Minimal Partitioning Kernel with Time Protection and Predictability

Henrik A. Karlsson
KTH Royal Institute of Technology
Stockholm, Sweden
henrik10@kth.se

***Abstract*—We assess the effectiveness of the temporal fence instruction, introduced by Wistoff et al. (2023), primarily in enhancing both security and timing predictability for embedded systems. Originally demonstrated to eliminate microarchitectural side-channels by flushing the on-core microarchitectural state using the seL4 kernel, we extend the evaluation of the temporal fence to a minimal kernel using RISC-V's PMP to protect applications, examining its implications for both security and safety. Our experiments demonstrate that by using scratchpad memory and the temporal fence, we can improve the security and time predictability of applications on both embedded and application-level processors. Furthermore, we find that the performance overhead introduced by the temporal fence remains within acceptable limits for hard real-time systems, which demonstrates the temporal fence's potential in securing applications against side-channel attacks while enhancing system reliability.**

1. Introduction

Embedded systems, essential to sectors like healthcare, infrastructure, and industry, are growing increasingly complex and are driven by heightened functional and connectivity demands despite facing real-time, space, weight, and power constraints. This complexity poses risks to both safety and security, underscoring the need for better reliability and security tools.

Rushby introduced the separation kernel in 1981, a security kernel isolating system components to enhance fault tolerance and enable independent security analyses [28]. The concept of separation kernel was further developed into the partitioning kernel [29], which also isolates partitions in time to address safety concerns in avionics. Side-channel attacks, which exploit system implementations to infer sensitive information, present significant risks to both general-purpose and embedded systems. Traditional kernels, including the separation and partitioning kernels, are ill-equipped to counter these threats.

To combat such attacks, Ge et al. suggested that new hardware mechanisms that reset the microarchitectural state are needed to address intra-core side-channel attacks [12]. Following this, Wistoff et al. introduced the temporal fence, designed to reset a system's on-core microarchitectural state and eliminate associated timing side-channel attacks [33]. Using an experimental version of the seL4 microkernel that colors the L2 cache and provides copies of the kernel to each security domain, they demonstrate that the temporal fence can be used to eliminate intra-core side-channels.

The approach of seL4, which relies on virtual memory to color the L2 cache, may not be suitable for all use cases. For example, the Keystone Enclave framework and MultiZone Security separation kernel utilize RISC-V ISA primitives such as the Physical Memory Protection (PMP) to create trusted execution environments (TEEs) [4], [18]. While Keystone and MultiZone are both designed to prevent untrusted hosts or partitions from accessing a TEE's resources directly, they do not prevent information from leaking via timing side-channels. To prevent such leakage in these systems, a method without the use of virtual memory is needed as the PMP operates below the virtual memory layer.

Contributions. This paper explores the use of the temporal fence not just for mitigating timing side-channel attacks but also for improving the timing predictability of applications. We specifically consider the use of the temporal fence in improving the security and reliability of systems such as Keystone and MultiZone that uses RISC-V's PMP for memory protection. Although our studies were conducted on an application-grade softcore, the findings also apply to embedded systems.

For our evaluation, we developed OpenMCZ, a minimal partitioning kernel based on MultiZone Security, with the temporal fence added to mitigate timing side-channel attacks. This kernel, alongside the programs under examination, is placed within a 64 KiB scratchpad memory to improve the timing characteristics and mitigate memory-based off-core side-channel. Our security evaluation shows that a simple partitioning kernel like OpenMCZ can effectively eliminate timing side-channels that exploit the on-core microarchitectural state when using the temporal fence and scratchpad memory. Furthermore, our safety evaluation demonstrates that it is also possible to ensure deterministic timing of tasks on a partitioning kernel when using the temporal fence and scratchpad memory.

Lastly, we explore the impact of the temporal fence and kernel on system performance, noting that the overhead varies with task characteristics, reaching up to 14% in some cases. This additional overhead may render both soft real-time and enclave systems impractical. On the other hand, it does not pose any significant concern for hard real-time systems that must be scheduled according to their worst-case execution time.

2. Background

Embedded Systems refer to computers that are specifically designed to perform particular tasks within larger systems. These systems typically have stringent demands

for real-time performance and limitations on size, weight, and power consumption (SWaP). To guarantee reliability, each component is usually run on a separate computer, allowing for independent evaluation of their correctness and timeliness. However, there is a rising demand for integrating multiple components onto a single computer, which can reduce the system's overall SWaP footprint and make room for new functionalities. Yet, such integration risks the system's correctness and timeliness due to potential interference between components. Additionally, as systems become more interconnected, the demand for robust security also increases. To ensure that an integrated system is safe and secure, specialized OS kernels are used to rigorously isolate components spatially and temporally within the shared hardware environment.

