fence.t.s: Closing Timing Channels in High-Performance Out-of-Order Cores through ISA-Supported Temporal Partitioning

Nils Wistoff[1], Gernot Heiser[2], and Luca Benini[1,3]

[1] IIS, ETH Zürich, Switzerland / Email: {nwistoff, lbenini}@iis.ee.ethz.ch
[2] UNSW Sydney, Australia / Email: gernot@unsw.edu.au
[3] DEI, University of Bologna, Italy

Abstract. Microarchitectural timing channels exploit information leakage between security domains that should be isolated, bypassing the operating system's security boundaries. These channels result from contention for shared microarchitectural state. In the RISC-V instruction set, the temporal fence instruction (fence.t) was proposed to close timing channels by providing an operating system with the means to temporally partition microarchitectural state inexpensively in simple in-order cores. This work explores challenges with fence.t in superscalar out-of-order cores featuring large and pervasive microarchitectural state. To overcome these challenges, we propose a novel SW-supported temporal fence (fence.t.s), which reuses existing mechanisms and supports advanced microarchitectural features, enabling full timing channel protection of an exemplary out-of-order core (OpenC910) at negligible hardware costs and a minimal performance impact of 1.0 %.

1 Introduction and Related Work

Computing systems are often time-shared between mutually untrusting applications, such as servers running different users' workloads, cyber-physical systems running control and service tasks concurrently, and personal devices that process sensitive data while accessing the internet. To prevent malicious attackers and faulty programs from accessing restricted data, modern instruction set architectures (ISAs) specify a set of mechanisms for memory protection, allowing an operating system (OS) to isolate the memory views of concurrent applications.

However, as the Spectre attacks [1] have demonstrated, existing memory protection is insufficient to isolate applications reliably. Microarchitectural *timing channels* create undesired information transfer across security boundaries. They leverage observable differences in an application's execution time depending on the microarchitectural resource usage of a previously running application.

Some works propose spatial partitioning [2–4] or randomisation [5–7] to close timing channels in data caches. However, these approaches do not address other microarchitectural components, such as branch predictors or buffers. As a general solution, Ge et al. propose *time protection*: A methodology for the OS to partition time-shared hardware (HW) [8, 9]. However, they observe that most processors do not provide the means to temporally partition (i.e., flush) on-core microarchitectural state and call for an ISA extension. Wistoff et al. propose the temporal fence instruction, `fence.t`, to systematically clear microarchitectural state and thus allow the OS to fully isolate applications [10, 11]. A similar approach was presented by Escouteloup et al. [12]. While `fence.t` was shown to be highly effective and inexpensive on a simple in-order core, the viability of `fence.t` on complex out-of-order (OoO) cores remains unknown.

Evaluating `fence.t` in complex OoO cores is relevant for multiple reasons: (a) OoO cores are commonly deployed in security-critical environments such as those described above, (b) they contain large and often pervasive microarchitectural state that needs to be partitioned, (c) flushing and restoring on-core state to and from off-core memory incurs higher penalties and can, therefore, be expected to be more expensive than in simple cores, (d) the overall HW-software (SW) implications of dynamically clearing complex microarchitectural state have not been studied yet. Hence, we specifically make the following contributions:

- we show that integrating `fence.t` in an OoO core with advanced microarchitectural features combining architectural and microarchitectural state comes with challenges,
- we address these challenges by presenting the SW-supported temporal fence (`fence.t.s`) methodology,
- we implement `fence.t.s` in a fully-featured, open-sourced, commercial, high-performance, 64-bit RISC-V core, namely OpenC910,
- we show that `fence.t.s` closes all on-core timing channels in OpenC910 without measurable HW overhead at a minimal performance impact of 1.0 %.

2 Background

2.1 Processor State

In the remainder of this paper, we will use the term *architectural state* to refer to all processor state that is specified as part of the ISA and thus can be accessed directly by SW. Examples are the logical registers, control and status registers (CSRs), and memory. Conversely, *microarchitectural state* refers to all state that is transparent and not directly accessible by SW, such as caches, branch predictors, and potentially any state-holding elements within the core, such as flip-flops and static random-access memories (SRAMs) in the processor pipeline.

2.2 Timing Channels

A *covert channel* is a communication channel that uses mechanisms that are not intended for information transfer [13]. A microarchitectural *timing channel* is a covert channel that leverages differences in execution time due to contention for shared microarchitectural resources. Exploitable resources are any microarchitectural, sequential components with a possible timing impact, such as data caches [14, 15], instruction caches [16], and branch predictors [17].

