

The PMP Snapshot Engine: Fast and Fault-Resilient PMP Reconfiguration for RISC-V

Christian Larmann
TU Delft, CognitiveIC
Delft, The Netherlands
c.j.larmann@tudelft.nl

Abdullah Aljuffri
NGCWS Institute, KACST
Riyadh, Saudi Arabia
aaljuffri@kacst.gov.sa

Adrian Marotzke
NXP Semiconductors
Hamburg, Germany
adrian.marotzke@nxp.com

Alejandro Garza
NXP Semiconductors
Hamburg, Germany
alejandro.garza@nxp.com

Said Hamdioui
TU Delft, CognitiveIC
Delft, The Netherlands
s.hamdioui@tudelft.nl

Mottaqiallah Taouil
TU Delft, CognitiveIC
Delft, The Netherlands
m.taouil@tudelft.nl

Abstract—This paper presents a **Physical Memory Protection Snapshot Engine (PSE)**, a lightweight hardware extension for RISC-V that addresses both performance and security challenges of Physical Memory Protection (PMP) reconfiguration. By storing and restoring full PMP configurations in a single cycle, the PSE drastically reduces the overhead of context switches typically used in Trusted Execution Environments (TEEs) and secure real-time systems. At the same time, the redundant storage and two-dimensional parity protection provide an efficient and effective defense against fault injection attacks that target PMP registers. In 100k randomized trials, our experimental results demonstrate that the PSE can reliably detect and prevent FI-induced privilege escalations, while incurring only 11.7% area overhead. This makes it a practical solution for embedded devices where both efficiency and trustworthiness are essential.

Index Terms—RISC-V, PMP, RTOS, CV32E40S, TEE

I. INTRODUCTION

RISC-V is a rapidly emerging Instruction Set Architecture (ISA), gaining traction as an open-source alternative to existing ISA's [1]. Its modular design and flexibility make RISC-V suitable for mainstream embedded and domain-specific systems. Security must be an integral consideration when designing these systems [2] such as the integration of a Physical Memory Protection (PMP) unit. It is used by Trusted Execution Environments (TEEs) like Keystone [3] and MultiZone [4] to achieve high security by separating applications from each other, using multiple privileges to enforce the isolation and managing access rights to critical data. In addition, secure Real-Time Operating Systems (RTOS) like Zephyr [5] and FreeRTOS [6] use memory protection features to isolate tasks. Many embedded TEEs and RTOSes operate under strict real-time constraints, making the overhead introduced by PMP reconfiguration during application switches critical. A second key challenge is their security; PMP attacks can compromise the

security of TEEs and secure RTOSes, rendering their isolation guarantees ineffective and leaving the entire system exposed. This has been shown before in [7], where fault injection (FI) was used to corrupt PMP registers. Several countermeasures against such FI attacks have been proposed. In general, detection mechanisms such as clock or voltage glitch sensors can be lightweight. However, they are typically limited and do not cover the wide spectrum of FI attacks [8]–[10]. Therefore, it would require the deployment of mechanisms capable of detecting all types of FI attacks to achieve complete protection. More specifically, register duplication of the PMPs would be a simple protection, as offered by many RISC-V cores [11], [12]. However, duplication cannot prevent corrupted data from being written to the PMP from previously corrupted memory. It also comes at the high cost of duplicated area. The authors in [13] proposed a hash-based mitigation that masks each application's jump addresses by XOR'ing them with a hash of the PMP configuration. At runtime, the address is unmasked using the current PMP settings. This way, if the PMP configuration is tampered with, the unmasking will not yield the correct entry point, preventing execution of the application. Despite the need for a hash accelerator, a corrupted jump address could still land in a NOP sled that eventually reaches privileged code, which might be executable with the corrupted PMP settings. Until now, there is no low-cost and secure solution for the PMP reconfiguration.

In this work we present the *PMP Snapshot Engine*: A minimal hardware unit that can rewrite all PMP Control and Status Registers (CSRs) in a single cycle by holding multiple full configurations in area-efficient SRAM. An efficient error detection mechanism is leveraged to mitigate (malicious) corruption in these CSRs. Despite huge performance and security improvements, we show that it incurs only minimal area overhead. The main contributions of this work are:

- Design of the PSE, a hardware unit that accelerates context switches and protects the PMP from FI attacks.
- Extensive fault-injection simulation experiments on a simple bare-metal RTOS, with 100k randomized trials across

The project is supported by the Chips JU and its members including top-up funding by the Netherlands Enterprise Agency under grant agreement No. 101112282, as well as the CONVOLVE project of the EU's Horizon Europe research and innovation programme under grant agreement No. 10107037, and the Federal Ministry of Research, Technology and Space (BMFT) of the Federal Republic of Germany (grant 16KIS1572K, SASVI).

