Hardware/Software Runtime for GPSA Protection in RISC-V Embedded Cores

Louis Savary, Simon Rokicki, Steven Derrien

▶ To cite this version:

Louis Savary, Simon Rokicki, Steven Derrien. Hardware/Software Runtime for GPSA Protection in RISC-V Embedded Cores. DATE 2025 - Design, Automation and Test in Europe Conference, Mar 2025, Lyon, France. pp.1-7. hal-04788484v2

HAL Id: hal-04788484
https://hal.science/hal-04788484v2

Submitted on 18 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Hardware/Software Runtime for GPSA Protection in RISC-V Embedded Cores

Louis Savary,
Univ Rennes, Inria, CNRS, IRISA

Simon Rokicki
Univ Rennes, Inria, CNRS, IRISA

Steven Derrien
UBO, Lab-STICC

***Abstract*—State-of-the-art hardware countermeasures against fault attacks are based, among others, on control-flow and code integrity checking. Generalized Path Signature Analysis and Continuous Signature Monitoring can assert these integrity properties. However, supporting such mechanisms requires a dedicated compiler flow and does not support indirect jumps. This work proposes a technique based on a hardware/software runtime to generate those signatures while executing unmodified off-the-shelf RISC-V binaries. To the best of our knowledge, this is the first solution for providing this level of protection against fault injection on unmodified binaries. The proposed approach has been implemented on a pipelined processor, and experimental results show an average slowdown of $\times 3.35$ and an area overhead of at least $\times 1.86$ compared to unprotected implementations.**

I. Introduction

Due to their exposure to physical threats, embedded systems are particularly vulnerable to fault injection attacks. Numerous studies have shown that even well-designed cryptographic applications, considered secure in their implementation, can be compromised through such attacks. These attacks can be executed using various techniques (e.g., laser, electromagnetic interference, clock glitches, or power glitches) with the goal of causing incorrect processor behavior or leaking sensitive data [1].

Countermeasures against fault injection attacks are categorized based on the type of integrity they protect. Mechanisms may be designed to safeguard: Data integrity, ensuring the correctness of the values used in computations; Code integrity, ensuring that the instructions being executed have not been tampered with; Control-flow integrity (CFI), ensuring that the execution path of the program follows the intended flow; Control integrity, ensuring that micro-architectural control signals remain uncorrupted. These countermeasures can be implemented in software or hardware. Software-based solutions, applied during compilation, typically involve program duplication to detect faults. Hardware-based solutions, on the other hand, modify the processor's microarchitecture to detect faulty signals directly.

Among the numerous techniques for ensuring CFI, Generalized Path Signature Analysis (GPSA) and Continuous Signature Monitoring (CSM) [2] provide an effective balance between sensitivity and performance/area overhead. GPSA+CSM utilizes cryptographic signatures to verify control-flow integrity. During execution, the processor generates a dynamic signature based on previously executed instructions. Patches are applied to adjust this signature when control-flow instructions (e.g. jumps or branches) are executed, maintaining a consistent signature for each instruction. At designated checkpoints, this dynamic signature is compared with a reference to detect control-flow errors.

The ARSENE project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference "ANR-22-PECY-0004"

However, existing GPSA+CSM implementations share several key limitations: They require a custom compilation process to embed signature references and patches in the application; Indirect branches are difficult to handle without making strong assumptions about their possible targets; Handling function calls, returns, and interrupts requires saving and restoring signatures, increasing the potential attack surface.

In this paper, we introduce a method that addresses these limitations. Rather than relying on a custom compiler toolchain, our approach uses a hardware/software runtime that dynamically computes patches and signature references during the execution of unmodified RISC-V binaries. As a result, the proposed solution can transparently handle indirect branches, function calls, interrupts, and context switches.

In summary, the key contributions of this paper are:

- A hardware/software runtime that dynamically computes the necessary information for signature-based control-flow integrity on unmodified binaries;
- Support for indirect branches, function calls, interrupts, and context switches without requiring strong assumptions or custom compilation flows;
- An interrupt-driven mechanism that triggers the software routine on demand when the information needed for CFI is required;
- An extension to the processor architecture that includes components aimed at minimizing the overhead introduced by the dynamic analysis;
- An implementation of the proposed approach on the Comet processor [3], along with an experimental evaluation of its area and performance overhead.