Spatial Isolation refers to the concept of isolating a component's resources to prevent both sensitive information and faults from propagating. Components in a distributed system are spatially isolated by default, but those sharing the same hardware are not. For spatial isolation on the same hardware, one can use a separation kernel. First introduced by Rushby [28], a separation kernel partitions the computer's resources among components, provides secure communication between components, and schedules them. The separation kernel effectively emulates a distributed environment, thereby granting the components spatial isolation akin to a distributed environment.

To enforce spatial isolation, secure kernels, such as separation kernels, must rely on hardware such as memory management units (MMUs) or memory protection units (MPUs). MMUs provide both memory virtualization and protection and are used by many secure kernels to enforce spatial isolation [7], [17], [24]. The issue with MMU is that it introduces additional costs in terms of power, size, performance, and jitter. Therefore, secure kernels for real-time and resource-constrained systems often use MPUs, which provide only memory protection but are cheaper, simpler, and have more predictable execution time than an MMU [9], [26], [30], [31].

Time Predictability involves accurately predicting the timing of tasks or processes within a system. A common metric is the worst-case execution time (WCET) used to allocate sufficient execution time for real-time tasks so they can meet their deadlines [14]. However, accurately measuring the WCET of tasks is exceedingly challenging due to modern processor complexity, dynamic behavior, and resource interference [6], [14]. Therefore, practitioners typically estimate the upper bound of the WCET and incorporate a pessimistic margin of error, which may vary depending on the system's complexity, ranging from 10% for simpler systems to 100% or more for more complicated ones.

Temporal Isolation refers to safeguarding a component's schedulability, ensuring that it starts in time to complete its tasks before their deadlines [29]. Effective temporal isolation ensures that, despite any faults, a system component does not impact the schedulability of other components. This does not imply zero impact on their timing but rather ensures that any impact is not severe enough to cause failures.

Temporal isolation can be achieved through priority-driven schedulers like Earliest Deadline First (EDF) [19] and sporadic servers [11]. Although theoretically ideal, their complexity [11], [21], [23], [36] often poses challenges in guaranteeing temporal isolation in practice. Hence, simpler time-driven schedulers are commonly preferred in safety-critical real-time applications [20]. For instance, the ARINC 653 specification for aeronautics necessitates a partitioning kernel [27], which mandates components to be scheduled in a cyclic executive manner while allowing tasks within the components to use alternative scheduling approaches [29].

Time Protection is about mitigating side-channel attacks exploiting microarchitectural footprints of components to infer information [15]. Ge et al. [12] outlined five requirements for time protection: flushing the on-core microarchitectural state; partitioning of the kernel; deterministic data sharing; deterministic flushing of the microarchitectural state; and partitioned interrupts. These measures aim to eliminate side-channels that leak information through the microarchitecture, kernel, and interrupts for a general-purpose OS. To meet these requirements, they identified the need for new hardware mechanisms.

Temporal Fence (`fence.t`) introduced by Wistoff et al. [33] became one such hardware mechanism. The temporal fence is an instruction that resets the on-core microarchitectural state, such as L1 cache and branch predictors, to prevent intra-core side-channels from exploiting said state. Meanwhile, they excluded off-core state such as that of the L2 cache for performance reasons, which must therefore be dealt with separately.

They demonstrate that the temporal fence is effective against microarchitectural side-channels on the CVA6 softcore [35] together with an experimental version of the seL4 microkernel [17]. The seL4 kernel has a two-level hierarchical scheduler, with a cyclic scheduler for domains (i.e., component), and a priority-based round-robin scheduler of threads within domains, guaranteeing both temporal and spatial isolation. They implement time protection by executing the temporal fence in between domain switches. As the timing of the temporal fence execution can leak information, they also implement a mechanism for the temporal fence so it stalls until a set number of cycles (`cspad`) after a timer interrupt. Finally, to counter L2-based timing side-channels, seL4 was modified to color the L2 cache so pages of distinct domains map to distinct cache lines. These pages include copies of the seL4 kernel and domain metadata, thereby isolating the effect of a domain's system calls. These mitigations are evaluated using covert channels that exploit the microarchitectural state. The results show that the mitigations eliminated channels to below measurement accuracy.

