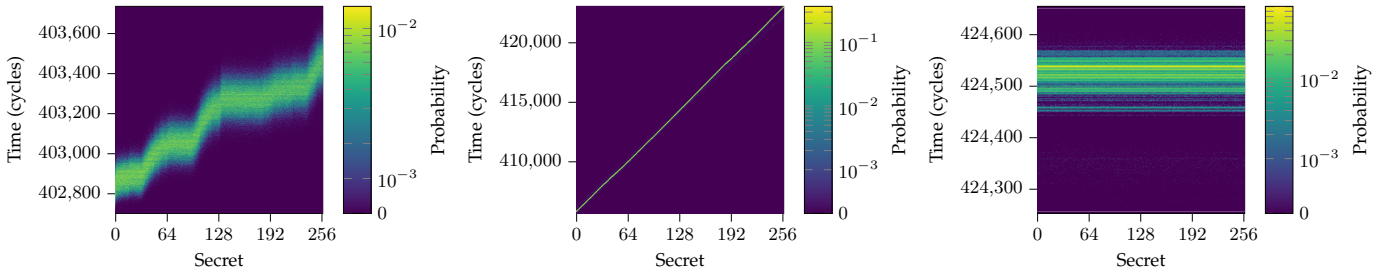
The Software mitigation (*SW*) is significantly more expensive than the `fence.t` approaches while only partially mitigating the L1 data cache channel. A more extensive mitigation would make this approach even more costly.

Comparing MICRORESET to both flush approaches, there are no significant performance differences. The reason is that most components of CVA6 are reset in parallel—therefore, flushing or resetting more state usually does not require more cycles. The dominating factor for both approaches is the write-back of the L1 data cache padded for the worst case. This means that a principled reset generally does not imply higher costs than a selective flush.

Compared to the cold, unmitigated case, `fence.t` adds less than 21 000 cycles to the context-switch routine. Assuming a processor running at 1 GHz and a context-switch frequency of 100 Hz, this increase corresponds to an overhead of about 0.2 %. Decreasing the frequency of `fence.t` (e.g. in a hypervisor scenario or by clustering mutually trusting applications into security domains) would decrease the relative overhead accordingly. It is important to note that `fence.t` is padded for the worst-case slowpath context switch, while *Cold/Dirty* gives the fastpath latency. When applied to a slowpath context switch, the overhead would be smaller. In addition to the direct costs shown here, the dirty cache would cause indirect costs resulting from cache misses and write-backs experienced after the context switch, while after `fence.t`, execution continues with a clean cache. As such, Table 2 overestimates the performance impact of `fence.t`.

5.3.2 Indirect Costs

Resetting the microarchitectural state on a context switch potentially removes an application's information from the on-core state that is still required at a later point, resulting



$M = 1297.9\,\text{mbit}$
$M_0 = 0.5\,\text{mbit}$

a: Unmitigated.

$M = 7257.2\,\text{mbit}$
$M_0 = 0.4\,\text{mbit}$

b: `fence.t` *without* time padding.

$M = 1.4\,\text{mbit}$
$M_0 = 1.6\,\text{mbit}$

c: `fence.t` *with* time padding.

Figure 8: Evicted time measured by spy over secret number of L1 data cache lines written by Trojan.

in indirect costs due to cache misses or mis-speculation once the application is re-scheduled. However, Ge et al. have shown that application-specific information is mostly evicted from the on-core state (L1 caches, branch predictors etc.) after one or several time-slices of execution of other application(s), and that the indirect costs of a reset on context switch are therefore limited.

A difference in indirect costs between MICRORESET and the selective flush can generally only be caused either by a poor choice of reset values for one of both approaches, or by a residual timing channel.

5.3.3 Benchmark results

We evaluate the overhead introduced by `fence.t` with MICRORESET using the Splash-2 benchmarks [28]. We set up two domains: the first executing the benchmark and the second concurrently running an idle thread. We perform a context switch between both domains every 10 million cycles, which corresponds to a typical 10 ms timeslice period on a processor running at 1 GHz.

Figure 9 shows the slowdown of the benchmark when executing a `fence.t` on every context switch. The standard error is below 0.03 % for all results. The average slowdown is 0.7 %, which confirms the low performance overhead of `fence.t`.

We note that this is a rather pessimistic evaluation: the idle thread has no significant memory footprint, meaning that for the baseline, the benchmarks benefit from a hot microarchitecture. In a setting with high memory contention, the latency of `fence.t` (dominated by the cache flush) may be almost entirely hidden.

5.3.4 Hardware Overhead

We synthesise the original and modified versions of CVA6 in GLOBALFOUNDRIES 22 FDX technology at 1 GHz at worst-case conditions (0.72 V, 125 °C). We convert the results to gate equivalent (GE), a technology-independent unit for the complexity of a circuit. The area overhead of our modifications is negligible at 0.4%, with the `fence.t` controller being the largest addition at around 1.6 kGE compared to a total core area of 1.2 MGE. There is no significant impact on the critical path.

