Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core

Nils Wistoff
ETH Zurich and RWTH Aachen
and HENSOLDT Cyber GmbH
Zurich, Switzerland
nwistoff@ethz.ch

Moritz Schneider
ETH Zurich
Zurich, Switzerland
moritz.schneider@inf.ethz.ch

Frank K. Gürkaynak
ETH Zurich
Zurich, Switzerland
kgf@iis.ee.ethz.ch

Luca Benini
ETH Zurich
Zurich, Switzerland
lbenini@iis.ee.ethz.ch

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

ABSTRACT

Covert channels enable information leakage across security boundaries of the operating system. Microarchitectural covert channels exploit changes in execution timing resulting from competing access to limited hardware resources. We use the recent experimental support for time protection, aimed at preventing covert channels, in the seL4 microkernel and evaluate the efficacy of the mechanisms against five known channels on Ariane, an open-source 64-bit application-class RISC-V core. We confirm that without hardware support, these defences are expensive and incomplete. We show that the addition of a single-instruction extension to the RISC-V ISA, that flushes microarchitectural state, can enable the OS to close all five evaluated covert channels with low increase in context switch costs and negligible hardware overhead. We conclude that such a mechanism is essential for security.

CCS CONCEPTS

• **Security and privacy** → **Hardware security implementation**; **Operating systems security**; • **Computer systems organization** → Reduced instruction set computing.

KEYWORDS

covert channels, timing channels, computer architecture, microarchitecture, operating systems, system security, time protection

1 INTRODUCTION

A covert channel is an information flow that uses a mechanism not intended for information transfer [21], and thereby violates a system's security policy that the OS is meant to enforce. For example, some untrusted code, such as a mail client, may be given access to secrets but should be *confined* to only communicate with the outside world via an encrypted channel. A covert channel can enable the mailer to leak the raw secrets, bypassing encryption.

Covert channels that utilise OS-managed spatial resources (storage channels) can be eliminated completely, as was proved for the seL4 microkernel [24]. Harder to control are channels that target physical quantities not directly managed by the OS, such as processor temperature [23] or power draw [18]. Particularly dangerous are *timing channels*, which exploit information encoded in the timing of events, as they can be exploited remotely.

Of particular importance are microarchitectural timing channels; these exploit competition for limited hardware resources that are hidden by the instruction set architecture (ISA) [9]. For example, the Spectre attack [20] uses speculation to construct a Trojan from "gadgets" in innocent code, with the Trojan leaking arbitrary information through a microarchitectural timing channel. Exploitable resources are those holding state that depends on execution history, which includes caches, TLBs, branch predictors, and prefetchers.

Time protection, a set of OS mechanisms complementing the established memory protection, aims to prevent timing channels [8]. However, its proponents also demonstrated that on contemporary hardware full time protection is unachievable, as some exploitable microarchitectural states cannot be reset by software. They consequently argue that the hardware-software contract must be amended to provide the OS with the mechanisms for resetting exploitable microarchitectural state [10].

In this work, we investigate such mechanisms by implementing them in Ariane, an open-source, application-class, in-order, RISC-V RV64 core. We evaluate their efficacy on five known microarchitectural channels, and the overheads imposed by their use. Specifically, we make the following contributions:

(1) We measure the capacites of five established microarchitectural covert channels on the unmodified Ariane core, and confirm that they are comparable to those found in high-performance Intel and Arm processors.
(2) We confirm previous observations on Intel and Arm cores that software-only approaches are expensive and ineffective.
(3) We demonstrate the importance of resetting *all* microarchitectural state by showing that after resetting first-order state (valid bits), secondary state (e.g. state bits in the cache replacement policies) can still be exploited.
(4) We propose a new RISC-V `fence` instruction, whose argument lets the OS control which state is flushed, and demonstrate that it completely eliminates the studied channels.