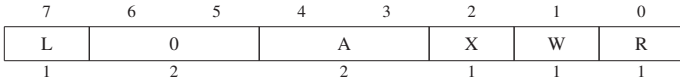


Fig. 1: PMP Configuration Register Format [1].

different attack scenarios.

- Deployment of the PSE integrated into a RISC-V soft-core on FPGA.
- Overhead analysis of the PSE’s area and performance.
- All code and experiments are available at github.com/Quantum-Computer-Engineering/PMP-Snapshot-Engine.

The remainder of the paper is organized as follows. Section II presents the related work. Section III provides the background on RISC-V’s PMP feature. Section IV presents the proposed solution. Section V explains how it efficiently thwarts FI attacks using a simple secure bare-metal RTOS. Section VI analyzes the impact on performance and area of the PSE. Sections VII and VIII discuss and conclude the paper.

II. RELATED WORK

Fault injection (FI) countermeasures have been extensively studied in both hardware and software contexts. Instruction or register duplication provides protection, but come at the high cost of increased execution time or duplicated area. Alternatively, error detection mechanisms can be used, many of which have a cost close to simple duplication [14]. Simpler schemes based on parity checks reduce area overhead [15], however, they provide only limited fault coverage, typically detecting a few bit flips. More heavyweight hardware countermeasures exist, such as dual-core lock-step execution [16], which replicate the entire processor pipeline and compare results cycle-by-cycle, adding prohibitive area and power overhead.

Control-flow checking [17], [18] is a software-only countermeasure that enforces the intended execution flow by inserting and verifying runtime signatures. Classical schemes are effective, but they usually introduce performance overheads of 10 to 75 %. Recent embedded-focused solutions, such as Kage [19], reduce this cost to only $\approx 5.2\%$ runtime overhead, albeit with increased code size and intrusive compiler modifications. Control-flow checking can mitigate FI attacks that manifest as instruction skips or branch-manipulation faults. However, they are ineffective against data corruption faults in PMP configuration registers, which leave the control flow intact but allow unprivileged processes to access unauthorized memory.

Attacks on PMP configuration registers can directly undermine the isolation guarantees of TEEs and secure RTOSes that rely on the PMP mechanism. Qu  n  herv   et al. [7] showed how clock glitch attacks can lead to corruptions in the PMP configuration registers on a CV6 RISC-V core. Nashimoto et al. [13] took a more general perspective and developed a clock glitching scheme to circumvent PMP protection for TEEs by skipping instructions during the process switch, causing some PMP registers to retain values from the previous process. They proposed a countermeasure that calculates a hash value over the whole PMP configuration, which must be combined with a precomputed value for an enclave to calculate its entry address. Despite the hashing algorithm slowing down the PMP

reconfiguration, the number of entry points to return to the enclave is fixed, and PMP regions cannot be changed during run-time.

In summary, existing FI countermeasures either impose prohibitive overheads (e.g., duplication, lock-step execution), address only control-flow deviations while leaving critical CSRs unprotected (e.g., CFI), or restrict PMP flexibility (e.g., hash-based schemes). To the best of our knowledge, no prior work provides a lightweight mechanism that not only accelerates PMP reconfiguration but also protects its configuration registers against FI-induced corruption.

III. PHYSICAL MEMORY PROTECTION (PMP)

RISC-V offers an optional Physical Memory Protection (PMP) feature that can be used to restrict read, write, or execute accesses to certain memory regions at runtime [1]. PMP requires a processor to support at least two privilege modes: machine or M-mode and user or U-mode. Optionally, a third privilege mode can be used, i.e., supervised or S-mode. The address ranges of the memory regions with their associated permissions are configured in the Control and Status Registers (CSRs) `pmpaddrx` and `pmpcfgx` (x is the index of the memory region (0-63)). The `pmpaddrx` register holds the address, and `pmpcfgx` holds the configuration as shown in Figure 1. The three lowest bits R, W, and X respectively set the read, write, and execute permissions of the region. The two bits of the field A specify the address modes. Three different options can be selected for the address mode field A: Top of Range (TOR), naturally aligned four-byte (NA4), or naturally aligned power-of-two (NAPOT). TOR specifies ranges by indicating the upper boundary of the region in `pmpaddrx`. The lower boundary is defined by the address of the previous region, or 0 in case it is the first region. NA4 specifies address ranges that are aligned to four-byte boundaries. With this, only 4 bytes can be protected by one region. NAPOT extends this concept to address ranges that align with any power-of-two boundary. Bits 5 and 6 are reserved, and the highest bit L is the Lock-bit which is used to enforce permissions in M-mode. The PMP configurations can only be modified in M-mode and they are enforced when the system is set back to S-mode or U-mode. The PMP checks are applied to all memory accesses, i.e., instruction fetches and data accesses.

IV. PMP SNAPSHOT ENGINE (PSE)

In this section, we introduce the PSE concept, define its threat model, describe the implementation in detail, and finally analyze its security.