3. OpenMCZ

OpenMCZ is a minimal partitioning kernel for embedded multicore RISC-V systems using an RISC-V's PMP for spatial isolation and the temporal fence to provide time protection. OpenMCZ is based on OpenMZ [16], an open-source implementation of the MultiZone Security API [5], introducing better IPC, support for multicore, and time protection. With both spatial isolation and time protection, information flow in OpenMCZ is restricted to explicitely defined IPC channels. Therefore, any inter-component attacks must exploit said IPC channels. Al-

though the kernel supports multicore, we present only the single-core version of the kernel in this paper.

3.1. Kernel Design

Zones are user-space partitions managed by the OpenMCZ kernel running in machine-mode. Machine-mode is the highest privilege level in RISC-V, having access to RISC-V's PMP unit, allowing the kernel to spatially isolate the zones.

Inter-Zone Communication, the IPC of OpenMCZ, includes message buffers, and message queues. Each IPC channel is statically defined and allocated to zones, enabling the controlled flow of information among zones.

Scheduling of zones is done with a cyclic executive, cooperative, or timed-cooperative scheduler:

- **Cyclic Executive.** Zones are scheduled in a cyclic manner with a fixed execution time. Once the execution time has been exhausted, the zone is preempted.
  To increase efficiency, zones may have a dedicated *slack zone* to which remaining execution time can be donated using the `yield()` system call.
- **Cooperative.** A zone is only preempted when explicitly invoking the `yield()` system call.
- **Timed-Cooperative.** Each zone can explicitly yield as in a cooperative scheduler, but they also have a limited time budget.

Figure 1 illustrates a scenario involving three zones: A, B, and C. The zones are scheduled using a cyclic executive scheduler where Zone C is A's slack zone. The scheduling frequency for Zones A and B remains steady, whereas Zone C's scheduling is directly related to Zone A's slack.

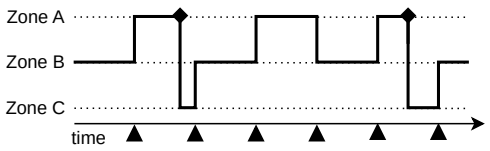

Figure 1. OpenMCZ schedules three zones, A, B, and C. Zone C is a slack zone for A, when zone A yields, zone C is resumed. By using a slack zone, we increase the efficiency of the system while maintaining a predictable schedule.

Time Protection of zones is implemented by using the temporal fence instruction (`fence.t`) on zone-switches. As described in Section 2, the temporal fence enhances system security by flushing the on-core microarchitectural state, preventing information leakage through microarchitectural side-channels. However, executing `fence.t` is not sufficient to prevent information leakages as *when* it is executed may leak information [33]. For instance, if timing `fence.t` depends on the execution of the previous zone, the context switching time will have the same dependency. To address this concern, there is an additional CSR register, `cspad`, that causes the core to stall the execution of `fence.t` until `cspad` cycles after a timer interrupt.

OpenMCZ can use `cspad` in both the cyclic executive and the cooperative scheduling modes. For the cyclic executive schedule, the `cspad` is chosen so it masks the WCET from the timer interrupt to the completion of `fence.t` instruction as illustrated in Figure 2. In the cooperative schedule, a zone can set up timer interrupt so the kernel stalls a `yield()` system call until the timer is triggered, at which point the `fence.t` is executed with padding.

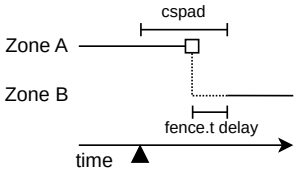

Figure 2. OpenMCZ timer-triggered zone switch from zone A to zone B. There is an execution-dependent delay from the timer interrupt (triangle) to the call of `fence.t` (box) after which zone B starts execution. The `cspad` delays the completion of `fence.t` so the switch is relative to the timer interrupt.

3.2. Safety and Security Properties

Spatial Isolation prevents unauthorized access between zones, mitigating information leakage and interference risks. In OpenMCZ, the developer is tasked with allocating memory in a way that achieves spatial isolation. This allocation is then enforced by the kernel, with the support of the PMP. To ensure effective spatial isolation, memory allocation must adhere to the following principles:

- No zone can access kernel memory, timers, or other memory-mapped regions that control the core.
- Private memory of a zone must be inaccessible to other zones.
- Partitions are prevented from accessing devices that could bypass the PMP, such as DMA controllers.

When these conditions are satisfied, OpenMCZ guarantees the spatial isolation of zones.

Memory sharing is allowed but not recommended due to the safety and security concerns. Shared libraries, buffers, and devices may represent scenarios where memory sharing is considered acceptable, provided risks are carefully managed. For instance, shared libraries should have read-execute permissions only, and for time protection, be allocated to non-cached memory.

Inter-Zone Communication in OpenMCZ enables data exchange between zones through predefined in-kernel buffers, and message queues, ensuring information flow control, and error-free operations. Shared memory communication is also possible, but should be used as a simple buffer due to potential integrity issues with structured data.