2 BACKGROUND

2.1 Threat Model

We examine covert-channel leakage under a *confinement* scenario [21]: An untrusted program possesses a secret, and the OS encapsulates the program's execution in a security domain that only

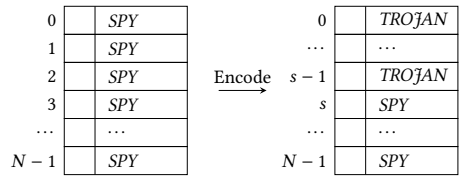


Figure 1: A cache before and after the Trojan encodes the secret s.

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. A second, unconfined, and also untrusted security domain contains a spy which is trying to read the secret leaked by the Trojan.

Intentional leakage by a Trojan represents the worst case leakage; if we can prevent it, we also preclude any other leakage through the same channel. In particular, this rules out *side channels*, where instead of a Trojan, the leakage originates from an unwitting victim.

We assume that the Trojan and spy time-share a core, meaning cross-core leakage is out of scope. We only consider microarchitectural timing channels. Covert channels that abuse other characteristics, such as power draw, are out of scope.

2.2 Exploitable Microarchitectural State

Exploits of data and instruction caches have been known for decades [15]. The cache lines used by the Trojan create a footprint that can be sensed by the spy: It observes the latencies of its own memory accesses, which are high where a cache line has been replaced by the Trojan (see Section 2.3 for details). TLBs are caches for translation data and can be similarly exploited [16].

The branch predictor also contains caches that can be exploited [1]: the branch target buffer (BTB), which caches the destination addresses of indirect jumps, and the branch history table (BHT), which predicts whether conditional branches are taken.

Instruction and data prefetchers contain state machines which accumulate history and can be exploited [8]. However, simple processors such as our Ariane core do not feature prefetching; we therefore do not investigate this channel.

2.3 Exploiting Covert Channels

Techniques for exploiting covert channels are well established; for our scenario of intentional leakage, the *prime-and-probe* (P&P) attack [26] is simple and effective.

In a P&P attack, the spy first forces the exploited hardware resource into a known state (*prime*). For the D-cache this means traversing a large buffer (in cache-line-sized strides for efficiency), for the I-cache by executing a series of linked jumps. The TLBs are similarly primed by accessing or jumping with page-size strides. (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.) With a correctly-sized priming buffer, this leaves the hardware resource in a state where further accesses by the spy within the same address range are fast, as illustrated on the left of Figure 1.

At the end of its time slice, the OS preempts the spy and switches to the Trojan, which accesses a subset of the hardware resource to encode the secret. Given a D-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 on the right of Figure 1. 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.

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 would have to measure the time of each individual access and perform a more complex analysis.

2.4 Time Protection

Time protection is a recently proposed, principled approach to *eliminating* timing channels [8]. 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-architected 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* [17], 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 impossible for on-core resources for lack of hardware support. These are usually also fairly small and highly utilised by a single program, so partitioning would result in unacceptable performance degradation. Furthermore, on-core resources are accessed by virtual address, which is not under OS control, making approaches such as colouring infeasible.

This leaves temporal partitioning for on-core resources. In order to prevent any interference between security domains, each such resource must be reset to a state that is independent of execution history before handing it to a different domain. This means that the OS must be provided with the means to perform such a reset of all microarchitectural state, creating the requirement of extending the hardware-software contract to refer (in a highly abstract way) to such non-architected state [10]. The authors specifically show that contemporary Intel and Arm processors lack the mechanisms required for implementing time protection.

2.5 Proposed Temporal Fence

We introduce such a mechanism in the form of a *temporal fence* instruction, `fence.t`, which isolates the timing of any subsequent execution from what happened before.[1] Our fence instruction specifically applies to on-core state only, as off-core state can be spatially partitioned. We realise that this definition is less abstract than one

[1] Krste Asanović introduced the notion of a temporal fence on the RISC-V mailing list.