A. PSE Concept

In a secure system, every time a context switch occurs, either between enclaves in a TEE or between tasks in an RTOS, the memory access rights have to be updated as each enclave or task typically has memory restrictions. In RISC-V, this is managed by the PMP, and all PMP-CSRs must be updated with configuration values stored in memory. Ideally, these updates should be very fast. However, for each region 40 bits of data (consisting of 32 bits of `pmpaddr` and 8 bits of `pmpcfg`) must

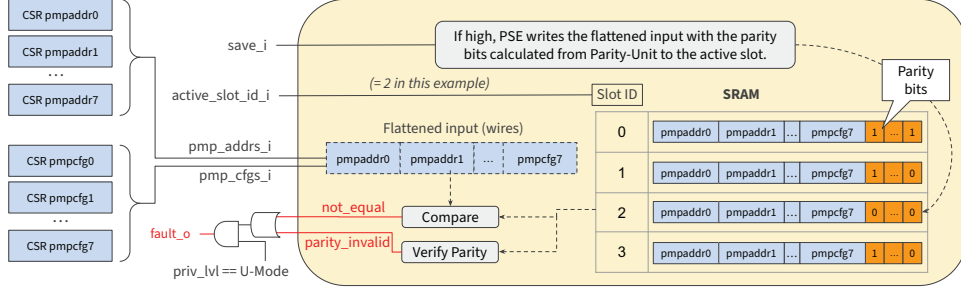
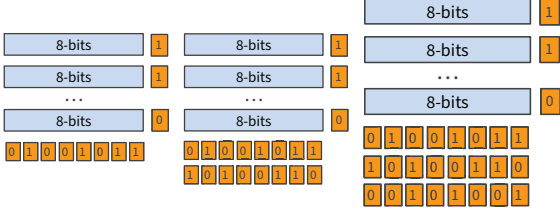


Fig. 2: Added PMP Snapshot Engine (PSE) With 4 Slots Into the CSR Unit.



(a) $D = 2 \cdot 2 = 4$ (b) $D = 2 \cdot 4 = 8$ (c) $D = 2 \cdot 8 = 16$

Fig. 3: 2-Dimensional Parity with Different Column Parities.

be loaded from memory and written to the CSRs. We refer to this data as a *PMP configuration* from now on. Typically, between 8 and 64 regions are used and hence, a context switch may require transferring several hundred to a few thousand bits of PMP configuration data, significantly increasing the context-switch latency.

The PSE is a hardware unit that is added to the CSR unit, as shown in Figure 2. It is able to write the PMP configurations of all regions simultaneously. It has a small SRAM memory that stores all PMP configurations for defined tasks and/or enclaves. Each PSE slot consists of a “snapshot” holding N PMP configurations (up to 64). During run-time, a complete snapshot can be loaded in a single cycle to the PMP registers. The active snapshot is determined by the slot ID which is set during M-mode; it belongs to a task or enclave. Custom CSRs are added to communicate with the PSE, e.g., to initialize a new snapshot, modify its content or reconfigure the PMPs.

B. Threat Model

We assume that attackers have the capability to inject faults that can corrupt PMP-CSRs or the PSE or skip single instructions during execution. The adversary can execute arbitrary code in U-mode but does not have control in M-mode. The attacker’s goal is to corrupt PMP configurations to violate memory isolation. Under this model, our design aims to detect all single-bit and random multi-bit corruptions, as well as single-instruction skips.

C. Efficient Countermeasure Against FI

As explained before, registers in the CPU core are susceptible to FI attacks. This also applies to the SRAM, where laser or EM FI attacks can cause bit flips [20]. We propose an efficient solution to exploit the entries in the PSE that already holds the data in SRAM to detect corruptions induced by any FI attack. Unlike prior approaches discussed in Section II, our

mechanism provides area-efficient protection without sacrificing reconfiguration flexibility, while simultaneously improving the performance. We consider the PMP unit the backbone of secure RTOSes and TEEs, and argue that by ensuring robust process isolation, the impact of FI attacks on other parts of the processor becomes significantly less powerful. In our design, we identified four attack vectors for FI injections:

- (1) Corrupt the data in the standard PMP-CSRs.
- (2) Corrupt the data in the PSE SRAM.
- (3) Skip instructions that write to the PMP-CSRs.
- (4) Skip the instruction that saves the snapshot to the PSE.

The PSE mitigates (1) by comparing the SRAM output (see also Figure 2) with the PMP-CSRs registers. In case they mismatch, either the PMP registers or the SRAM has been faulted. The check is only active when the core is in U-mode. When a fault is detected, the fault signal (i.e., *fault_o*) is set to high. The PSE mitigates (2) by adding parity bits to the PSE slots. The parities are checked each cycle and can detect faults in case some bits are corrupted in the SRAM. We use a two-dimensional parity scheme, which is more reliable than one-dimensional parity, by allowing cross-verification [21]. For this, a PSE slot is structured as a matrix as shown in Figure 3. A snapshot can be divided into 8-bit chunks and parity bits are added for each row and column. Three configurations are shown that differ in the number of column parity bits. These additional column parity bits directly influence the code’s error-detection capability. The strength of a code is measured by its minimum Hamming distance D , defined as the smallest number of bit flips that can transform one valid matrix into another. A code with distance D can detect up to $D - 1$ erroneous bits. For product codes like two-dimensional parity, D is the product of the distances of the row and column codes.

