Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core

Nils Wistoff [1]
ETH Zurich
Zurich, Switzerland
nwistoff@iis.ee.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 and University of Bologna
Zurich, Switzerland
lbenini@iis.ee.ethz.ch

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

***Abstract*—Microarchitectural timing channels use variations in the timing of events, resulting from competition for limited hardware resources, to leak information in violation of the operating system's security policy. Such channels also exist on a simple in-order RISC-V core, as we demonstrate on the open-source RV64GC Ariane core. Time protection, recently proposed and implemented in the seL4 microkernel, aims to prevent timing channels, but depends on a controlled reset of microarchitectural state. Using Ariane, we show that software techniques for performing such a reset are insufficient and highly inefficient. We demonstrate that adding a single flush instruction is sufficient to close all five evaluated channels at negligible hardware costs, while requiring only minor modifications to the software stack.**

***Index Terms*—covert channels, timing channels, computer architecture, microarchitecture, operating systems, system security, time protection**

I. Introduction

A covert channel is an information flow that uses a mechanism not intended for information transfer [1], and thereby violates a system's security policy that the operating system (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 [2]. Harder to control are channels that target physical quantities not directly managed by the OS, such as processor temperature [3] or power draw [4]. 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) [5]. For example, the Spectre attack [6] 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 [7]. 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 [8].

In this work, we investigate such mechanisms by implementing them in Ariane, an open-source, application-class, in-order, 64-bit RISC-V 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 capacities 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 prior observations that software-only approaches are expensive and ineffective, even in simple single-issue processors.
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.t` instruction, which flushes specific microarchitectural state, and demonstrate that it completely eliminates the studied channels at an imperceptible hardware cost, a low performance penalty, and with minimal modifications to the software stack.

II. Background

A. Threat Model

We examine covert-channel leakage under a *confinement* scenario [1]: An untrusted program possesses a secret, and the 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

[1]At the time of this work also affiliated with RWTH Aachen and HENSOLDT Cyber GmbH

978-3-9819263-5-4/DATE21/©2021 EDAA

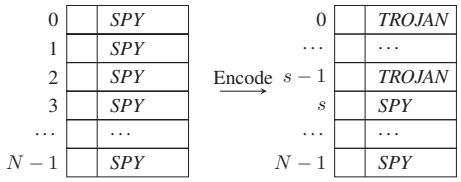


Figure 1: Trojan encoding the secret s into a cache.

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 worst-case leakage; if we can prevent it, 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 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.

B. Prime-and-Probe

Techniques for exploiting covert channels are well established; for our scenario of intentional leakage, the *prime-and-probe* (P&P) attack [9] 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* and *evict-and-reload*.

In a P&P attack, the spy first forces the exploited hardware resource into a known state (*prime*). For the D-cache it traverses a large buffer (in cache-line-sized strides for efficiency); for the I-cache it executes 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 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 could, for example, exploit the time measurements of each individual access and thus extract more information.

C. Time Protection

Time protection is a principled approach to *preventing* timing channels [7]. 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* [10], 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. To prevent any interference between security domains, before handing a resource to a different domain, it must be reset to a state that is independent of execution history. The OS must therefore have the means to reset all microarchitectural state, requiring an extension to the hardware-software contract to refer (in an abstract way) to such non-architected state [8]. The authors specifically show that contemporary Intel and Arm processors lack the mechanisms required for implementing time protection. We will propose such a mechanism in Section III-C.

III. Methodology

A. Measuring Leakage

We adopt the approach of Ge et al. [8] 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.

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 $\{(s_1, t_1), \ldots, (s_N, t_N)\}$.

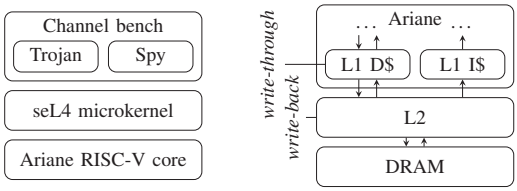


Figure 2: Evaluation platform.