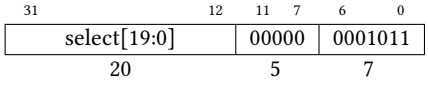


Figure 2: Encoding of the `fence.t` instruction.

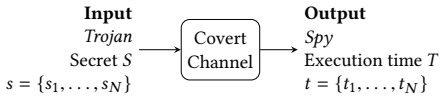


Figure 3: Relationship of the measured parameters.

might wish, and is therefore unlikely to be the last word on the topic. However, it suits our present purpose of evaluating the desired functionality.

We parameterise the `fence.t` instruction by the microarchitectural state targeted, as suggested by Ge et al.. This helps the OS to minimise flushing according to its security requirements. For this study, it has the additional benefit that we can target individual channels for a fine-grained examination of efficacy.

We encode the `fence.t` instruction as a custom U-type instruction with the RISC-V opcode *custom-0* (Figure 2). A bitmap passed as the 20-bit immediate value selects the components to be flushed.

3 METHODOLOGY

We adopt the approach of Ge et al. [10] for quantifying and evaluation leakage and prevention strategies.

3.1 Measuring Leakage

We run each attack for a number of iterations, the *sample size*, usually 1 million. In each iteration, 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, see Figure 3. 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 combination of two indicators: The *channel matrix* as a visual representation of leakage, and the *discrete mutual information* $\mathcal{M}$ as a quantitative metric.

3.1.1 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 (see Figure 6 for examples). A variation of colour along any horizontal line through the graph indicates a dependence of the output on the input, and thus a channel.

3.1.2 Mutual Information. For quantifying channel capacity we use *mutual information* $\mathcal{M}$, the amount of information gained about a random variable by observing another, possibly correlated random variable [28]. It can be expressed as the difference between the marginal entropy $\mathrm{H}(T)$ and the conditional entropy $\mathrm{H}(T \mid S)$:

$$\mathcal{M} = \mathrm{I}(S;T) = \mathrm{H}(T) - \mathrm{H}(T \mid S)$$

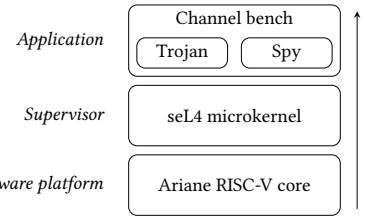


Figure 4: Evaluation platform.

$\mathcal{M}$ is measured in bits; as most of our channel capacities are small, we use millibits (1 mb = 10^{-3} b). 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 the mutual information becomes high. Conversely, if both random variables are uncorrelated, we have $\mathrm{H}(T) = \mathrm{H}(T \mid S)$ and therefore $\mathcal{M} = 0$.

Zero Leakage Upper Bound $\mathcal{M}_0$. Since all measurements are affected by noise, $\mathcal{M}$ will never 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 rearrange the input/output pairs into uniformly random pairs and thus remove any correlation between them, while retaining their original value ranges 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. We conclude that a channel is definitely present if $\mathcal{M} > \mathcal{M}_0$, else that the result is consistent with no channel.

We use the leakiEst tool [3] to compute mutual information $\mathcal{M}$ and zero leakage upper bounds $\mathcal{M}_0$.

3.2 Evaluation Platform

3.2.1 Ariane. The hardware platform for evaluating channels and defences is based on *Ariane*, an open-source, RV64GC, 6-stage RISC-V core developed at ETH Zurich [33]. It is implemented in SystemVerilog and publicly available on GitHub [11]. It features three privilege levels and virtual memory (SV39) from the privileged ISA specification [31], and thus supports full-fledged operating systems. Its configurability, simplicity, and openness make it a good candidate for architectural exploration.