The remainder of this paper is organized as follows: Section II provides a background on control-flow integrity techniques from the literature. Section III presents an overview of our approach, with detailed explanations of both the software and hardware components. Section IV discusses the experimental results and analyzes the performance and area overhead of the proposed solution.

II. Background and related work

In this Section, we provide all the necessary background on the signature system used in this paper, and we present previous work on control-flow integrity against fault attacks.

A. Signature system for control-flow integrity

The signature system operates on the principle that each instruction is assigned a unique signature, which is determined

	Support for: Legacy binaries	Indirect jumps	Datapath protection	Attacker model
Mafia [4]	✘	✘	Out-of-Scope	Multiple occurrence of multi-bit faults
SWIFT [5]	✘	✔	✔	Single occurrence of single-bit fault
On-line monitoring [6]	✔	✔	Out-of-Scope	No faults on first execution
CONFIDAENT [7]	✔	✘	✔	Data/Instr. corruption outside of CPU
Dual-core Lockstep [8]	✔	✔	✔	Single occurrence of single-bit fault
Proposed approach	✔	✔	Out-of-Scope	Multiple occurrence of multi-bit faults

TABLE I: Comparison of previous approach on control-flow integrity and data-protection

using a cryptographic function, denoted f, applied on the binary value of the current instruction and the signature of the preceding instruction. When dealing with control-flow instructions, patch values are applied to the signature to align it with the target instruction's signature. This is particularly important in cases of convergent control-flows, where multiple distinct execution paths lead to the same instruction. Without a patch, the signature would differ based on the execution path, causing a mismatch even when control-flow is correct. The patch values ensure that despite different paths, the signature at the target instruction converges to the correct reference signature, thus resolving this issue.

The function f exhibits several key properties: reliability, error preservation, non-associativity, and invertibility, as previously established in the literature [2, 9].

- **Reliability** ensures that any control-flow error results in a signature that deviates from the expected value.
- **Error preservation** prevents an erroneous signature from reverting to the correct value, even if the correct instruction sequence is subsequently executed.
- **Non-associativity** means that for the same set of instructions, altering their order will produce a different signature.
- **Invertibility** simplifies the process of generating patch values and calculating signatures.

A cyclic redundancy check (CRC) function is often used, as it satisfies these properties.

While the system could verify the signature after every instruction, this would impose significant overhead. Instead, due to the error preservation property of function f (see below), the dynamic signature only needs to be checked at control-flow instructions—such as jumps, branches, or returns—where deviations in flow are more likely to occur.

At runtime, when a control-flow instruction is encountered, the system compares the dynamically computed signature with the expected reference signature associated with the target instruction. If they match, execution continues as intended. If not, a control-flow error is detected, signaling potential tampering or corruption in the program's execution.

B. Control-flow integrity against fault attacks

Several approaches have been proposed to address control-flow integrity (CFI) in the context of fault attacks.

Some techniques rely entirely on software-based solutions. For example, SWIFT [5], a software-implemented hardware fault tolerance technique, uses a compiler pass to duplicate program instructions and compare register values to detect data errors. To protect against control-flow errors, static signature checks are inserted at the boundaries of basic blocks and in complex control-flow structures, ensuring that store instructions are not compromised by faults. The COMPAS compiler [10] can be used in order to protect RISC-V binaries with SWIFT. This compiler aims at realizing software modifications for Software Implemented Hardware Fault Tolerance on RISC-V applications.

Other approaches are implemented purely in hardware. Kim *et al.* [6] proposed a technique similar to GPSA and CSM signature systems, where signatures are generated during the program's first execution. However, this method assumes that the initial execution is fault-free, which is often an unrealistic assumption in the context of fault attack mitigation.

Hybrid solutions, combining both hardware and software, have also been explored. MAFIA [4] uses a compiler to generate signature references and patch values, while hardware modifications dynamically compute signatures and protect control signals throughout the execution pipeline. MAFIA also extends the GPSA implementation by introducing the concept of *pipeline state*, where control signals from the pipeline are encoded in the dynamic signature alongside the executed instructions. However, because the signature references are statically generated, MAFIA is unable to handle indirect jumps or context switches.