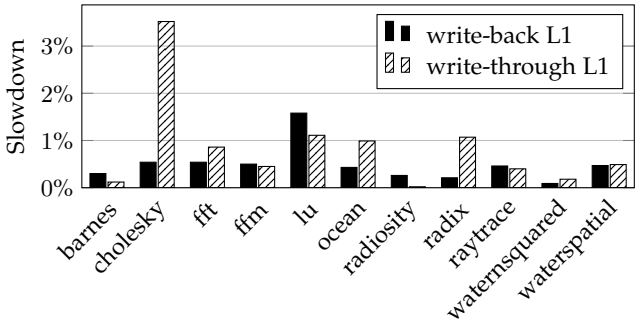


Figure 9: Splash-2 benchmark slowdown by `fence.t`

6 RELATED WORK

The L1 data cache is the focus of several previous works on on-core timing channel mitigation. One possible approach is spatial partitioning of the cache, proposed by Page [29] and followed up on by Domnitser et al. [30] and Dessouky et al. [31]. Since L1 caches are relatively small and time-shared between applications, spatial partitioning of these components is not very efficient. Wang and Lee [32], [33] propose a dynamic, randomised cache-remapping to mitigate targeted cache collisions. An improved implementation was presented by Qureshi [34]. We find this approach insufficient, as in general, randomisation merely adds noise to a communication channel without fundamentally closing it. As Constable and Unterluggauer [35] demonstrate, cache line mappings be designed to prevent specific attacks (i.e. prime-and-probe on the index of an accessed cache line). Besides imposing impractical constraints on the cache layout, this approach is not suited for other attacks, such as prime-and-probe on the *number* of accessed cache lines, which we use in this work.

All works mentioned above do not consider microarchitectural components besides the L1 data cache. Tiwari et al. [36] propose a major architectural modification to fundamentally separate data from control flow. Our work extends that of Ge et al., who propose time protection and the need for flushing all microarchitectural on-core state on a partition switch, and demonstrate the need for

hardware support [1], [7], [23], which is what our temporal fence provides. There exist several approaches in a similar direction: Bourgeat et al. [37] present a processor with a `purge` instruction, similar to our `fence.t`, that flushes on-core microarchitectural components to secure enclaves. Li et al. [38] propose FENCEX, an instruction similar to the basic flush version of `fence.t` presented in Section 4.2.1 of this work, which we found insufficient to close all timing channels reliably. Escouteloup et al. [39] present an ISA extension that allows the allocation of hardware resources to security domains. They implement and evaluate their proposal bare-metal on an embedded RISC-V core designed to model known microarchitectural vulnerabilities, whereas this work targets an existing, application-class RISC-V core, optimised for efficiency and running a full operating system.

To the best of our knowledge, all previous works on temporal partitioning first identify vulnerable microarchitectural components, and then add them to a partition set. A major drawback of this approach is that it remains difficult to make hard claims, such as that *all* microarchitectural covert channels are closed. Our MICRORESET proposed in Section 4.2.3 works the other way around: all stateful components are reset per default. Only a selected set of architectural state is explicitly excluded from the reset. Furthermore, we also consider the flush latency itself, since neglecting it can open significant new channels, as demonstrated in Section 5.2.5.

7 CONCLUSIONS

In this work, we present the temporal fence instruction, `fence.t`, which allows an OS to reliably prevent on-core timing channels. We propose and compare different hardware implementations of `fence.t`, ranging from an basic flush of well-known vulnerable microarchitectural components, over an exhaustive flush of manually identified vulnerable secondary components, to a systematic erasure of all non-architectural state, which we call MICRORESET. Evaluating these mechanisms on the open-source RV64GC CVA6 core with different cache configurations and running an experimental, unverified version of the seL4 microkernel, we find that `fence.t` with MICRORESET is the only approach that consistently closes all timing channels, while offering a low implementation effort, a performance impact of less than 1 % for a typical 1 GHz system with a 10 ms context switch period, and negligible hardware costs.

ACKNOWLEDGMENTS

The authors would like to thank Qian Ge and Curtis Millar for their support with the covert channel measurement framework, Florian Zaruba for his help and insights on CVA6, and Paul Scheffler for his feedback on the manuscript.

The work of Wistoff, Gürkaynak, and Benini was supported in part by the European Union's Horizon 2020 research and innovation programme FRACTAL project funded by ECSEL-JU grant agreement #877056, ETH4HC, and the ETH4D Humanitarian Action Challenges Application on "Secure Infrastructure for Humanitarian Organizations".