Setup. We instantiate the Ariane core on an FPGA (Digilent Genesys II), running at 50 MHz, using the standard configuration with an 8-way, 32 KiB write-through L1-D and a 4-way, 16 KiB L1-I cache. Both use 16-byte lines and a pseudo-random replacement strategy driven by an 8-bit linear-feedback shift register (LFSR). The L1-D is accessed by the load-, store-, and memory-management units, with concurrent accesses arbitrated round-robin. The branch predictor has a 64-entry BHT and a 16-entry BTB. There is a single-level, unified, fully associative, 16-entry TLB using a pseudo-LRU replacement policy. For reducing write-stalls we increase the write buffer to 40 entries. We add some off-core components, including a timer and a 512 KiB write-back L2 cache [27] that is connected to DRAM. Figure 5 shows the memory architecture.

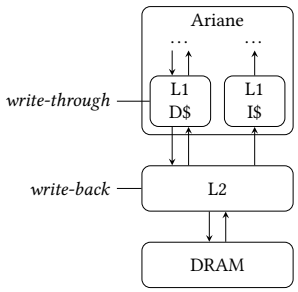


Figure 5: Memory system of the Ariane SoC.

We partition the L2 cache by colouring [17], which precludes channels in the memory backend and allows us to focus on channels resulting from on-core state.

3.2.2 OS. Ge's *channel bench* [7, 12] provides a minimal OS and data collection infrastructure; we port it to RISC-V and adapt to Ariane. Channel bench uses attack implementations from the *Mastik* toolkit [32], running on an experimental version of seL4 [19] that supports time protection. seL4 is an open-source, high-performance OS microkernel with formal proofs of implementation correctness and security enforcement, making it highly suitable for security evaluations, although our experimental version is not verified.

4 COVERT-CHANNEL CAPACITIES

4.1 Baseline: No Time Protection

To establish a baseline and compare to other architectures, we apply the P&P attacks to our Ariane RV64GC core, with time protection disabled in seL4. We observe strong channels through each of the five microarchitectural resources targeted. As shown in the *Unmitigated* column of Table 1, capacities range from 0.4 to 4 bits. The $\mathcal{M}_0$ are all well below 1 mb, indicating that the channels are real. To put those numbers into context: Assuming the OS uses a 1 ms time slice, Trojan and spy will each execute 500 times per second. The 1.6 b capacity of the D-cache thus means the channel has a bandwidth of 833 b/s, able to leak a 1024-bit RSA key in just over a second. Also, these channels use a rather primitive encoding scheme; more sophistication could increase the bandwidth significantly.

The channel capacities we observe agree nicely with the prior work, which showed unmitigated capacities of 0.3–4 b on Intel and 7.5 mb to 2.5 b on Arm processors [8].

Figure 6a and Figure 7a show the unmitigated channel matrices for the L1-D cache and the BHT, respectively; N is the number of iterations. The clear diagonal pattern indicates a strong correlation of output with input signals, establishing efficient channels. For example, Figure 7a shows that if the spy observes a probe time of 380 cycles, it can infer with a high confidence that the Trojan has encoded the value *48*.

4.2 Using Existing Instructions Only

Ge et al. report that neither the x86 nor the Arm architectures provide sufficient mechanisms for implementing time protection, with Arm coming closer in at least providing targeted L1 cache flushes. The Intel architecture does not have those, and the authors implemented software flushing by touching all cache lines, similar to the prime phase of the P&P 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.

Table 1: Mutual information and corresponding zero leakage upper bound in millibits.

	Unmitigated		First `fence.t`		Final `fence.t`	
	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$
L1 D$	1667.3	0.5	10.4	6.0	33.3	39.1
L1 I$	1905.0	0.5	8.3	4.9	37.9	39.4
TLB	408.7	0.1	5.0	5.9	3.1	7.7
BTB	3211.4	0.1	35.7	59.3	28.2	60.3
BHT	3770.6	0.2	45.2	58.8	44.1	60.8

With RISC-V, the situation is presently worse, as specification of cache management is still under discussion. While implementations generally support some cache management, this is consequently not standardised. To explore this aspect, we implement a "software only" defence, 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.