Beyond the control-flow integrity, several existing works tackle the problem of data integrity. In hardware-only redundancy techniques, the dual-core lockstep method by Nikiema *et al.* [8] duplicates the entire core to compare control and data signals in real time, allowing the detection of bit flips. However, this technique is limited by design to detecting only a single fault injection during execution, making it unsuitable for more complex fault injection attacks, like faults considered in Section II-C.

There are also several solutions aimed at protecting data integrity from fault injection. The work of Medwed *et al.* [11] and Fetzer *et al.* [12] are examples of data path protection techniques. These solutions use redundancy mechanisms, such as residue number systems or AN-codes, to detect multiple-bit faults within the processor's datapath. Such techniques can be implemented either entirely in software, requiring a specialized toolchain, or entirely in hardware, with corresponding changes to the micro-architecture.

Other solutions focus on defending against fault injection attacks targeting areas outside the processor. For instance, CONFIDAENT by Savry *et al.* [7] addresses fault injection in memory regions external to the processor. Their work proposes

a fault model that specifically targets external memory, and their solution involves encrypting both data and instructions to protect them from fault attacks while ensuring control-flow integrity. However, this approach requires a custom compiler toolchain and specialized micro-architecture support.

Table I summarizes the different existing techniques and their assumptions. The proposed approach tackles the limitations of previous signature-based CFI mechanisms, as it works on unmodified binaries and supports indirect jumps.

C. Threat Model

The attacker is considered to have only physical access to the device and can inject faults multiple times during execution. We consider two types of fault: overwriting up to 32 bits with a random value or overwriting up to 8 bits with a value chosen by the attacker. The attacker does not have logical access to the device. The datapath is assumed to be protected by another mechanism, as described in Section II-B, and the memory is assumed to be protected by error-correcting codes. Fault tolerance solutions aiming at Single Event Upset (SEU) model, like modular redundancy (DMT, TMR), are too weak to handle this attacker model.

III. Our approach

A. Overview

The proposed approach is designed to ensure control-flow integrity (CFI) for processors running off-the-shelf binaries. As discussed in the previous section, the GPSA+CSM mechanism offers robust protection, but current implementations rely on compile-time analysis to generate the necessary runtime data (i.e. signature references and patches). Moreover, existing methods face challenges with indirect branches, as the target address is not always known at compile-time. Function calls, returns, and interrupts also present difficulties for similar reasons.

In this work, we propose a runtime method that dynamically computes signature references whenever the CFI mechanism requires them. By introducing this dynamic analysis, our approach addresses the limitations of previous methods. However, this introduces the challenge of minimizing performance overhead. The dynamic analysis has only partial visibility of the binary being executed, requiring it to determine which sections to analyze.

To achieve this, we employ an on-demand generation mechanism. When the CFI system requires a signature reference or patch, an interrupt is triggered, invoking the GPSA generation routine. Once the required reference or patch is computed, execution resumes with all the necessary information to ensure control-flow integrity. To optimize future execution of the same instructions, the GPSA routine builds a partial control-flow graph of the running application, caching generated signature references and patches in a dedicated memory, as discussed later. More details on the GPSA generation routine are provided in Subsection III-B.

In addition to this interrupt-based mechanism, several new components are added to the processor architecture. To compute and update the dynamic signature, a CFI block with a hardware CRC operator is integrated into the processor's

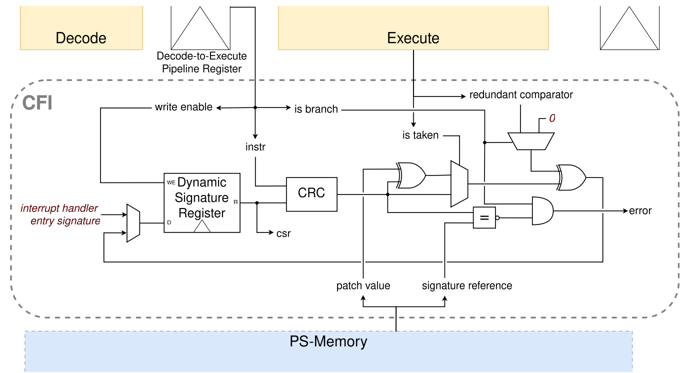


Fig. 1: CFI component scheme

Execute stage, as depicted in Figure 1. The dynamic signature is updated using its previous value, the instruction leaving the decode stage, and the patch value if the instruction involves a taken branch. This CFI block is also responsible for triggering the GPSA routine when required values are unavailable.