2.3 Time Protection and fence.t

Ge et al. have proposed time protection as a methodology to prevent timing channels [9]. The key idea is to consistently partition all shared HW resources, either spatially (e.g., through *colouring* the last level cache [18]) or temporally (i.e., resetting the resource to a pre-defined value).

The temporal fence instruction, `fence.t`, was proposed to temporally partition on-core microarchitectural state [10, 11]. It achieves this by writing back dirty cache state, clearing on-core SRAMs and microarchitectural flip-flops, and padding its completion to a worst-case execution time.

2.4 OpenC910

OpenC910 is an industrial, 64-bit, application-class, 12-stage, OoO RISC-V core developed by T-Head Semiconductor Co., Ltd., featuring the RV64GCXtheadc ISA [19]. It was open-sourced in 2021 under the Apache license and features the custom Xtheadc extension [20].

In this work, we leverage the following custom instructions and CSRs: `dcache.call` clears the L1 data cache (L1D), writing back any dirty cache lines. The `mcor` CSR enables targeted invalidations of different SRAMs in the design. SW can trigger an invalidation of the L1 caches and branch predictors by setting corresponding bits. `sync.i` serves as an execution barrier in the instruction stream, ensuring that all instructions preceding it have been completed before executing instructions following it. The `mrvbr` CSR contains the address from which the core will start executing after reset.

3 Temporally Partitioning Out-of-Order Cores

3.1 Challenges of fence.t in Complex Cores

Challenge 1: The Problem of Mixed State `fence.t` (Section 2.3) clears all microarchitectural state and retains architectural state as defined in Section 2.1, meaning that while clearing the processor's microarchitectural state, functionally, `fence.t` behaves as a `nop`.

This approach requires a clear separation of architectural and microarchitectural state, which may not be possible in a complex processor. For instance, *register renaming*, a mechanism previously targeted in side-channel attacks [21], dynamically allocates logical registers to a large physical register file to mitigate data dependencies. A register allocation table (RAT) stores this mapping between logical and physical registers. The RAT is not specified in the ISA or accessible by SW, classifying it as microarchitectural state. Since it impacts execution time, it can be exploited as a timing channel and, therefore, needs to be cleared by `fence.t`. However, it also contains crucial information on the location of the processor's logical registers. Simply clearing it on `fence.t` would, therefore, corrupt the processor's architectural state. We refer to such state as *mixed* state, since it combines microarchitectural and architectural information. Handling mixed state in a way that does not leak timing information while maintaining functional correctness is challenging to achieve in HW alone. Section 3.2 will propose SW support to achieve these goals.

Algorithm 1 `fence.t.s` procedure.

```
1: for all $r \in ArchRegs$ do
2:     stack ← $r$
3: end for
4: $scratch \leftarrow sp$
5: ClearL1D
6: InvalSRAMs
7: ClearFFs
8: $sp \leftarrow scratch$
9: for all $r \in ArchRegs$ do
10:     $r$ ← stack
11: end for
12: PadTime
```

Challenge 2: Reusability of Instructions The `fence.t` instruction performs multiple operations, such as clearing SRAMs, flip-flops, and padding for a worst-case execution time. These operations are orchestrated by a finite-state machine (FSM) in HW upon executing `fence.t`. However, this monolithic approach does not give the OS any flexibility in selecting mitigation mechanisms according to the system's security and performance requirements, nor does it allow the reuse of the mechanisms for other purposes. Therefore, we propose to split the functionality of `fence.t` in multiple instructions performing single tasks.

3.2 Software-Supported Temporal Fence

To address the challenges introduced in Section 3.1, we propose a SW-supported temporal fence (`fence.t.s`) by splitting the functionality of `fence.t` into discrete instructions and executing these as required. An overview of this approach is shown in Algorithm 1.

First, all architectural (logical) registers are stored on the stack. The stack pointer is saved at a well-known location not affected by the temporal fence, such as a scratch CSR. Next, the L1D is cleared by writing all dirty entries back into higher-level memory. This is followed by a (set of) instruction(s) that invalidate on-core SRAMs such as caches, TLBs, and branch predictors. Once this is done, we clear all on-core flip-flops except for the CSRs and resume execution at the next instruction. We restore the stack pointer and the architectural registers and finally pad the execution time to a worst-case delay to prevent leakage through the context-switch latency. This worst-case delay happens when the microarchitecture is not trained for kernel execution, and the L1D is fully dirty and needs to be written back completely in line 5 of Algorithm 1. In particular, note that by saving the logical registers on the stack, the RAT can be safely cleared in line 7 of Algorithm 1 without losing the architectural register values, solving the problem of mixed state discussed in Section 3.2.