Figure 6b shows the result for the L1-D cache channel, where the OS performs *two* priming runs per context switch. While fuzzier than in the unmitigated case, a clear diagonal pattern persists, and the measured capacity is only reduced by 70%, making this defence highly ineffective, a result that is much worse than Ge et al. observed for Intel. The reasons are for one the Ariane's replacement policy, which uses a pseudo-random sequence with a period of 256. This makes it practically impossible to flush the cache reliably through priming. Furthermore, there is secondary state that is even harder to reset, as we will find in Section 4.3.2.

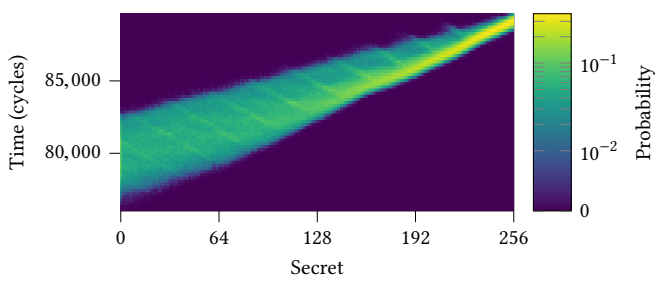
4.3 Using a Temporal Fence

As discussed in Section 2.5 we add a new `fence.t` instruction to the Ariane. When `fence.t` is committed, Ariane's controller sends a flush signal to all stateful microarchitectural components. To give the operating system maximum control, an immediate value selects the components to be flushed.

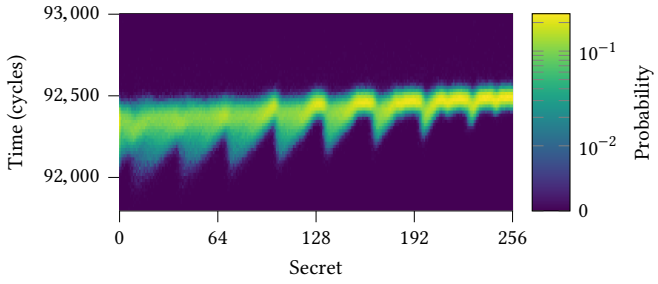
4.3.1 First Attempt. As we want to minimise the cost of the fence, we only flush state that seems to be relevant for the P&P attacks. In particular, we reset the L1 cache and TLB state by clearing the valid bits (remember, our Ariane's L1-D is a write-through cache so there is no dirty state to write back). Similarly, we reset the saturation counter of the BHT. To avoid interfering with in-flight computations, we also flush the pipeline.

We find that this is insufficient, as shown in Figure 6c. While the channel pattern is gone, the channel is not quite closed: The channel matrix shows slight patters along horizontal lines. The mutual information, shown in Table 1 as "First `fence.t`", is almost twice the zero-leakage bound, confirming the channel.

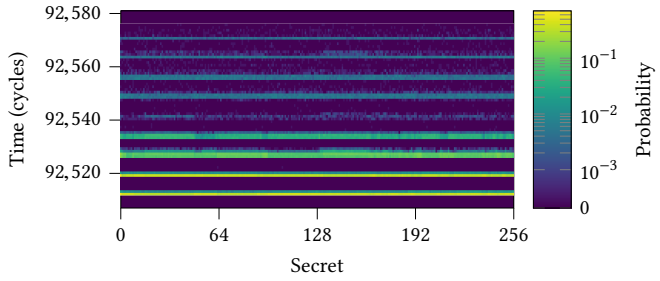
4.3.2 Improved Fence. Investigating the source, we identify further state that indirectly affects timing. Most prominently,