Furthermore, a CSI component is added to other pipeline stages to detect faults in control signals during instruction execution. This component utilizes redundant pipeline registers, which are compared against the original values to detect potential faults.

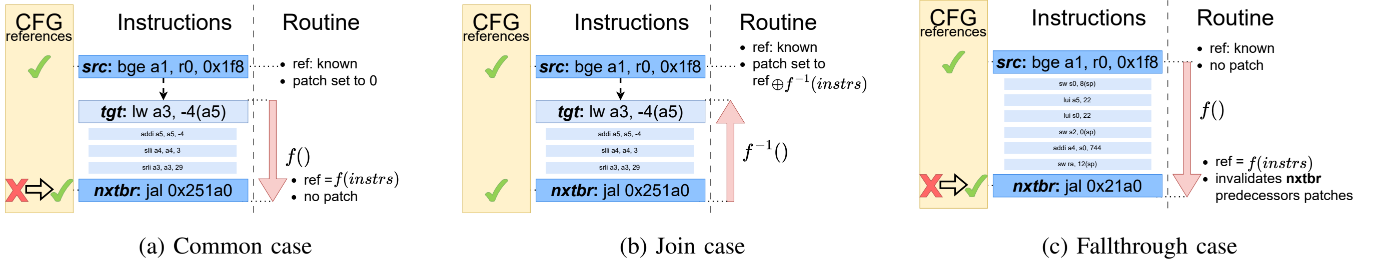
Previous approaches such as MAFIA and SWIFT [4, 5] stored signature references and patch values in the program binaries, which required additional instructions to load or to compute these values before each control-flow instruction. However, in our approach, modifying the binaries to insert extra instructions is inefficient, as it would also require updating all branch targets. To resolve this, we introduce a dedicated memory, referred to as *ps-mem*, which provides parallel access to signature references and patch values. This memory is managed as a set-associative cache by the GPSA routine, utilizing several privileged custom instructions, as detailed in Subsection III-D. Access to this memory is restricted to the CFI component or through the aforementioned custom instructions.

B. GPSA generation routine

This routine generates the signature references and the patch values and stores these values in the *ps-mem*.

The dynamic signature is only verified when executing control-flow instructions to reduce the memory cost of verification. When a control-flow instruction is executed, its corresponding values are loaded from the *ps-mem* for patching and verification. If both values are available in the *ps-mem*, the dynamic signature is verified and is patched if the branch is taken. Otherwise, if one of these values is missing, the instruction and the signature computation are canceled. An interrupt signal is raised, which triggers the routine in charge of computing the missing value.

For example, let's consider that the control-flow instruction `src` is executed by the processor while its patch value is not available in the *ps-mem*. The routine follows this scheme: First, the dynamic CFG is accessed to find the entry corresponding to `src`. An entry of the CFG contains the signature reference and the patch value, computed in a previous call to the routine.



(a) Common case

(b) Join case

(c) Fallthrough case

Fig. 2: Signature Generation Cases

As signature references are always computed ahead of the execution of corresponding instructions, the entry corresponding to `src` should always exist. A missing entry can only happen in case of erroneous control-flow.