4 Implementing fence.t.s in OpenC910

We implement `fence.t.s` in the industrial, 12-stage, OoO OpenC910 core introduced in Section 2.4. We select this core as an evaluation target due to its openness, high performance, and commercial background, making it a suitable target for a representative case study.

The only HW modification needed in OpenC910 is the addition of the `ff.clr` instruction, which clears all on-core flip-flops except for the CSRs according to line 7 of Algorithm 1. To do so, we extend the decoder and the synchronous reset controller and route the required pipeline signals.

Listing 1. `fence.t.s` in OpenC910.

```
fence_t_s:
  // store logical registers
  // on stack
  addi sp, sp, -31*8
  csrw sscratch, sp
  sd   x1, 0*8(sp)
  ...
  sd   x31, 30*8(sp)

  // configure reset address
  la   t0, post_ff_clr
  csrw mrvbr, t0

  // clear and invalidate
  // caches, branch predictors
  dcache.call
  li   t0, 0x70011
  csrs mcor, t0

  // clear on-core flip-
  // flops except CSRs
  sync.i
  ff.clr
post_ff_clr:
  sync.i

  // restore logical
  // registers from
  // stack
  csrr sp, sscratch
  ld   x1, 0*8(sp)
  ...
  ld   x31, 30*8(sp)
  addi sp, sp, 31*8

  ret
```

Listing 1 shows an implementation of `fence.t.s` in OpenC910. We start by storing the logical registers on the stack. The stack pointer is saved on the standard RISC-V `sscratch` CSR. Next, we load the address of the instruction after `ff.clr` into the custom `mrvbr` CSR to continue execution from this address after coming out of reset. Then, the data cache is cleared with `dcache.call`, and the OpenC910's on-core SRAMs (caches, branch predictors) are invalidated using the custom `mcor` CSR. Now, we can safely execute `ff.clr` to clear all on-core flip-flops. We insert `sync.i` instructions to prevent a premature reset. Finally, we restore the logical registers from the stack and return from the function. We pad the execution time of `fence.t.s` to 15 k cycles, which proved sufficient in all experiments even with a fully dirty L1D.

5 Evaluation

5.1 Platform

For our evaluation, we embed OpenC910 into a minimal system with peripherals and memory required to preload and execute binaries. We map this design onto a Xilinx VCU128 FPGA board running at 50 MHz. Figure 1 illustrates this setup.

Furthermore, we port an experimental version of seL4, a formally verified microkernel with strong security guarantees [22], onto this system. Since this work focuses on on-core timing channels (within *Core 0* shown in Figure 1), we configure seL4 to colour the L2 cache of OpenC910 to prevent timing interference through off-core cache refill latencies [18].

5.2 Security Analysis

To evaluate the efficacy of `fence.t.s`, we adopt the methodology of previous works [8–11] and port *channel bench* [8], an evaluation framework for timing channels, to OpenC910. *Channel bench* sets up two applications, a Trojan and a spy, that actively try to communicate through different microarchitectural components using a *prime-and-probe* attack [15]. For a given secret s, the Trojan performs a corresponding action (e.g. evict s lines from the L1D), and the spy subsequently measures its own execution time. The results of these experiments are captured in *channel matrices*; see Figure 3 for examples. The x-axis displays the secret values encoded by the Trojan, and the y-axis denotes the subsequent

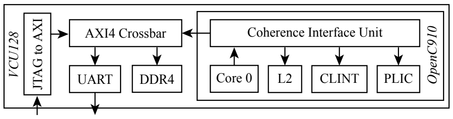


Fig. 1. FPGA evaluation platform.