Predictable Scheduling guarantees that a system's behavior can be analyzed and replicated. Both cooperative and cyclic executive schedulers are predictable. In the cooperative model, the time of execution is unpredictable, but the order of execution is predictable. Meanwhile, the cyclic executive scheduler ensures that both the timing and execution order of zones are fully predictable.

Temporal Isolation is guaranteed when the cyclic executive scheduler or timed-cooperative scheduler is used.

In these cases, there exists a scheduling cycle, denoted T, defined by the sum of the zones' time slices $t_1, t_2, \ldots, t_n$. In a timed-cooperative schedule, each of these time slices repeats with a maximum period of T. Meanwhile, in the cyclic executive scheduler, a time slice repeats precisely with a period of T.

Time Protection in OpenMCZ requires the temporal fence, cyclic executive scheduler, and careful memory allocation. The use of the temporal fence is decided through the cyclic executive scheduler, ensuring that information is not leaked via the on-core microarchitecture or the scheduler, leaving the off-core state as the primary concern.

The kernel's design ensures that only its code and scheduler data must be shared among all zones, while zone-private memory is partitioned. To make the temporal fence effective, the shared in-kernel memory must be allocated so they do not influence off-core state, such as placing them in non-cached memory. Both private in-kernel and user-space memory of zones can be freely allocated, but to temporally protect security-sensitive zones, their memory allocation must be similar to that of the shared in-kernel memory.

4. Evaluation

We evaluate the safety, security, and performance characteristics of zones running on OpenMCZ. The evaluation platform is a modified Cheshire [25] with a single CVA6 softcore implementing the temporal fence, running on the Genesys2 FPGA. The CVA6 core operates at a core clock speed of 50 MHz, with a real-time clock at 1 MHz, and has a 32 KiB write-through data and 16 KiB instruction cache. Off core we have a 64 KiB write-back last-level cache (LLC), and 64 KiB scratchpad memory (SPM).

In these experiments, the kernel binary is under 8 KiB, and the total experimental setup is less than 64 KiB, allowing everything to fit within the SPM or LLC. The SPM is the default location for the experiments unless explicitly stated otherwise. By using the SPM we bypass LLC-based side-channels, which are challenging to mitigate without an MMU. Furthermore, the SPM is favored for its constant access time, unlike the LLC, which exhibits minor variations in access time, as shown in Section 4.1.2.

It should be noted that for general-purpose OSs, using the SPM like our kernel is often infeasible due to size constraints. For example, the seL4 kernel requires a minimum of 162 KiB of memory solely for the kernel with at least 12 KiB data, including page tables, per thread — something which neither fits inside the LLC or SPM of our evaluation platform.

4.1. Safety

The safety evaluation investigates the time predictability of the kernel and the applications with and without a temporal fence.

4.1.1. Dispatch Period. We assess the time predictability and temporal isolation of the kernel's cyclic executive scheduler by measuring the *dispatch period* of the measurer zone illustrated in Figure 3, both with and without temporal fences.

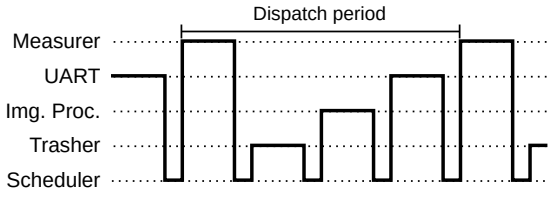

Figure 3. Illustration depicting the dispatch period, which is the time interval between the start of a zone's user-mode execution. The scheduler's share of the execution time is exaggerated in the illustration for clarity.

To simulate a realistic operating environment, we introduce noise into the experiment using three additional zones: an image processor (see Figure 4), a UART driver, and a trasher. The image processor and UART driver zones contribute to the noise by communicating with each other using IPC message queues. The IPC system calls are non-interruptible, delaying timer interrupts and potentially affecting scheduling. Meanwhile, the trasher zone introduces noise by randomly evicting content from both the data and instruction cache, impacting the system's execution time.

`cspad`	-	0	250	500	750	1000
Standard deviation	83	14	13	15	1	0
Worst-case deviation	406	107	114	127	36	0

Table 1. ESTIMATED STANDARD DEVIATION AND WORST-CASE DEVIATION OF THE DISPATCH PERIOD FOR 10,000 SAMPLES PER CASE. THE FIRST COLUMN SHOWS THE DEVIATION WITHOUT `FENCE.T`. THE SUBSEQUENT COLUMNS IS WITH `FENCE.T` AND VARYING `CSPAD` (CYCLES AFTER TIMER INTERRUPT TO STALL `FENCE.T`).