When the entry is found in the CFG, the signature reference is stored in the *ps-mem*. If it contains a valid patch, it is also stored, and the routine ends. Otherwise, a patch value must be computed. To this end, the next control-flow instruction (`nxtbr`) must be identified. It is found by enumerating the instructions following the target instruction (`tgt`) of `src`. The `nxtbr` instruction is added to the CFG, with `src` as a predecessor. There are three different cases for computing a patch, as illustrated in Figure 2:

a) **The common case:** the `src` instruction jumps to `tgt` and executes normal instruction until the next control-flow instruction `nxtbr`. The `nxtbr` instruction does not have a signature reference yet. In this situation, a default patch is given to `src`, and a signature for `nxtbr` is computed by calling the f function on each instruction between `tgt` and `nxtbr`, then stored in the *ps-mem*.

b) **The join case:** the `src` instruction jumps to `tgt` and executes normal instruction until the next control-flow instruction `nxtbr`. The `nxtbr` instruction already has a signature reference. To get the signature of `tgt`, the f^{-1} function is applied from `nxtbr` to `tgt`. Then a patch value for `src` is computed by xoring its signature and that of `tgt`, and stored in the *ps-mem*.

c) **The fallthrough case:** the `src` instruction is a conditional branch, we have to handle the fallthrough case (i.e. when a conditional branch is not taken). After the `src`, normal instructions are executed until the next control-flow instruction `nxtbr`. As the conditional branch already uses the patch to handle the case where the branch is taken, we cannot apply a patch for the fallthrough case. Consequently the signature of `nxtbr` is directly constrained by the signature of `src`. The signature for `nextbr` is computed by calling the f function on each instruction between `src` and `nxtbr`. If `nxtbr` already have a reference signature in the CFG and/or in the *ps-mem*, this signature is updated and all the patches built based on this signature are marked as incorrect and cleared from the *ps-mem*.

Then the program execution resumes at the `src` instruction, now with the needed values available in the *ps-mem*.

C. *Protection of the routine software*

Adding complex control-flow to the execution widens the attack surface: a fault occurring during the execution of the routine has to be detected. Moreover, as the system relies on an interrupt mechanism to trigger the routine, we have to manage carefully the dynamic signature during the whole routine execution depicted in Figure 3: i) The signature has to be modified when jumping to the interrupt handler; ii) The signature is updated and checked during the execution of the routine; iii) The dynamic signature has to be restored to its initial value when the execution of the program resumes.

When an interrupt is triggered, the execution jumps to the interrupt handler which has its own signature. As an interruption can be triggered at any point of the execution, we cannot rely on a patch mechanism to obtain the entry signature of the interrupt handler. Consequently, the Dynamic Signature Register has to be writable.

In their work, Chamelot *et al.* [4] loads the entry signatures of interrupt handlers from an interrupt vector table to the Dynamic Signature Register. The Dynamic Signature Register is reset to the previous signature from a *context register* when returning from the interrupt handler. However, having a mechanism capable of overwriting the dynamic signature with different values might induce vulnerabilities. In this work, the dynamic signature register can only be overwritten with a hard-coded value, corresponding to the entry signature of the Interrupt Handler, as illustrated in Figure 1. However, overwriting the dynamic signature when interrupting might erase an erroneous dynamic signature, and lead to an undetected control-flow error. To prevent this, the overwritten dynamic signature must be saved when the interruption is triggered and restored when the program execution resumes. We use a dedicated CSR to store this value.

During the execution of the routine, control-flow instructions are executed, which require signature references and patch values. If one of these values were missing, the routine would trigger itself in an endless loop. To avoid the routine being triggered while running, all its signature references and patch values are statically computed and cannot be evicted from the *ps-mem*.

Finally, when the routine ends, the execution jumps to the interrupt terminator which is in charge of restoring the correct dynamic signature from the value stored in the CSR. As the Dynamic signature register can only be overwritten with a constant value, it uses the patch mechanism to obtain

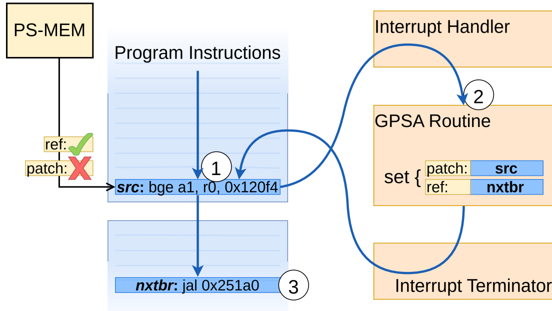


Fig. 3: Control-flow and handling of GPSA interrupts.

the correct value. The needed patch value is built by the terminator for its last instruction (i.e. the `mret` instruction used to return from an interrupt). It is built by xoring the signature stored in the CSR, and the signature reference of the `mret` instruction, which is statically known. After applying this patch, the execution resumes, restoring the exact state of the execution before the interrupt, along with eventual control-flow errors encoded in the dynamic signature.