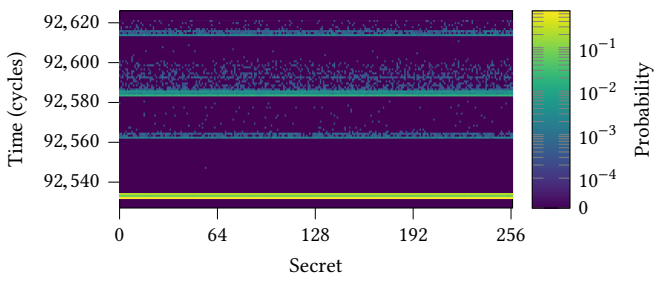
(a) Unmitigated. $N = 10^6$, $\mathcal{M} = 1667.3$mb, $\mathcal{M}_0 = 0.5$mb.



(b) Attempted reset by priming twice. $N = 10^6$, $\mathcal{M} = 515.7$mb, $\mathcal{M}_0 = 1.1$mb.

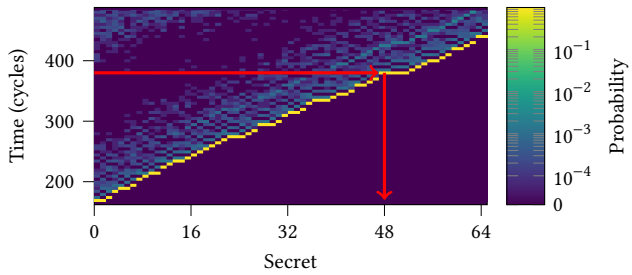


(c) Original fence clearing first-order state. $N = 10^6$, $\mathcal{M} = 10.4$mb, $\mathcal{M}_0 = 6.0$mb.

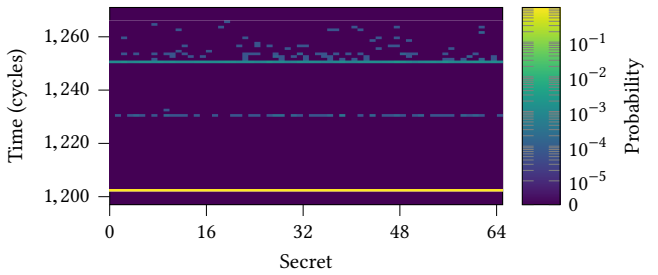


(d) Improved fence clearing first- and second-order state. $N = 10^6$, $\mathcal{M} = 33.3$mb, $\mathcal{M}_0 = 39.1$mb.

Figure 6: L1 data cache channel matrices.



(a) Unmitigated. $N = 10^6$, $\mathcal{M} = 3770.6$mb, $\mathcal{M}_0 = 0.2$mb.



(b) Improved fence. $N = 10^6$, $\mathcal{M} = 44.1$mb, $\mathcal{M}_0 = 60.8$mb.

Figure 7: BHT channel matrices.

- the LFSR used for victim selection in L1 cache replacement
- the round robin arbiter of the L1 data cache
- the pseudo-LRU tree for the TLB replacement strategy.

We include these components in the flush and re-run the experiments, the results are shown as "Final `fence.t`" in Table 1. All measured channel capacities are now clearly below the zero-capacity threshold, meaning that there is no evidence of a residual channel. The channel matrices confirm this: Figure 6d and Figure 7b only show patterns that appear to be random noise (confirmed by visual comparison between multiple runs).

5 COST

5.1 Context-Switch Latency

Time protection resets hardware state on a switch of security partition, which implies a full context switch. seL4's IPC is essentially a user-triggered context switch [13] with roughly the same cost as a time-slice preemption, and the seL4 benchmark suite [22] provides a convenient rig for measuring its latency. We use inter-address-space IPC for evaluating flush cost. Table 2 compares the latencies of various configurations. Here "hot" measures the best-case of switching for and back in a tight loop, where the whole working set fits into the L1 caches. The cold-cache scenario is the realistic baseline for our purposes, as a security-domain switch is normally triggered by time-slice preemption; as time slices are 1 ms or longer, the newly executing domain is unlikely to have any hot data left in the small L1 caches. We achieve the cold state by executing `fence.t` from user mode (before the timed context-switch).