The results of Table 1 show that there is a minor variance in the dispatch period for the case when a temporal fence is not used, and no variance for the temporal fence case with a padding of 1000 cycles. The variance of non-temporal fence case would peak at 8.1 µs on a 50 MHz processor, or around 0.4 µs on a 1 GHz processor. As this variance is already quite small, the temporal fence should not be necessary except in the extreme cases.

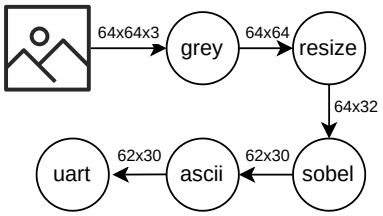

Figure 4. Schematic of the image processing application, outlining tasks for reading an image from memory, greyscaling, resizing, applying the Sobel operator for edge detection, converting to an ASCII representation, and finally, printing the processed image to UART. The edge labels indicate the byte size of the data/image.

4.1.2. Processing Time. We evaluate the impact of the temporal fence, the SPM and the LLC on the time predictability of the image processing zone from Section 4.1.1. This zone reads an image, processes it, and then forwards the result to the UART driver zone via the kernel's IPC message buffers. Execution time is measured

starting from the moment the image is loaded into local memory, and ends when the processed results have been forwarded to the UART driver. To simulate operating variability, we use the *trasher* zone from Section 4.1.1, which randomly evicts entries from the cache. The zones are scheduled using a cooperative scheduler, given that the scheduling does not affect the results.

Our evaluation consists of eight experimental setups, divided equally between scenarios with and without the temporal fence. Each scenario explores different configurations of zone allocations to SPM and LLC to identify their effects on the execution time.

The results of Table 2 show that the temporal fence significantly affects the determinism of the application's execution time, while the SPM and LLC have minor effects. The worst-case variance occurs when both the kernel and the application are in the LLC. Placing the image processing zone inside the SPM reduces the worst-case deviation by 8000 core-clock cycles, roughly 30% of the variance, and with a temporal fence, the variance is almost eliminated. The kernel presence in the SPM has a small influence on the execution time. If both the kernel and the application are in the SPM, the variance is eliminated, indicating that the LLC cache is a source of variance.

Image	Kernel w/ `fence.t`		Kernel w/o `fence.t`	
Processor	SPM	LLC	SPM	LLC
SPM	0/0	12/32	4289/18372	4454/18489
LLC	29/76	36/100	6079/26388	6309/26426

Table 2. THE STANDARD DEVIATION/WORST-CASE DEVIATION OF THE IMAGE PROCESSING TIME MEASURED IN CORE-CLOCK CYCLES.

The results from these measurements provide valuable insights for enhancing system predictability using a specialized kernel like OpenMCZ, particularly for improving WCET estimates and verifying the constant-time execution of programs. Firstly, the use of a temporal fence significantly reduces execution time variance by resetting the on-core state and making it less dependent on the environment, thereby increasing the accuracy of time estimates such as WCET, enabling more aggressive real-time scheduling. Secondly, using the temporal fence with the kernel and application in SPM allows for an evaluation of whether a program has a constant execution time in a real system, isolating observed variations in execution time to the running program or relevant devices.

Conversely, achieving the predictability seen in specialized kernels is significantly more challenging with a general-purpose OS like seL4. Firstly, as discussed earlier, having a general-purpose OS kernel within an SPM is often infeasible due to size constraints, but suppose that there is enough space. Then, another hurdle is ensuring the virtual memory system operates deterministically, as page misses can lead to considerable timing discrepancies. Furthermore, making system call execution times deterministic, a simpler task in specialized kernels is notably more complex in general-purpose systems. Therefore, attaining an equivalent degree of predictability in general-purpose systems poses a significantly greater challenge.

4.2. Security

The security assessment examines the potential for establishing a covert channel between two distinct zones, a Trojan and a spy, where the Trojan should leak one bit of information per scheduling cycle. These zones are managed by a cyclic-executive scheduler and do not share any resources. We specifically focus on memory-based side-channels, as the way memory is used is the primary difference from the experiments of Wistoff et al. [33]. In particular, we investigate whether the temporal fence is effective when the kernel, the Trojan, and the spy are placed in SPM.

The assessment evaluates information leakage by constructing covert channels through two mediums: data cache and instruction cache.

Data cache: The spy primes the D-cache by writing to it; the Trojan writes to memory to signal a '1' or does nothing for '0'; and lastly, the spy measures the access time to the cache, with a longer time indicating a '1'.