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 4a shows a clear diagonal pattern indicating a channel: if the spy observes a probe time of 380 cycles, it can infer with high confidence that the Trojan has encoded the value *48*.

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 [11]. 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, $\mathcal{M} = 0$. $\mathcal{M}$ is measured in bits; as most of our channel capacities are small, we use millibits ($1\,\mathrm{mb} = 10^{-3}\,\mathrm{b}$).

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 pick uniformly random pairs of input and output values, 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 [12] to compute mutual information $\mathcal{M}$ and zero leakage upper bounds $\mathcal{M}_0$.

B. Evaluation Platform

Figure 2 shows an overview of our evaluation platform.

1) Ariane: We evaluate channels and defences on *Ariane*, an open-source, RV64GC, 6-stage RISC-V core developed at ETH Zurich [13]. It is implemented in SystemVerilog and publicly available on GitHub [14]. 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 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 branch history table (BHT) and a 16-entry branch target buffer (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 [15] that is connected to DRAM. Figure 2 shows the memory architecture.

We partition the L2 cache by colouring [10], which precludes channels in the memory backend and allows us to focus on channels resulting from on-core state.

2) Testbench: Ge's *channel bench* [16], [17] 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 [18], 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.

C. Temporal Fence

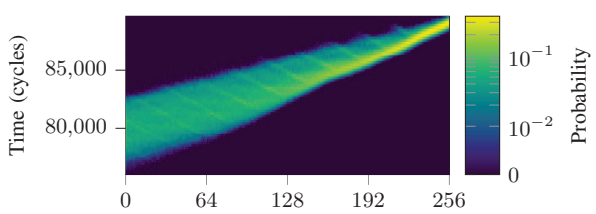
We create a mechanism for resetting microarchitectural state in the form of a *temporal fence* instruction, `fence.t`, which isolates the timing of any subsequent execution from what happened before.[1] Our fence instruction specifically applies to on-core state only, as off-core state can be spatially partitioned. We realise that this definition is less abstract than one 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. If not declared otherwise, we configure `fence.t` to flush all supported components for the highest efficacy.

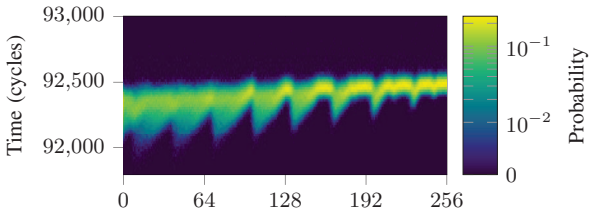
We encode the `fence.t` instruction as a custom U-type instruction with the RISC-V opcode *custom-0*. A bitmap passed as the 20-bit immediate value selects the components to be flushed. When `fence.t` is committed, Ariane's controller sends a flush signal to the stateful microarchitectural components identified by the immediate operand. Our implementation of `fence.t` in Ariane is publicly available on GitHub [20].

We implement two versions of `fence.t`. The first (*naive*) implementation only flushes state that seems directly relevant for the P&P attacks: we reset the L1 cache, TLB, and BTB state by clearing the valid bits (as the Ariane's L1-D is write-through, there is no dirty state to write back). Similarly,

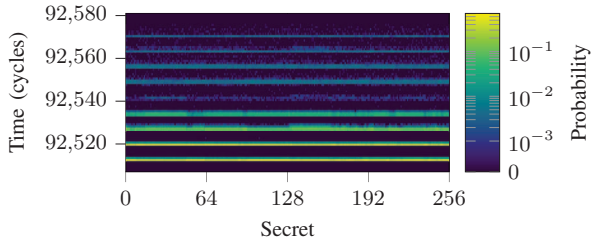
[1] Krste Asanović introduced the notion of a temporal fence on the RISC-V mailing list.



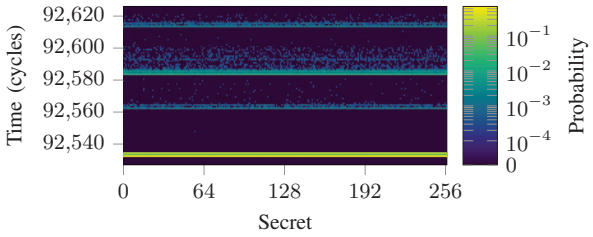
a: Unmitigated. $N = 10^6$, $\mathcal{M} = 1667.3\,\text{mb}$, $\mathcal{M}_0 = 0.5\,\text{mb}$.