In previous approaches relying on a compiler to generate the GPSA values, the indirect jumps cannot be handled. Here, as a routine is used to generate these values on demand, indirect jumps can be handled dynamically. The routine needs the source address and the target address, which can change between executions, to compute a correct patch value. Consequently, the target of the control-flow instruction, which is known at the end of the decode stage, is also stored in a CSR when an interrupt is triggered. Based on these two values, the routine builds a valid patch value and stores it in a part of the *ps-memory* dedicated to indirect branches, as detailed in III-D. If an indirect branch is executed while a valid *ps-mem* entry with the same source and target couple is available, its patch value can be reused.

D. Micro-architecture modifications

In addition to the CFI and CSI components used to ensure the control-flow and execution integrities, we have modified the processor micro-architecture to simplify the dynamic generation of signatures. As mentioned before, we use a dedicated memory to store the signature reference and patch values: the *ps-mem*. It prevents loading those values from memory before executing a branch. Several dedicated instructions have been defined to manage the *ps-mem* and to compute CRC and CRC^{-1} functions on 32-bit values. Additionally, the outcome of branch conditions is inserted in the signature mechanism, as depicted in Figure 1. Indeed Schilling *et al.* [13] state that branch conditions should also be encoded in dynamic signature to prevent an undetected control-flow error caused by a simple bit flip.

The *ps-mem* contains three different parts: the main memory, the private memory and the indirect jumps memory.

The main memory is a set-associative memory containing all the signature references and patch values dynamically generated by the routine. This memory is accessed by the CFI component every time a control-flow instruction is executed. The routine can write signature references and patch values in this memory using three different dedicated instructions:

- The `stw-ref` instruction is used to allocate an entry in the main memory and store a signature value. It takes two 32-bit operands: the PC value of the control-flow instruction processed by the routine and the signature value to be stored. The instruction first allocates the entry. If none is available in the given set, the least recently used value is evicted.
- The `stw-patch` instruction stores a patch in the main memory. It takes two operands: the PC and patch values. It is assumed that a `stw-ref` instruction has already allocated an entry.
- The `stw-v` instruction is used for writing the `valid` bit in a given entry. It takes the PC value and the boolean value as operands.

The private memory is a read-only memory containing the signature references and patch values for the routine control-flow. Those values are computed statically. Using a dedicated memory for the routine GPSA values prevents their eviction from the memory. As for the main memory, this memory is accessed by the CFI component every time a control-flow instruction is executed. The *ps-mem* selects the correct memory based on the PC value (the routine is placed in a known range of addresses).

Finally, the indirect jumps memory is a direct-mapped small memory containing the signature and patch values for the last indirect branch executed. This memory is accessed using the address of the indirect jumps. The tag value contains the indirect jumps' source and target addresses. Consequently, the different patches can be distinguished if a single indirect jump has multiple targets.

IV. Experimental study

The proposed approach has been implemented on the Comet RISC-V processor [3]. This processor supports rv32im RISC-V ISA, along with two 32kb caches for instructions and data. We use a 32-bit CRC function for the signature mechanism, and an AN-code redundancy system for the CSI, where the protected signals are multiplied by 5.

The benchmark application are taken from Embench-IOT [14]. The proposed software routine is compiled statically, analyzed to compute all the patch and signature values, and statically linked with the application being run. The resulting executable file is executed on the modified micro-architecture to obtain performance results. The hardware blocks are synthesized for STElectronics 28 nm FDSOI technology to measure the area overhead. Subsection IV-A presents and discuss the performance results, while Subsection IV-B is focused on the area overhead.