Instruction cache: The spy primes the I-cache using the `fence.i` instruction; Trojan uses system calls to signal '1' or does nothing to signal '0', and lastly, the spy measures the execution time of the same system calls, with a shorter duration indicating a '1'.

We measure the effectiveness of these covert channels using discrete mutual information [13] defined as:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x,y) \log_2 \frac{p_{X,Y}(x,y)}{p_X(x) p_Y(y)}. \quad (1)$$

The mutual information quantifies the number of bits of information obtained about an unseen variable Y when observing another variable X. In scenarios where n bits of information are transmitted, the mutual information's range is $[0, n]$, where 0 signifies the absence of information flow, while a n indicates an optimal flow of information.

In our evaluation, the observed variable X is the spy's measurements, the unseen variable Y represents the bit encoded by the Trojan, and the range of mutual information is $[0, 1]$. To calculate the mutual information, we estimate the probability density functions p_X, p_Y, and $p_{X,Y}$ from the sampled data.

Table 3 indicates that without a temporal fence, there is a perfect flow of information, thus a highly reliable covert channel. However, with a temporal fence, the mutual information is nullified, indicating the absence of any information through these channels.

`fence.t`?	D$	I$
No	1000.0	1000.0
Yes	0.0	0.0

Table 3. THE EFFECTIVENESS OF THE COVERT CHANNELS WITH AND WITH `FENCE.T`, MEASURED IN DISCRETE MUTUAL INFORMATION QUANTIFIED AS MILLIBIT. ANALYSIS WAS CONDUCTED WITH 100,000 PAIRS OF SAMPLES TO ASSESS THE COVERT CHANNEL'S EFFECTIVENESS.

The results of the experiments confirm that OpenMCZ correctly uses `fence.t`, and supports the findings of Wistoff et al. [33]. More importantly, the results show that, for specialized systems, time protection can be achieved without kernel partitioning or using cache coloring as suggested by Ge et al. [12] for hypervisors. A significant

limitation of using non-cached memory for time protection is, however, their small capacities. Consequently, the results suggest a partitioning strategy for constrained systems where non-security-sensitive applications use the DRAM, while the separation kernel and security-sensitive applications are allocated to the SPM or similar. The results also suggest an alternative strategy for general-purpose systems like seL4, where shared kernel code can be placed inside the SPM while domain-specific data is cache colored.

4.3. Performance

Performance is assessed with an image processing application depicted in Figure 4, where tasks are encapsulated in separate zones, scheduled cooperatively, and communicate through pairwise shared buffers located on the LLC.

The system's execution time is measured in core-clock cycles across three scenarios: (a) kernel with temporal fences, (b) kernel without temporal fences, and (c) bare-metal execution.

Table 4 presents the performance metrics, with and without UART communication included. Excluding UART communication, a performance bottleneck, reveals a marginal kernel overhead of 0.8%, which escalates to 14% upon introducing a temporal fence. However, when including slow UART communication, the kernel, and the temporal fence's overhead is made insignificant.

	excl. UART	**incl. UART**
Bare-metal	408	8,702
no `fence`	411	8,705
`fence.t`(0)	453	8,748
`fence.t`(2500)	456	8,751
`fence.t`(5000)	466	8,761

Table 4. THE ESTIMATED WORST-CASE PERFORMANCE OF THE IMAGE PROCESSING APPLICATION IS QUANTIFIED IN 1000 CORE CLOCK CYCLES. THE IMPLEMENTATIONS USING THE KERNEL WITH THE TEMPORAL FENCE HAVE THE PAD IN PARENTHESIS.

The results reveal that the impact of temporal fences on system performance depends on the characteristics of the zones. For instance, zones like the UART driver, which inherently have longer execution times, are minimally impacted by the inclusion of temporal fences. In contrast, tasks with shorter execution times encounter a significant performance degradation.

The distinction holds significance for real-world safety- and security-critical applications, particularly concerning the feasibility of temporal fence in real-time systems. In soft real-time systems and enclaves, where reduced performance degrades the system, the increased cost of partition switches or enclave calls may make the system infeasible. Conversely, in hard real-time systems where tasks are scheduled according to their WCET, the temporal fence may have no discernible effect. This can be seen by noting that the state established by the temporal fence must be considered in the WCET analysis of any isolated component.

5. Related Work

OpenMCZ is a specialized separation kernel for real-time resource-constrained embedded devices that also temporally protect system components. There are many separation kernels or partitioning kernels for constrained-device [1]–[3], [8], [26], however, OpenMCZ is the first with implemented and tested time protection. However, for more powerful systems with MMU, we have seL4 [33].