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



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



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

Figure 3: L1 data cache channel matrices.

we reset the saturation counter of the BHT. To avoid interfering with in-flight computations, we also flush the pipeline.

As we demonstrate in Section IV-C, this is insufficient, as there is further state that indirectly affects timing, which we also reset in the *final* implementation of `fence.t`. On Ariane these are:

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

IV. Covert-Channel Capacities

A. Baseline: No Time Protection

To establish a baseline and compare to other architectures, we apply the P&P attacks to our Ariane RV64GC core, without time protection in seL4. We observe strong channels through each of the five microarchitectural resources targeted. As shown in the *Unmitigated* column of Table I, capacities range from 400 to 4000 millibits. 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 bit 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 [7].

Figure 3a and Figure 4a 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 channel.

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

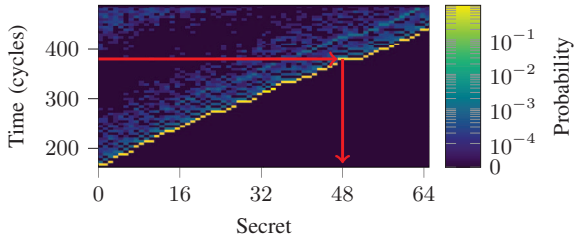
	Unmitigated		Naive `fence.t`		Final `fence.t`	
	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$	$\mathcal{M}$	$\mathcal{M}_0$
L1 D$	1667.3	0.5	10	6	33	39
L1 I$	1905.0	0.5	8	5	38	39
TLB	408.7	0.1	5	6	3	8
BTB	3211.4	0.1	36	59	28	60
BHT	3770.6	0.2	45	59	44	61

B. Using Existing Instructions Only

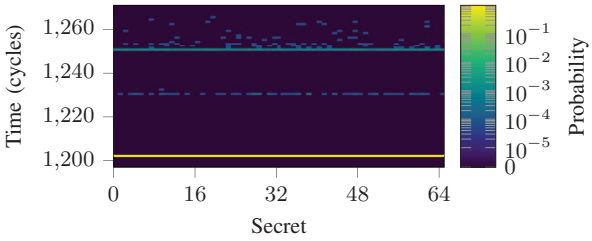
Ge et al. report that neither the x86 nor the Arm architecture provides sufficient mechanisms for implementing time protection. Arm provides targeted L1 cache flushes but no mechanism for flushing other microarchitectural state. The Intel architecture does not even provide that much, and the authors 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.

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 not yet 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 3b 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



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



b: Final fence. $N = 10^6$, $\mathcal{M} = 44.1\,\text{mb}$, $\mathcal{M}_0 = 60.8\,\text{mb}$.

Figure 4: BHT channel matrices.

pattern persists, and the measured capacity is only reduced by 70%. Hence this defence highly ineffective, a result that is much worse than Ge et al. observed for Intel. One reason is the Ariane's replacement policy, which uses a pseudo-random sequence with a period of 256, making it practically impossible to flush the cache through priming. Furthermore, the secondary state discussed in Section III-C is even harder to reset.

C. Using the Temporal Fence

Employing the naive implementation of our fence instruction, we find that it is insufficient, as shown in Figure 3c. 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 I as "Naive `fence.t`", is almost twice the zero-leakage bound, confirming the channel. Concurrently to our work, Vila et al. observed a similar phenomenon on Intel processors [21].

Re-running the experiments with the final implementation of `fence.t` produces the results in the rightmost columns of Table I. All measured channel capacities are now clearly below $\mathcal{M}_0$, meaning that there is no evidence of any residual channel. The channel matrices confirm this: Figure 3d and Figure 4b only show patterns that appear to be random noise (which we confirm by visual comparison between multiple runs).