A. Performance overhead

In order to evaluate the impact of the proposed approach, we have executed the different benchmarks with the hardware/software runtime described in Section III. The impact of the *ps-mem* is also evaluated by varying its size and its associativity. Figure 4 shows the slowdown factors of the proposed approach, normalized over the unprotected execution of the benchmarks. Note that these results are measured after the warm-up phase used in Embench-IOT. We can see that the

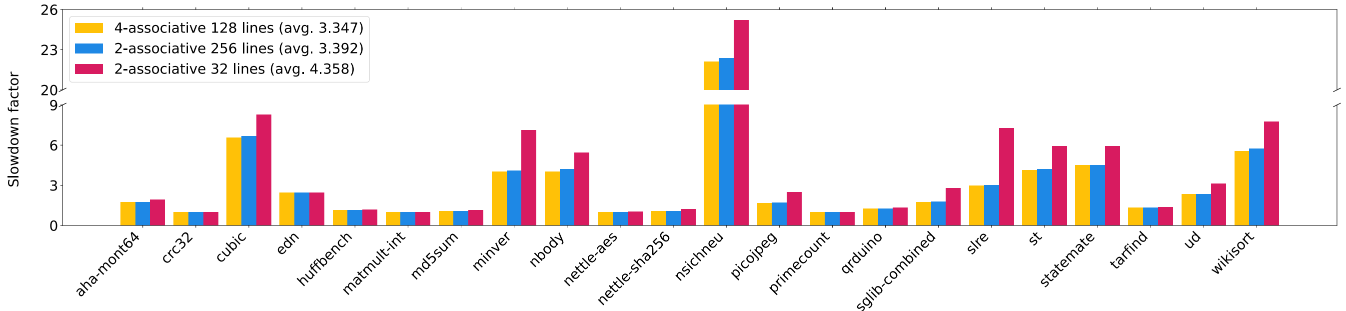


Fig. 4: Embench-IOT slowdowns, normalized over baseline Comet

performance level varies depending on the size of the *ps-mem* size. Indeed, if a value is evicted from the *ps-mem*, the routine has to be called again the next time the corresponding control-flow instruction is executed. Evictions are more frequent if the cache memory is smaller.

We can see on Figure 4 that the proposed approach induce a slowdown between $\times 1$ and $\times 25$, with an average between $\times 3.35$ and $\times 4.36$, depending on the *ps-mem* size. We can observe that most of the benchmarks have low impact on their performance, as the signatures and patches fit in the *ps-mem*. However, several benchmarks see an important slowdown because of the important number of control-flow instructions, leading to an increased number of evictions. The worst benchmark is *nsichneu* which has more than a thousand branch instructions on its code. This example if symptomatic of what could be inefficient in the proposed approach.

Existing solutions requiring compile-time modifications can have up to $128, 88\%$ execution time overhead [5, 10]. However this solution assumes a SEU fault model, which is weaker than the fault model used for the proposed approach. On the other hand, the MAFIA countermeasure [4] offers the same level of protection than the proposed approach, and causes an average slowdown of 18.4%. Still, note that the comparison is not fair as we handle a more general use case: we can handle any unmodified binaries, regardless of the presence of indirect jumps.

B. Area overhead

The modified micro-architecture has been synthesized to evaluate the area overhead of the proposed approach. As for the previous experiment, we used different sizes for the *ps-mem* main memory. The size of the private memory and structure depends on the implementations of the GPSA routine. In this experimental study, we used a 7-associative read-only memory of 32 rows. Additionally, 16 128-bit registers are used to store indirect jumps signature and patch values. Note that the memory and datapath are considered protected (as stated in Section II). The area numbers presented here does not take into account a potential overhead caused by these protections.

The obtained results show an area overhead factor between $\times 1.86$ and $\times 2.27$, depending on the size of the *ps-mem* main memory.

Compared to other solutions relying on a compiler, as MAFIA [4] which has an area overhead of 6.5%, the obtained overhead is important for the same level of protection. This is due to the need for a dedicated cache memory for storing signature and patch values. For solutions which do not rely on a compiler, as hardware duplication or triplication (DMR or TMR), these solutions have an approximate area overhead of $\times 2$ and $\times 3$. The area overheads may be equivalent between these solutions and our approach, but our protection level is higher, as DMR and TMR target simpler attacker model, usually SEU tolerance.