PipMPU is an adaptation of PipMMU to constrained devices [8], slightly larger (10 KiB) than OpenMCZ (4–8 KiB), but has been formally verified in COQ [9], [32]. A popular RTOS is FreeRTOS which has an MPU-based variant [1]. Both of these kernels and many others, have a priority-based scheduling with tasks being dynamically created, unlike OpenMCZ, so the feasibility of implementing time protection is unknown. Meanwhile, kernel with static scheduling and memory allocation such as MultiZone Security [26], on which OpenMCZ is based, can easily implement time protection as demonstrated in this paper.

Adding time protection into priority-based kernels with dynamic task creation faces significant challenges. Firstly, the kernel must operate deterministically with respect to dynamically created resources to prevent any unintended information leakage through its implementation, for instance, via task creation or kernel interactions. Secondly, the frequent task interruptions inherent to priority-based scheduling necessitate careful design to avoid leaking sensitive task information and to mitigate the performance impact of the temporal fence. This requires ensuring that the scheduling does not inadvertently disclose sensitive information and that it is optimized to minimize the temporal fence's overhead by allocating sufficient contiguous execution time for tasks.

An alternative to temporal fences is *security domes* by Escouteloup et al. [10], enabling a comprehensive system partitioning for both single-core and multi-core systems. Unlike temporal fences, which focus on the core-local state, security domes partition both processor cores and external resources into distinct partitions known as domes. At any given time, a core is associated with a single dome, but resources may be shared across multiple domes. The number of domes a resource can support simultaneously depends solely on the implementation. This approach allows for a multi-core system where cores in different domes can access the same resource without interference. This architecture contrasts with the narrower scope of temporal fences, which isolate on-core states, necessitating additional measures such as cache coloring for comprehensive system isolation.

6. Discussion

We evaluate OpenMCZ using Cheshire as it has a CVA6 core implementing the temporal fence, can support multi-core, and is Linux compatible. These features enable us to further develop and evaluate OpenMCZ as a multi-core partitioning kernel or security monitor for TEEs. Moreover, it is open-source, allowing us to develop new hardware, such as a bus and memory if required.

Cheshire's application-level features, such as MMU, do not affect the findings of this paper regarding time protection for embedded systems, as those features are not used in the evaluation. Cheshire has both its LLC and SPM on the bus, meaning that the SPM of Cheshire functions as SRAM. Thus, if we exclude the MMU, LLC,

and DRAM of Cheshire, we obtain an MCU with SRAM accessed through a shared bus.

For multi-core time protected OpenMCZ, there are two cases to consider: multi-threaded partitions, where each partition has multiple threads running in parallel, and parallel partitions, where multiple partitions may run in parallel.

In the case of multi-threading, the primary concern is to limit the inter-core interference of the system. This is because the padding of the temporal fence depends on the kernel's worst-case response time (WCRT) with respect to the scheduler's timer interrupt. If this WCRT is too high due to interference on shared resources, a single-core system may be faster in practice.

If the kernel is placed in shared memory, such as SRAM, the associated shared bus is a source of interference [22]. A standard bus will serve one core at a time, so cores will naturally interfere. Depending on core activity, and the bus, this interference may be significant enough to make the system infeasible. To reduce bus interference, there are special buses for real-time systems that rate-limit a core's requests [34]. For multi-threaded OpenMCZ on shared memory, such a bus may be needed to obtain a reasonable WCRT. However, OpenMCZ is designed so that every part of the kernel, except parts related to inter-core IPC, can be placed in SPM. In such a case, the kernel's WCRT is only affected by interference from inter-core IPC.

In the case of parallel partitions, the situation is more challenging as the kernel and partitions must be interference-free. The kernel and partitions may be placed in SPM to eliminate interference, but this may not be an option if these partitions must access devices on the bus. Therefore, for parallel partitions, we need an interference-free bus, and possibly devices, to mitigate inter-core side-channels.

7. Conclusion

The temporal fence in OpenMCZ significantly improves the safety and security of embedded systems. By using scratchpad memory and the temporal fence we demonstrate that a class of side-channel attacks exploiting microarchitectural states can be eliminated on a resource-constrained system using a minimal partitioning kernel using RISC-V's PMP for memory protection. Beyond eliminating side-channel attacks, we show that temporal fence also enhances timing predictability by bringing the on-core state to a known state. When the temporal fence is combination with deterministic memory we get a fully deterministic program execution even on modern application-grade processors. This enables a more accurate estimate of a task's worst-case execution time, allowing for more aggressive real-time scheduling, and improved protection against timing attacks. Moreover, deterministic execution allows for the evaluation of whether a constant-time program truly runs in constant time on a real-world system, as any variance must be attributed either to the program or related devices, not to concurrently running applications.