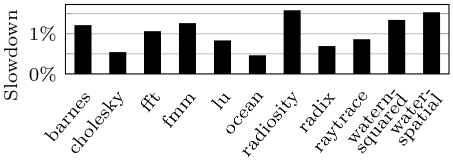
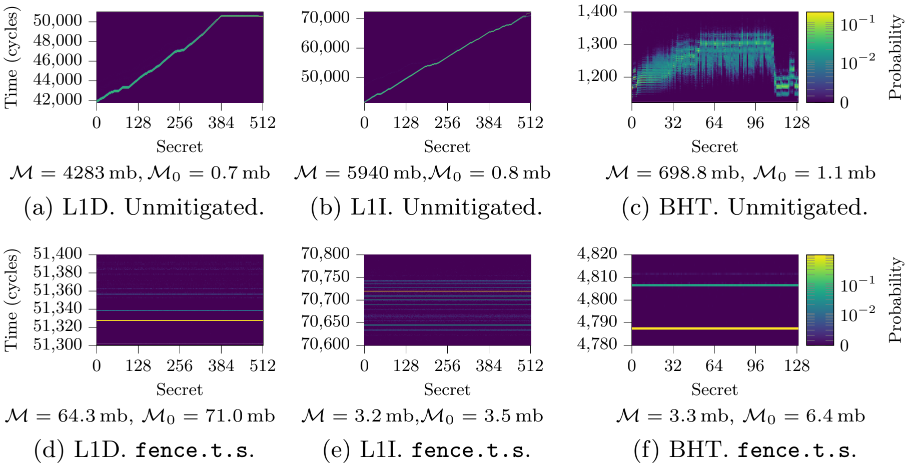


Fig. 2. Splash-2 slowdown by `fence.t.s`.



$\mathcal{M} = 4283\,\text{mb}, \mathcal{M}_0 = 0.7\,\text{mb}$

(a) L1D. Unmitigated.

$\mathcal{M} = 5940\,\text{mb}, \mathcal{M}_0 = 0.8\,\text{mb}$

(b) L1I. Unmitigated.

$\mathcal{M} = 698.8\,\text{mb}, \mathcal{M}_0 = 1.1\,\text{mb}$

(c) BHT. Unmitigated.

$\mathcal{M} = 64.3\,\text{mb}, \mathcal{M}_0 = 71.0\,\text{mb}$

(d) L1D. `fence.t.s`.

$\mathcal{M} = 3.2\,\text{mb}, \mathcal{M}_0 = 3.5\,\text{mb}$

(e) L1I. `fence.t.s`.

$\mathcal{M} = 3.3\,\text{mb}, \mathcal{M}_0 = 6.4\,\text{mb}$

(f) BHT. `fence.t.s`.

Fig. 3. Channel matrices and corresponding mutual information.

execution time of the spy's prime function. Colours indicate the rate of incidence for a given secret-time pair. If a channel matrix shows a horizontal variation, this indicates a correlation between the spy's execution time and the Trojan's secret and, thus, a potential timing channel.

In addition, we use leakiEst [23] to compute the mutual information $\mathcal{M}$ for a channel matrix to quantify the channel's capacity. $\mathcal{M}$ measures the information gained about one random variable (e.g., the Trojan's secret) by observing another (e.g., the spy's execution time) [24]. In our security analysis, we report $\mathcal{M}$ in millibits (mb). Since a non-zero $\mathcal{M}$ can be caused by random noise, we simulate a channel-less measurement with the same output distribution and compute the zero-leakage upper bound, $\mathcal{M}_0$, from it. $\mathcal{M} > \mathcal{M}_0$ indicates a timing channel.

Figure 3 shows the channel bench results on OpenC910 for the L1D, the L1 instruction cache (L1I), and the branch history table (BHT) without timing channel mitigations (top) and with `fence.t.s` on a context switch (bottom). We use these components as examples for our security evaluation but stress that `fence.t.s` aims at preventing timing channels through *any* microarchitectural state. Without mitigations (top), all three components show significant leaks, evident from the diagonal patterns in their channel matrices and their $\mathcal{M}$ being several orders of magnitude greater than the corresponding $\mathcal{M}_0$. Executing `fence.t.s` on a context switch (bottom) reliably closes all timing channels, as there are no variations along the x-axis. The $\mathcal{M}$ values support this, as all three are smaller than $\mathcal{M}_0$.

5.3 Performance Overhead

To evaluate the performance impact of `fence.t.s`, we run one application executing the Splash-2 benchmarks [25] and one idle application concurrently on OpenC910. seL4 performs a context switch every 10 M cycles, corresponding to a typical time slice length of 10 ms at a clock frequency of 1 GHz.

Figure 2 shows the relative slowdown of the benchmarks when calling `fence.t.s` on a context switch. The combined performance impact of the increased context-switch latency (direct costs) and the cold microarchitecture after context-switching (indirect costs) is, on average, 1.0 %, while being consistently below 1.6 % for all benchmark applications.