The row parity bits in the figure are calculated by summing the number of 1s in each row and taking the results modulo 2, resulting in a row distance of 2. With a 1-bit column parity as in Figure 3a, the same is true for the the column, resulting in a total distance D of 4. In Figure 3b, 2 parity bits are used for each column, leaving 4 different possibilities for the 2 parity bits. The bits are calculated by adding up the 1s and taking the sum modulo 4, resulting in a column distance of 4 and a D of 8. Arbitrarily more column bits can be added to detect more bit flips. Figure 3c contains 3 bits for the column and hence $D=16$ and up to 15 bit flips can be detected.

The block width of 8 bits in Figure 3 is chosen as an example, however, the block width does not change the distance.

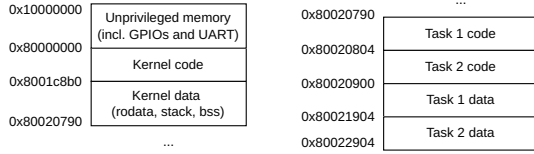


Fig. 4: Memory Layout Used in Experiments.

Therefore, the block width can be chosen such that the number of added parity bits is minimal. The overhead of a block-based layout with N_{cbits} column parity bits can be determined by dividing the width of a slot by the block size and adding the block size for each parity row. Differentiating and setting $O'(b) = -\text{width}/b^2 + \text{rows} = 0$ yields the optimal block width, expressed in Equation 1.

$$b_{\text{opt}} = \sqrt{\frac{\text{width}}{N_{\text{cbits}}}} \quad (1)$$

Attack vectors (3) and (4) are not mitigated by the PSE itself but by comparing the values and duplicating the write instructions when writing to the PSE, as shown in Algorithm 1. This allows the detection of single skipped instructions. Two identical copies of the addresses and configurations are provided to the algorithm to ensure that the memory holding these settings has not been tampered with. The first copy is written to the CSRs, while the second serves as the reference for comparison. If they differ, it indicates that an instruction was skipped or altered, and the fault handler is invoked. Eventually, after all configurations have been written, they are stored in the PSE. If this last store operation is skipped, the slot will remain empty, i.e., all PMP regions are turned off, which will result in a fault once the slot's enclave or task is launched as the PMP-CSRs will have different content.

D. FI Security Analysis

This parity scheme is a simple and efficient method to detect bit flips from corruption. Flipped bits in the PMP-CSRs will always be detected as they are compared to the active PSE snapshot, which only allows for one valid bit string. Flipping bits in non-active snapshots can be a target for an attacker. In the simplest setting with only one parity bit for each column, the attacker needs to flip exactly a multiple of 4 bits. However, a valid state is only achieved if the positions of the flipped bits form a rectangle. For example, flipping the first four bits in the first row preserves the row parity, but results in incorrect parity in each of the corresponding columns. This implies that an undetected error requires the simultaneous flipping of specific bits across multiple rows and columns. This demands precise fault injection. Such precision is typically infeasible without specialized equipment and significant technical expertise. This difficulty is increased by using more column parity bits.

V. FAULT INJECTION CORRUPTION ATTACK EXPERIMENTS

This section demonstrates how effective the PSE can mitigate the impact of FI attacks.

Algorithm 1: FI-Resilient PMP Initialization.

Input: $A^{(1)}, A^{(2)}$: identical arrays of PMP addresses
Input: $C^{(1)}, C^{(2)}$: identical arrays of PMP configs

// Phase 1: Write and verify addresses

foreach $r \in R$ **do**

- // (1) Commit address (first copy) to its `pmpaddr` CSR
- `pmpaddr[r] ← A(1)[r]`
- // (2) Verify address against the second copy
- `addrread ← pmpaddr[r]`
- if** `addrread ≠ A(2)[r]` **then**
- | `faultHandler()`

// Phase 2: Write and verify configurations

foreach $r \in R$ **do**

- // (3) Commit configuration (first copy) to `pmpcfg`
- `pmpcfg[r] ← C(1)[r]`
- // (4) Verify configuration against the second copy
- `cfgread ← pmpcfg[r]`
- if** `cfgread ≠ C(2)[r]` **then**
- | `faultHandler()`

// Phase 3: Save to PSE

`saveCSRsToPSE()`

TABLE I: PMP Region Configuration for RTOS.