Given the broader implications of these findings, the approaches utilized by OpenMCZ could extend to general-purpose systems, enabling time protected and deterministic enclave execution. For example, OpenMCZ could be modified to function as a security monitor, with an operating system like Linux using DRAM memory, while the OpenMCZ and TEEs are allocated to scratchpad memory. Additionally, these insights could simplify existing the time protected seL4 by allocating kernel code to scratchpad memory and applying cache coloring to domain data. This strategy would eliminate the need for duplicating the kernel, simplifying the system architecture, while increasing its time predictability.

References

[1] FreeRTOS-MPU - ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[2] Mbed OS | Mbed. https://os.mbed.com/mbed-os/.

[3] The Zephyr Project – A proven RTOS ecosystem, by developers, for developers. https://www.zephyrproject.org/.

[4] MultiZone Security TEE for RISC-V, June 2019.

[5] Hex-five/multizone-api. Hex Five Security, Inc., November 2022.

[6] Jaume Abella, Carles Hernandez, Eduardo Quinones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, Siegen, Germany, June 2015. IEEE.

[7] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, pages 223–234, Berlin, Germany, 2013. ACM Press.

[8] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. From MMU to MPU: Adaptation of the Pip Kernel to Constrained Devices. In *Artificial Intelligence, Soft Computing and Applications*, pages 109–127. Academy and Industry Research Collaboration Center (AIRCC), December 2022.

[9] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. Pip-MPU: Formal verification of an MPU-based separation kernel for constrained devices. *International Journal of Embedded Systems and Applications*, 13(02):1–21, 2023.

[10] Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier, and Jean-Louis Lanet. Under the Dome: Preventing Hardware Timing Information Leakage. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications*, pages 233–253, Cham, 2022. Springer International Publishing.

[11] Dario Faggioli, Marko Bertogna, and Fabio Checconi. Sporadic Server revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 340–345, Sierre Switzerland, March 2010. ACM.

[12] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, Dresden Germany, March 2019. ACM.

[13] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154, pages 426–442. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[14] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *DROPS-IDN/v2/Document/10.4230/OASIcs.WCET.2010.136*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

[15] Gernot Heiser, Toby Murray, and Gerwin Klein. Towards Provable Timing-Channel Prevention. *ACM SIGOPS Operating Systems Review*, 54(1):1–7, August 2020.

[16] Henrik Karlsson. *OpenMZ: A C Implementation of the MultiZone API*. PhD thesis, KTH Royal Institute of Technology, 2020.

[17] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, Big Sky Montana USA, October 2009. ACM.

[18] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[19] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[20] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[21] D. Locke, L. Sha, R. Rajikumar, J. Lehoczky, and G. Burns. Priority inversion and its control: An experimental investigation. *ACM SIGAda Ada Letters*, VIII(7):39–42, June 1988.

[22] Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022.

[23] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, Porto Portugal, April 2018. ACM.

[24] R. M. Needham and R. D.H. Walker. The Cambridge CAP computer and its protection system. *ACM SIGOPS Operating Systems Review*, 11(5):1–10, November 1977.

[25] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In, July 2023.

[26] Sandro Pinto and Cesare Garlati. Multi zone security for arm cortex-m devices. In *Embedded World Conference*, volume 2020, 2020.

[27] Paul J. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.E.5–1–1.E.5–10, St. Paul, MN, USA, October 2008. IEEE.

[28] J. M. Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, December 1981.

[29] John Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Technical Report NASA/CR-1999-209347, June 1999.

[30] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely, and Stephane Di Vito. An In-depth Study of MPU-Based Isolation Techniques. *Journal of Hardware and Systems Security*, 3(4):365–381, December 2019.

[31] Arash Vahidi. The Monotonic Separation Kernel. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 112–119, Milano, Italy, August 2014. IEEE.

[32] Florian Vanhems, Vlad Rusu, David Nowak, and Gilles Grimaud. A Formal Correctness Proof for an EDF Scheduler Implementation. In *RTAS 2022: 28th IEEE Real-Time and Embedded Technology and Applications Symposium*, Milan, Italy, May 2022.

[33] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Gernot Heiser, and Luca Benini. Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning. *IEEE Transactions on Computers*, 72(5):1420–1430, May 2023.

[34] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[35] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, November 2019.

[36] Yongwang Zhao, Zhibin Yang, and Dianfu Ma. A survey on formal specification and verification of separation kernels. *Frontiers of Computer Science*, 11(4):585–607, August 2017.