V. Cost

A. 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 [22] with roughly the same cost as a time-slice preemption, and the seL4 benchmark suite [23] provides a convenient rig for measuring its latency. We therefore use inter-address-space IPC for evaluating flush cost, based on the implementation of Ge et al. Table II 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).

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

Table II: 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)

We found in Section IV-B 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 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 [8].

B. Hardware Overhead

To estimate the hardware costs incurred by the `fence.t` instruction, we synthesise the extended Ariane in GF22FDX technology at 1 GHz. Compared to the original design, the reported area (1.2 MGe) remains similar within the noise floor introduced by the design compiler. Hence, the mechanism does not cause a notable increase in the chip area or leakage power of the design.

VI. Related Work

Past work has approached the hardware mitigation of microarchitectural covert channels from different angles. Page

[24] proposes static partitioning of the L1 cache while Wang and Lee [25] propose locking cache lines. While spatial partitioning can certainly prevent attacks, in the case of the L1, the reduction of available cache space could have a major impact on application performance. Wang and Lee [26] instead aim to defeat attacks by dynamically remapping cache lines.

Fadiheh et al. [27] 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], [16], which is what our temporal fence provides.

VII. 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. [5] 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. In addition, we show that the area and power overheads of `fence.t` are insignificant.

We find 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. Our experience also 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 [28].

Acknowledgment

We thank Qian Ge, Curtis Millar, Wolfgang Rönninger, and Florian Zaruba for their technical support. 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] B. W. Lampson, "A note on the confinement problem," *CACM*, vol. 16, pp. 613–615, 1973.
[2] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *IEEE S&P'13*, May 2013, pp. 415–429.
[3] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal covert channels on multi-core platforms," in *USENIX Security 15*. USENIX Association, Aug. 2015, pp. 865–880.
[4] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu, "POWERT channels: A novel class of covert communication exploiting power management vulnerabilities," in *2019 IEEE HPCA*, 2019, pp. 291–303.
[5] 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.
[6] 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 *IEEE S&P'19*, May 2019.
[7] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing OS abstraction," in *EuroSys '19*. ACM, 2019, pp. 1:1–1:17.
[8] Q. Ge, Y. Yarom, and G. Heiser, "No security without time protection: We need a new hardware-software contract," in *APSys '18*. ACM, 2018, pp. 1:1–1:9.
[9] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.
[10] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, p. 338–359, Nov. 1992.
[11] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.
[12] 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.
[13] 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 VLSI Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.
[14] F. Zaruba, "Ariane," 2017. [Online]. Available: https://github.com/pulp-platform/ariane
[15] W. Rönninger, "Memory subsystem for the first fully open-source RISC-V heterogeneous SoC," Master's thesis, ETH Zürich, Nov. 2019.
[16] Q. Ge, "Principled elimination of microarchitectural timing channels through operating-system enforced time protection," Ph.D. dissertation, University of New South Wales, Oct. 2019.
[17] ——, "Timing channel benchmarking tool," 2015. [Online]. Available: https://github.com/SEL4PROJ/channel-bench
[18] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," 2016. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf
[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM TOCS*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.
[20] N. Wistoff, "Ariane – branch fence-t," 2020. [Online]. Available: https://github.com/niwis/ariane/tree/fence-t
[21] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke, "Flushgeist: Cache leaks from beyond the flush," arXiv:2005.13853 [cs.CR], 2020.
[22] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM TOCS*, vol. 34, no. 1, pp. 1:1–1:29, Apr. 2016.
[23] A. Lyons, "seL4 benchmarking applications and support library," 2014. [Online]. Available: https://github.com/seL4/sel4bench
[24] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.
[25] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," Jun. 2007, pp. 494–505.
[26] ——, "A novel cache architecture with enhanced performance and security," in *MICRO 2008*, 2008, pp. 83–93.
[27] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *2019 DATE*, Mar. 2019, pp. 994–999.
[28] G. Heiser, G. Klein, and T. Murray, "Can we prove time protection?" in *HotOS '19*. ACM, May 2019, pp. 23–29.