The third column shows the latency with the OS trying to reset state by double priming as discussed in Section 4.2, note that this only attempts to mitigate the D-cache channel. Finally, "`fence.t`" uses the full flush provided by the temporal fence.

We already found in Section 4.2 that the software priming is highly ineffective, the results here show that it is also very expensive, increasing context-switch latency by a factor of 50 over the cold-cache case (while not even attempting to mitigate channels

Table 2: seL4 IPC latencies and standard deviations in cycles.

Unmitigated		Mitigated	
Hot	Cold	D-cache prime	`fence.t`
430 (±7.0)	1,180 (±1.0)	51,877 (±256)	1,502 (±0.9)

other than the L1-D). In contrast, the temporal fence, which we have found to be highly effective against *all* channels, only adds 320 cycles (less than 30%) to the cold-cache latency. With a switch rate of no more than 1 kHz, this adds negligible cost.

The dominating contribution to the direct latency of the `fence.t` instruction is the cache flush. A write-through cache is flushed by clearing all valid bits. This is a constant-time operation, which could in theory be performed in a single cycle. However, in Ariane's write-through cache, the valid bits are stored together with the tags in sequentially accessible SRAM, allowing for an invalidation of only one set per cycle, and thus resulting in a latency of 256 cycles. All other state can be reset in a single cycle.

A write-back L1-D cache would be more expensive to flush, as each dirty line must we written back to the L2. Since the L2 of our platform can process up to 8 B per cycle, the theoretical latency for a write-back variant varies between 0 cycles (clean cache) and 4,096 cycles (all lines dirty). In such a case of a variable latency, the OS must pad to the worst-case latency, to prevent the flush latency becoming a covert channel [10].

5.2 Hardware Overhead

To estimate the hardware costs incurred by the `fence.t` instruction, we examine the resource utilization of our FPGA. The number of deployed LUTs remains within 1% of the original size. Hence, the mechanism should not cause a notable increase in chip area or power draw of the design.

6 RELATED WORK

Past work has approached the hardware mitigation of microarchitectural covert channels from different angles. Page [25] propose static partitioning of the L1 cache while Wang and Lee [29] propose locking cache lines. While spatial partitioning can certainly prevent attacks, in the case of the L1, the reduction of available cache space would have a major impact on application performance. Wang and Lee [30] instead aim to defeat attacks by dynamically remapping cache lines.

A hardware feature that aims to detect an ongoing cache-based covert channel attack is proposed by Chen and Venkataramani [2]. Fang et al. present a method to scramble information transmitted over such a channel by leveraging cache prefetchers [5, 6]. This is not applicable to Ariane, which lacks prefetching.

Fadiheh et al. [4] suggest a formal method for detecting vulnerable microarchitectural components within the HW design. While such an approach could prove crucial for the systematic uncovering of microarchitectural covert channels, the question of their mitigation remains open.

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 [7, 8, 10], which is what our temporal fence provides.

7 CONCLUSION

On a simple in-order application-class RISC-V processor we evaluate microarchitectural covert timing channels, previously demonstrated on x86 and Arm processors, and find that they exist with similar capacities on the RISC-V core. We confirm the finding of Ge et al. [9] that existing architected mechanisms are insufficient for preventing those channels. Answering their request for improved hardware support that will enable a principled prevention of such channels, we propose a temporal fence instruction, `fence.t`.

An implementation of `fence.t` on our RISC-V core shows that the naive approach of just clearing all valid bits on cache lines is insufficient. Instead we find that secondary state, in our case the state machine controlling cache-line replacement, can also be exploited as a covert channel, and must be reset as well. We then demonstrate that a complete state flush is successful in eliminating all channels to well below measurement accuracy. We also find that while the (largely ineffective) attempts to close channels with existing instructions are extremely costly, the overhead of `fence.t` is very low, about 320 cycles on our core, which is insignificant at typical partition-switch rates of 1 kHz or lower. We similarly find that the area and power overheads of `fence.t` are insignificant.