The low overhead of the temporal fence can be explained by the large number of system clock cycles per OS time slice and, as Ge et al. [8] argue, the low persistance of L1 state across multiple time slices. We stress that the OS can choose *if* and at what interval to perform the temporal fence according to the system's performance and security requirements.

5.4 Hardware Costs

As discussed in Section 4, our HW modifications are limited to adding the minimal `ff.clr` instruction. To evaluate the HW costs of this modification, we synthesise the original and the extended version of OpenC910 in GlobalFoundries 12LP+ FinFET technology in typical corners at a clock speed of 2 GHz. The addition of `ff.clr` does not cause a notable change in the area or timing of the design.

6 Conclusion

This work proposes the SW-supported temporal fence (`fence.t.s`) that closes timing channels even in complex OoO cores. By implementing it in OpenC910, an industry-grade, high-performance RISC-V core, we observe that this processor already provides a set of mechanisms to clear most on-core state required for time protection. We add the simple `ff.clr` instruction, which clears all on-core flip-flops except for thee CSRs at negligible HW costs, and demonstrate that an OS can combine it with custom instructions already provided by OpenC910 to reliably close timing channels at a minimal performance overhead of 1.0 %.

We conclude that by exposing mechanisms already available in commercial HW to SW, an OS can efficiently enforce temporal isolation when needed. Adding this abstraction to the ISA would enable a unified set of mechanisms to close timing channels across different RISC-V implementations.

Acknowledgements

This work was supported in part through the TRISTAN (101095947) project that received funding from the HORIZON CHIPS-JU programme.

References

1. Kocher, P. et al.: Spectre attacks: Exploiting speculative execution. In: IEEE S&P. (2019) 1–19
2. Page, D.: Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive (2005)
3. Domnitser, L. et al.: Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. ACM TACO **8** (2012) 35:1–35:21
4. Kiriansky, V. et al.: Dawg: A defense against cache timing attacks in speculative execution processors. In: IEEE/ACM MICRO. (2018) 974–987
5. Wang, Z. et al.: New cache designs for thwarting software cache-based side channel attacks. In: IEEE/ACM ISCA. (2007) 494–505
6. Qureshi, M. K.: CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. IEEE/ACM MICRO (2018) 775–787
7. Werner, M. et al.: SCATTERCACHE: Thwarting cache attacks via cache set randomization. In: USENIX SEC. (2019) 675–692
8. Ge, Q. et al.: No security without time protection: We need a new hardware-software contract. In: ACM APSys. (2018) 1:1–1:9
9. Ge, Q. et al.: Time protection: The missing OS abstraction. In: ACM EuroSys. (2019) 1:1–1:17
10. Wistoff, N. et al.: Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core. In: DATE. (2021) 627–632
11. Wistoff, N. et al.: Systematic prevention of on-core timing channels by full temporal partitioning. IEEE. Trans. Comput. **72**(5) (2023) 1420–1430
12. Escouteloup, M. et al.: Under the dome: preventing hardware timing information leakage. In: CARDIS. (2021) 1–20
13. Lampson, B. W.: A note on the confinement problem. Commun. ACM **16** (1973) 613–615
14. Hu, W.-M.: Lattice scheduling and covert channels. In: IEEE S&P. (1992) 52–61
15. Percival, C.: Cache missing for fun and profit. In: BSDCan. (2005)
16. Acıiçmez, O.: Yet another microarchitectural attack: Exploiting i-cache. In: ACM CSAW. (2007) 11–18
17. Acıiçmez, O. et al.: Predicting secret keys via branch prediction. In: CT-RSA. (2007) 225–242
18. Kessler, R. E. et al.: Page placement algorithms for large real-indexed caches. ACM TOCS **10**(4) (1992) 338–359
19. Chen, C. et al.: Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension : Industrial product. In: ACM/IEEE ISCA. (2020) 52–64
20. T-Head Semiconductor Co., Ltd.: OpenC910 core (2021) https://github.com/T-head-Semi/openc910.
21. May, D. et al.: Random register renaming to foil DPA. In: CHES. (2001) 28–38
22. Klein, G. et al.: Comprehensive formal verification of an OS microkernel. ACM TOCS **32**(1) (2014) 2:1–2:70
23. Chothia, T. et al.: A tool for estimating information leakage. In: CAV. (2013) 690–695
24. Shannon, C. E.: A mathematical theory of communication. Bell Labs Tech. J. **27** (1948) 379–423
25. Woo, S. C. et al.: The splash-2 programs: Characterization and methodological considerations. ACM SIGARCH Comput. Archit. News **23**(2) (1995) 24–36