REFERENCES

[1] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing OS abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, pp. 1:1–1:17.

[2] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[3] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19.

[5] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.

[6] X. Ren, L. Moody, M. Taram, M. C. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 361–374, 2021.

[7] Q. Ge, Y. Yarom, and G. Heiser, "No security without time protection: We need a new hardware-software contract," in *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*. ACM, 2018, pp. 1:1–1:9.

[8] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM (CACM)*, vol. 16, pp. 613–615, 1973.

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

[10] W.-M. Hu, "Lattice scheduling and covert channels," in *1992 IEEE Symposium on Security and Privacy (SP)*, 1992, pp. 52–61.

[11] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[12] O. Aciiçmez, "Yet another microarchitectural attack: Exploiting i-cache," in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, ser. CSAW '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 11–18. [Online]. Available: https://doi.org/10.1145/1314466.1314469

[13] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[14] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[15] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, p. 338–359, Nov. 1992.

[16] Intel, "Speculative execution side channel mitigations," 2018. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf

[17] O. Group, "CVA6," 2017. [Online]. Available: https://github.com/openhwgroup/cva6

[18] W. Rönninger, "Memory subsystem for the first fully open-source RISC-V heterogeneous SoC," Master's thesis, ETH Zürich, Nov. 2019.

[19] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[20] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[21] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.
[22] T. Chothia, Y. Kawamoto, and C. Novakovic, "A Tool for Estimating Information Leakage," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Springer Berlin Heidelberg, 2013, pp. 690–695.
[23] Q. Ge, "Principled elimination of microarchitectural timing channels through operating-system enforced time protection," Ph.D. dissertation, University of New South Wales, Oct. 2019.
[24] ——, "Timing channel benchmarking tool," 2015, https://github.com/SEL4PROJ/channel-bench.
[25] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," 2016. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf
[26] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke, "Flushgeist: Cache leaks from beyond the flush," arXiv:2005.13853 [cs.CR], 2020.
[27] A. Lyons, "seL4 benchmarking applications and support library," 2014. [Online]. Available: https://github.com/seL4/sel4bench
[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.
[29] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.
[30] L. Domnitser, A. Jaleel, J. Loew, N. B. Abu-Ghazaleh, and D. V. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization*, vol. 8, pp. 35:1–35:21, 2012.
[31] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," *ArXiv*, vol. abs/2110.08139, 2021.
[32] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2007, pp. 494–505.
[33] ——, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 83–93.
[34] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, 2018.
[35] S. D. Constable and T. Unterluggauer, "Seeds of seed: A side-channel resilient cache skewed by a linear function over a galois field," *ArXiv*, vol. abs/2109.14652, 2021.
[36] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 493–504, 2009.
[37] T. Bourgeat, I. A. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, p. 42–56, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358310
[38] T. Li, B. D. Hopkins, and S. Parameswaran, "Simf: Single-instruction multiple-flush mechanism for processor temporal isolation," *ArXiv*, vol. abs/2011.10249, 2020.
[39] M. Escouteloup, R. Lashermes, J. Fournier, and J.-L. Lanet, "Under the dome: preventing hardware timing information leakage," in *20th Smart Card Research and Advanced Application Conference (CARDIS)*, Lübeck, Germany, Nov. 2021, pp. 1–20. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03351957

Nils Wistoff received his B.Sc. and M.Sc. from RWTH Aachen University in 2017 and 2020, respectively. He is currently pursuing a Ph.D. at the Integrated Systems Laboratory of ETH Zürich. Wistoff's research interests include processor and system-on-chip design and secure computer architecture.

Moritz Schneider has received his B.Sc. and M.Sc. in electrical engineering from ETH Zurich and he is currently pursuing a Ph.D. in the System Security group at ETH Zurich. His research interests include hardware security, trusted execution, and system security.

Frank K. Gürkaynak has obtained his B.Sc. and M.Sc. in electrical engineering from the Istanbul Technical University, and his Ph.D. in electrical engineering from ETH Zürich in 2006. He is currently working as a senior scientist at the Integrated Systems Laboratory of ETH Zürich. His research interests include digital low-power design and cryptographic hardware.

Gernot Heiser is Scientia Professor and John Lions Chair at UNSW Sydney. His research interests are provably secure and dependable operating systems, and their application in security- and safety-critical uses cases. He also serves as the chairman of the seL4 Foundation. Dr Heiser is a Fellow of the ACM and the Australian Academy of Technology and Engineering (ATSE) and a winner of the ACM SIGOPS Hall of Fame award.

Luca Benini holds the chair of digital Circuits and systems at ETHZ and is Full Professor at the Universita di Bologna. Dr. Benini's research interests are in energy-efficient computing systems design, from embedded to high-performance. He has published more than 1000 peer-reviewed papers and five books. He is a Fellow of the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award.