Index	Addr. mode	Permissions	Address
0	OFF	-	Unprivileged memory begin
1	TOR	rw	Unprivileged memory end
2	NA4	rw	Task stack begin
3	TOR	rw	Task stack end
4	NA4	x	Task code begin
5	TOR	x	Task code end

A. Experimental Setup

We set up a minimal bare-metal RTOS to run two tasks. The RTOS runs in M-mode and the tasks in U-mode. The RTOS initializes the two tasks by setting up a Task Control Block (TCB) structure for each task and preparing the tasks' contexts by saving all initial register values `x0-x31` and `mstatus` on the tasks' stack. The RTOS provides a simple `delay()` function that issues an `ecall`, putting the calling task to sleep and activating the next ready task.

The memory layout is shown in Figure 4. It begins with a shared region that also contains memory-mapped GPIO and UART peripherals, followed by the kernel code and data, and the code regions for each task and their stacks.

Table I shows how the PMP regions are configured for each task. Region 0 and 1 grant full read, write and execute access to the unprivileged memory which is shared by all tasks and therefore identical across them. For this experiment, GPIO access is used to verify whether the illegal access attempt was successful, which is explained later. Region 2 and 3 give read and write access to each task's stack, and regions 4 and 5 gives executable access to the code. These regions are task-specific and change at each task switch. Additional regions could be added for task-specific needs or shared memory.

The execution flow is depicted in Figure 5. After an initialization phase, Task 1 is started and calls `delay()`. Then the core switches to M-mode to perform a context switch, which entails saving the context (SC) of Task 1, finding the next ready task

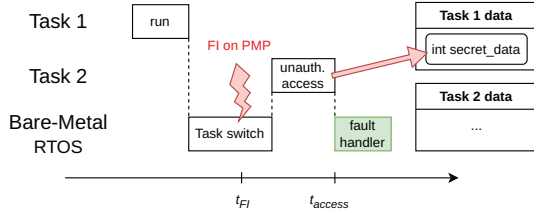


Fig. 5: Attack scenario.

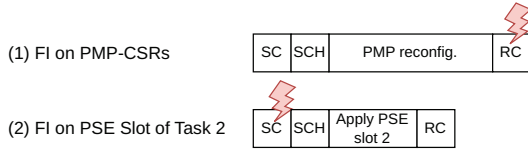


Fig. 6: FI timing for each attack.

by the scheduler (SCH), reconfiguring the PMP, and restoring the context (RC) of Task 2. At this point, a fault is injected into the PMP unit. Thereafter, Task 2 is started, which tries to read private data of Task 1. Normally, the core should jump into the fault handler due to a PMP violation. However, in case the attack is successful due to FI, the PMP settings may be corrupted which allows Task 2 to run and access the secret data.

We used Questasim to simulate the fault injection campaign at RTL level. Questasim supports warm resets, enabling rapid return to a known state. This allows us to perform fast and accurate FI at the desired phases, such as RC and SC in Figure 6. For example, for attack scenario (2), we corrupt the PMP configuration by injecting faults in the PSE slot of task 2 and subsequently launch Task 2, and repeated this setup 100k times. After the corruption takes place, the access attempt of Task 2 results in one of 3 different outcomes: (A) A PMP violation is detected (either because Task 2 cannot access its own code and data anymore due to PMP-CSR glitches, or because of the access attempt to the secret data of Task 1) and the fault handler is entered. The fault handler sets the “done”-GPIO to 1 and the “pass”-GPIO to 0. The attack is unsuccessful in this case. (B) No PMP violation is detected and Task 2 successfully reads the variable to Task 1. In this case, Task 2 sets both “done” and “pass” to 1 and the attack is successful. (C) PMP violation is detected but the RTOS cannot access the fault handler because a corrupted PMP rule has the Lock-bit set to 1, which even restricts the RTOS in M-mode. In this case no GPIO is set. Also, here the attack is unsuccessful. The script waits for the “done” signal, logs the outcome and “pass” bit, and in case scenario (C) occurs, a timeout will be detected. Next, we describe the four attack scenario’s in more detail with their outcomes.

B. Attack (1)

In attack scenario (1), the PMP-CSRs are configured by reading values from the task’s TCB and writing them to the CSRs. This constitutes the conventional procedure for TEEs and secure RTOSes. For this attack, the FI must happen after the PMP-CSRs have been written, so corrupted values are present when Task 2 is run.

Table II presents the results with and without the security features for 100k FI runs. Without any added security features

TABLE II: FI Success Rate Comparison.

Attack	Description	Success Rate (%)			
		Baseline	PSE		
			D=0	D=4	D=8
1	On CSR-PMP	3.30	0.00	0.00	0.00
2	On PSE Slot	–	3.40	0.00	0.00

(i.e., no PSE), the chance for successfully accessing the variable with attack (1) is 3.3%. After adding the PSE, the comparison method detects all corruptions successfully, resulting in an attack success rate of 0.00%.