Our findings show that the mechanisms requested by OS researchers for principled timing-channel prevention are feasible and low cost, and there seems to be no good reason not to include them into the architecture. This confirms that security should be seen as a hardware-software codesign problem, where OS researchers and architects must collaborate closely.

We hope our findings will support current work that aims at provably eliminating microarchitectural timing channels [14].

ACKNOWLEDGMENTS

We thank Qian Ge and Curtis Millar for their support with the covert channel measurement framework, Wolfgang Rönninger for providing us with an L2 cache, and Florian Zaruba for his help and insights on Ariane. The support of HENSOLDT Cyber and the IDEA League for Wistoff's work at ETH is gratefully acknowledged. Heiser's work was supported by Australian Research Council (ARC) grant DP190103743 and the US Asian Office of Aerospace Research and Development (AOARD).

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*. Springer, 225–242.

[2] J. Chen and G. Venkataramani. 2014. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 216–228.

[3] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 690–695.

[4] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. 2019. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 994–999. https://doi.org/10.23919/DATE.2019.8715004

[5] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. 2019. PrODACT: Prefetch-Obfuscator to Defend Against Cache Timing Channels. *International Journal of Parallel Programming* 47, 4 (01 Aug 2019), 571–594. https://doi.org/10.1007/s10766-018-0609-3

[6] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. 2018. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 187–190.

[7] Qian Ge. 2019. *Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection.* Ph.D. Dissertation. UNSW, Sydney, Australia.

[8] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

[9] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* 8 (April 2018), 1–27. Issue 1.

[10] Qian Ge, Yuval Yarom, and Gernot Heiser. 2018. No Security Without Time Protection: We Need a New Hardware-Software Contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. ACM, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/3265723.3265724

[11] GitHub. 2020. Ariane – GitHub. https://github.com/pulp-platform/ariane [Online; accessed 27-March-2020].

[12] GitHub. 2020. Timing Channel Benchmarking Tool – GitHub. https://github.com/SEL4PROJ/channel-bench [Online; accessed 03-April-2020].

[13] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems* 34, 1 (April 2016), 1:1–1:29.

[14] Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can We Prove Time Protection?. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems*. Bertinoro, Italy, 23–29.

[15] Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*. Oakland, CA, US, 52–61.

[16] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*. San Francisco, CA, 191–205.

[17] R. E. Kessler and Mark D. Hill. 1992. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Trans. Comput. Syst.* 10, 4 (Nov. 1992), 338–359. https://doi.org/10.1145/138873.138876

[18] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. 2019. POWERT Channels: A Novel Class of Covert CommunicationExploiting Power Management Vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 291–303.

[19] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

[20] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[21] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16 (1973), 613–615. https://doi.org/10.1145/362375.362389

[22] Anna Lyons and sel4bench:contributors. [n.d.]. *sel4 benchmarking applications and support library*. Retrieved 2020-04-22 from https://github.com/seL4/sel4bench

[23] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal covert channels on multi-core platforms. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 865–880.

[24] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *IEEE Symposium on Security and Privacy*. San Francisco, CA, 415–429.

[25] Daniel Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive* 2005 (2005), 280.

[26] Colin Percival. 2005. Cache missing for fun and profit.

[27] Wolfgang Rönninger. 2019. *Memory Subsystem for the First Fully Open-Source RISC-V Heterogeneous SoC.* Master's thesis. ETH Zürich.

[28] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423.

[29] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. San Diego, CA, 494 – 505.

[30] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 83–93.

[31] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. https://riscv.org/specifications/privileged-isa

[32] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf [Online; accessed 03-April-2020].

[33] F. Zaruba and L. Benini. 2019. 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* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114