C. Security analysis

A first validation of the proposed protection has been performed following the methodology from previous work [15]. We have performed random fault injections of up to 32-bit random values on either the pipeline control registers, the dynamic signature register, and the program counter. All injections have either been successfully detected by our approach, or does not impact the execution of the benchmark. Further evaluation of the proposed protection needs to be conducted.

As we add components to the micro-architecture, we widen the attackable surface. However, added components are intrinsically protected, as faulting them causes the verification values to be different from the dynamic values. For example, faulting the CRC operator only impacts the signature reference and patch value computations. The fault is detected when the next verification involving either value fails. In the case of multiple faults, the cryptographic nature of f, and the imprecision of fault injection make the risk of collusion between the faulty signature and the erroneous control-flow very unlikely.

V. Conclusion and Future works

We propose a new Hardware/Software runtime aiming at protecting any RISC-V off-the-shelf program without having to recompile it. The proposed approach achieves GPSA and CSM protection thanks to computation of reference signatures during the execution of the program. The protection comes at the price of a $\times 3.35$ slowdown, for a $\times 2.27$ area increase. Future work will be focused on finding solution to handle applications where the number of branch is too large for the size of the signature memory. We also consider varying the size of the signature function to trade the security level and the area overhead.

References

[1] Johan Laurent et al. "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019, pp. 252–255. DOI: 10.23919/DATE.2019.8715158.

[2] Mario Werner, Erich Wenger, and Stefan Mangard. "Protecting the Control Flow of Embedded Processors against Fault Attacks". In: *Smart Card Research and Advanced Applications*. Ed. by Naofumi Homma and Marcel Medwed. Cham: Springer International Publishing, 2016, pp. 161–176. ISBN: 978-3-319-31271-2.

[3] Simon Rokicki et al. "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications". In: *ICCAD 2019 - 38th IEEE/ACM International Conference on Computer-Aided Design*. Westminster, CO, United States: IEEE, Nov. 2019, pp. 1–8. URL: https://hal.science/hal-02303453.

[4] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. "MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–1. DOI: 10.1109/TCAD.2023.3276507.

[5] G.A. Reis et al. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[6] Seongwoo Kim and A.K. Somani. "On-line integrity monitoring of microprocessor control logic". In: *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*. 2001, pp. 314–319. DOI: 10.1109/ICCD.2001.955045.

[7] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. "Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption". In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, pp. 246–253. DOI: 10.1109/DSD51259.2020.00048.

[8] Pegdwende Romaric Nikiema et al. "Design with low complexity fine-grained Dual Core Lock-Step (DCLS) RISC-V processors". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*. 2023, pp. 224–229. DOI: 10.1109/DSN-S58398.2023.00062.

[9] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. "SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Antwerp, Belgium: IEEE, Mar. 2022, pp. 556–559. DOI: 10.23919/DATE54114.2022.9774685.

[10] Uzair Sharif, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. "COMPAS: Compiler-assisted Software-implemented Hardware Fault Tolerance for RISC-V". In: *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 2022, pp. 1–4. DOI: 10.1109/MECO55406.2022.9797144.

[11] Marcel Medwed and Stefan Mangard. "Arithmetic logic units with high error detection rates to counteract fault attacks". In: *2011 Design, Automation & Test in Europe*. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763261.

[12] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware". In: *Computer Safety, Reliability, and Security*. Ed. by Bettina Buth, Gerd Rabe, and Till Seyfarth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 283–296. ISBN: 978-3-642-04468-7.

[13] Robert Schilling, Mario Werner, and Stefan Mangard. "Securing conditional branches in the presence of fault attacks". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 1586–1591. DOI: 10.23919/DATE.2018.8342268.

[14] David Patterson et al. *Embench: Open Benchmarks for Embedded Platforms*. https://github.com/embench/embench-iot.

[15] Joseph Paturel, Angeliki Kritikakou, and Olivier Sentieys. "Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 328–333. DOI: 10.1109/ISVLSI49217.2020.00067.