C. Attack (2)

In attack scenario (2), the goal of the FI is to corrupt the values in the PSE in slot 2 (while slot 1 is active). When the context changes from slot 1 to 2 (i.e., from task 1 to 2), the PMP-CSRs are written by the PSE in one shot. Hence, the FI must take place before the PMP-CSRs are written by the PSE.

The success rate of the attack on the PSE is 3.4% when no parity bits are used. As soon as parity bits are added, all illegal access attempts are successfully detected.

D. Attacks (3) and (4)

We injected instruction skips at every possible point during task initialization and task switching. In case 16 PMP regions are used, 40 instructions are required to reconfigure the PMP registers. In attack (3), without the countermeasure enabled, skipping the write to `pmpaddr2` leaves the register set to the previous value which belongs to Task 1, granting access from Task 1’s stack start to Task 2’s stack end and exposing secret data. In both attacks (3) and (4), skipping the load of the value for `pmpaddr1` reuses the value of `pmpaddr2`, so the unprivileged region ends at the start of Task 2’s stack, exposing all of Task 1’s stack. Thus, attack (3) is possible by targeting 2 of the 40 instructions, whereas for attack (4) only one instruction leads to a successful attack if skipped. With the PSE present, skipping single instructions in attack (3) causes a fault, as the PMPs are still configured for Task 1 while the PSE is expecting Task 2’s settings. Similarly, skipping any instructions during task initialization (attack (4)) will be detected using Alg. 1.

VI. PERFORMANCE AND AREA OVERHEAD

To evaluate the performance, we select the CV32E40S RISC-V core [22], which is an open-source soft-core supporting PMP. We modified the core and added the PSE unit to it and synthesized it on a Genesys 2 FPGA board [23] using the AMD Vivado Design Suite v2024.2 [24]. The core supports I (integer), M (multiply), and the C (compressed) extensions, and implements machine- and user-mode. The instruction and data caches are 8kB in size and have an access latency of two cycles. The latency to main memory is 30 cycles.

In Table III, we compare the overhead introduced by the PSE in number of cycles for the operations that impact the real-time behavior of the system. The PSE significantly accelerates the PMP reconfiguration time and context switching by a factor of 23.8 and 1.56, respectively. The PMP reconfiguration itself takes only one cycle, however, reading the slot number from the

TABLE III: Performance Overhead Expressed in Cycles.

	No PSE	PSE
PMP reconfiguration only	283	12
Context switch (SC, SCH, PMP, RC)	757	486
Task initialization	1,212	1,808

TABLE IV: SRAM Usage in Bits for One Slot.

PMP Regions	Block Width	Width Including Parity Bits			
		D=4	D=8	D=32	D=512
8	320	356	371	392	421
16	640	691	712	741	783
32	1280	1351	1381	1423	1482
64	2560	2661	2703	2762	2846

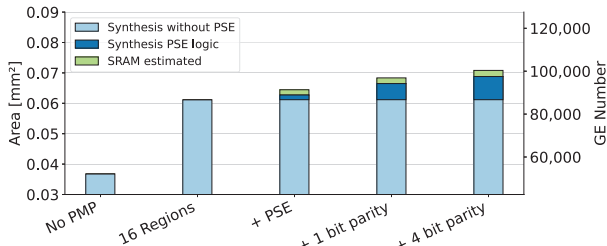


Fig. 7: Area Overhead of PMP-CSR and PSE Unit for 8 Slots.

task’s TCB takes additional cycles. Note that the execution time is not affected by the parity bits check, as their computation is not in the critical path. The task initialization increased by a factor of 1.49 due to the redundant write operations in Alg. 1. However, as it is performed only once at task creation, this overhead is acceptable.

To evaluate the area, we synthesized the CV32E40S core in TSMC 40 nm technology [25] at 100 MHz using Cadence EDA tools [26]. Since our synthesis tools cannot directly model the precise SRAM dimensions used in the PSE, we estimate the overhead based on TSMC’s reported area of $0.242 \mu\text{m}^2$ per SRAM cell [25], adjusted by a correction factor of 1.4 (corresponding to an assumed utilization factor of ≈ 0.7 [27]) to account for surrounding logic. The PSE slot widths of adding parity bits for different numbers of PMP regions is shown in Table IV. The block widths were selected according to Equation 1. The area results for the PSE when integrated into the core are shown in Figure 7. In the baseline configuration without PMP, the core occupies 0.0368 mm^2 . With 16 PMP regions enabled, the area increases to 0.0612 mm^2 . In case we add a PSE with 8 slots, the area increases to 0.0644 mm^2 , which is a 5.4% increment when 16 PMP regions are enabled. When parity protection is enabled, additional area is needed to store the parities in the SRAM and for the parity logic. Using a parity scheme with 1 column bit ($D=4$) raises the area to 0.0684 mm^2 , resulting in an overhead of 11.7% compared to the case where 16 PMP regions are activated. 4 column parity bits ($D=32$) results in a total of 0.0708 mm^2 , which equals a 15.8% overhead.

VII. DISCUSSION

The experiments show that the PSE can reliably detect corruption of PMP configurations. Our attack model considered random FI across PMP-CSRs or an entire PSE slot, which is representative of practical FI attacks. A limitation is a coor-

TABLE V: Comparison of PMP Protection Approaches.

Work	Register Corruption	Instr. Skip	Area Overhead	Context Switch Speed
Shadow PMP-Regs [11]	Yes (Duplication)	No	+6.4%	unchanged
Glitch Detector [10]	No	Yes	+ 0.47%	unchanged
Nashimoto et al. [13]	Yes (Hash)	Yes	unchanged	1.8–2.2× slower
PSE with Alg. 1	Yes (Parity)	Yes	+ 11.7%	1.56× faster

ordinated multi-bit fault pattern attack, but they require highly specialized tools and expertise. The PSE strengthens the PMP, which we regard as the most critical protection mechanism in embedded systems. Note that the PSE cannot address attacks that are beyond the scope of the PMP itself, such as side channel attacks.

Table V summarizes how PSE compares with representative state-of-the-art approaches. Prior mitigation schemes typically address only one aspect of the problem. Shadow PMP registers detect corruption but remain vulnerable to instruction-skip attacks [13], and we measured a 6.4% area overhead when enabling them on the CV32E40S. In contrast, the PSE’s atomic reconfiguration prevents skipped instructions from leaving PMP-CSRs with inconsistent values. In addition, 6.4% more area is only slightly less than the PSE with parity check. Conversely, glitch detectors are able to detect timing anomalies with very low area but provide no protection for PMP contents. The hash-based verification of Nashimoto et al. [13] protects against both classes of FI attacks, but comes at a significant cost for the computations, slowing down PMP updates. Since it is purely a software approach, it adds no area overhead. In contrast, the PSE uniquely combines fast reconfiguration with lightweight parity-based error detection, providing resistance against both corruption and instruction skips at a modest 11.7% area overhead. Systems that reconfigure all PMP regions during each context switch (e.g., FreeRTOS, Zephyr, or TEEs like MultiZone) benefit most, as shown in our experiments. TEEs that rely on static PMP mappings and toggle only a subset of regions (e.g., Keystone) may see smaller performance gains, but still benefit from the added FI resilience. In summary, PSE balances efficiency and security in a way that prior approaches (Table V) do not, providing a practical hardware primitive for secure PMP reconfiguration.

VIII. CONCLUSION

This paper introduced the PMP Snapshot Engine (PSE), a lightweight hardware extension for RISC-V that addresses both performance and security challenges of PMP reconfiguration. By storing and restoring full PMP configurations in a single shot, the PSE drastically reduces the overhead of context switches in TEEs and secure real-time systems. At the same time, the redundant storage and two-dimensional parity protection provide an efficient and effective defense against fault injection attacks that target PMP registers, while only incurring 11.7% area overhead. We demonstrated that the PSE can reliably prevent FI-induced privilege escalations by simulating faults in 100k trials for different attack scenarios. Our results confirm that the PSE is a compelling building block for robust and efficient Trusted Execution Environments on RISC-V.

REFERENCES

- [1] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanović, “The RISC-V Instruction Set Manual: Volume II,” <https://riscv.org/specifications/ratified/>, May 2025, [Online]. Accessed: Sept. 14, 2025.
- [2] J. Anders, P. Andreu, B. Becker, S. Becker, R. Cantoro, N. I. Deligiannis, N. Elhamawy, T. Faller, C. Hernández, N. Mentens, M. N. Rizi, I. Polian, A. Sajadi, M. Sauer, D. Schwachhofer, M. S. Reorda, T. Stefanov, I. Tuzov, S. Wagner, and N. Zidaric, “A survey of recent developments in testability, safety and security of RISC-V processors,” in *IEEE European Test Symposium, ETS 2023, Venezia, Italy, May 22-26, 2023*. IEEE, 2023, pp. 1–10. [Online]. Available: <https://doi.org/10.1109/ETS56758.2023.10174099>
- [3] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, “Keystone: an open framework for architecting trusted execution environments,” in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 38:1–38:16. [Online]. Available: <https://doi.org/10.1145/3342195.3387532>
- [4] Hex Five Security, Inc., “Multizone® security tee for risc-v processors,” <https://github.com/hex-five/multizone-sdk>, 2024, [Online]. Accessed: Sept. 14, 2025.
- [5] Zephyr Project, “RISC-V Architecture Support in Zephyr,” <https://docs.zephyrproject.org/latest/hardware/arch/risc-v.html>, 2025, [Online]. Accessed: Sept. 14, 2025.
- [6] FreeRTOS, “Memory Protection Unit (MPU) Support,” <https://www.freertos.org/Security/04-FreeRTOS-MPU-memory-protection-unit>, April 2025, [Online]. Accessed: Sept. 14, 2025.
- [7] K. Quénéhervé, W. Pensec, P. Tanguy, R. Dafali, and V. Lapôte, “Exploring fault injection attacks on CVA6 PMP configuration flow,” in *27th Euromicro Conference on Digital System Design, DSD 2024, Paris, France, August 28-30, 2024*. IEEE, 2024, pp. 43–50. [Online]. Available: <https://doi.org/10.1109/DSD64264.2024.00015>
- [8] A. Askeland, S. Nikova, and V. Nikov, “Who watches the watchers: Attacking glitch detection circuits,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 1, pp. 157–179, 2024.
- [9] K. Gomina, J.-B. Rigaud, P. Gendrier, P. Candelier, and A. Tria, “Power supply glitch attacks: Design and evaluation of detection circuits,” in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014, pp. 136–141.
- [10] H. Igarashi, Y. Shi, M. Yanagisawa, and N. Togawa, “Concurrent faulty clock detection for crypto circuits against clock glitch based DFA,” in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, May 19-23, 2013*. IEEE, 2013, pp. 1432–1435. [Online]. Available: <https://doi.org/10.1109/ISCAS.2013.6572125>
- [11] OpenHW Group, “CV32E40S User Manual - Xsecure extension,” <https://docs.openhwgroup.org/projects/cv32e40s-user-manual/en/latest/xsecure.html>, [Online]. Accessed: Sept. 14, 2025.
- [12] lowRISC Contributors, “Security Features — Ibex Reference Guide,” https://ibex-core.readthedocs.io/en/latest/03_reference/security.html, [Online]. Accessed: Sept. 14, 2025.
- [13] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, “Bypassing isolated execution on RISC-V using side-channel-assisted fault-injection and its countermeasure,” *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 1, pp. 28–68, 2022. [Online]. Available: <https://doi.org/10.46586/tches.v2022.i1.28-68>
- [14] T. Malkin, F. Standaert, and M. Yung, “A comparative cost/security analysis of fault attack countermeasures,” in *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, ser. Lecture Notes in Computer Science, L. Breveglieri, I. Koren, D. Naccache, and J. Seifert, Eds., vol. 4236. Springer, 2006, pp. 159–172. [Online]. Available: https://doi.org/10.1007/11889700_15
- [15] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*. IEEE Computer Society, 2005, pp. 243–247. [Online]. Available: <https://doi.org/10.1109/HPCA.2005.37>
- [16] M. Peña-Fernández, A. Serrano-Cases, A. Lindoso, M. García-Valderas, L. Entrena, A. Martínez-Álvarez, and S. Cuenca-Asensi, “Dual-core lockstep enhanced with redundant multithread support and control-flow error detection,” *Microelectronics Reliability*, vol. 100, p. 113447, 2019.
- [17] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 111–122, 2002. [Online]. Available: <https://doi.org/10.1109/24.994926>
- [18] D. S. Khudia and S. A. Mahlke, “Low cost control flow protection using abstract control flow signatures,” in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2013, LCTES '13, Seattle, WA, USA, June 20-21, 2013*, B. Franke and J. Xue, Eds. ACM, 2013, pp. 3–12. [Online]. Available: <https://doi.org/10.1145/2491899.2465568>
- [19] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, “Holistic control-flow protection on real-time embedded systems with kage,” in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2281–2298. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/du>
- [20] Q. Liu, L. Guo, and H. Tang, “Fault model analysis of dram under electromagnetic fault injection attack,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.
- [21] P. Calingaert, “Two-dimensional parity checking,” *Journal of the ACM (JACM)*, vol. 8, no. 2, pp. 186–200, 1961.
- [22] OpenHW Group, “OpenHW Group CORE-V CV32E40S RISC-V IP,” <https://github.com/openhwgroup/cv32e40s>, [Online]. Accessed: Sept. 14, 2025.
- [23] Digilent, Inc., “Genesys 2 Reference Manual,” <https://digilent.com/reference/programmable-logic/genesys-2/reference-manual>, [Online]. Accessed: Sept. 14, 2025.
- [24] Xilinx Inc., “Vivado Design Suite,” <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html>, [Online]. Accessed: Sept. 14, 2025.
- [25] TSMC, “Logic technology - 40nm,” https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_40nm, [Online]. Accessed: Sept. 14, 2025.
- [26] Cadence Design Systems, “Genus Synthesis Solution Datasheet,” https://www.cadence.com/en_US/home/resources/datasheets/genus-synthesis-solution-ds.html, [Online]. Accessed: Sept. 14, 2025.
- [27] J. Rabinowicz and S. Greenberg, “A new physical design flow for a selective state retention based approach,” *Journal of Low Power Electronics and Applications*, vol. 11, no. 3, p. 35, 2021.