Ahead of Time Generation for GPSA Protection in RISC-V Embedded Cores

Louis Savary, Simon Rokicki, Steven Derrien

▶ To cite this version:

Louis Savary, Simon Rokicki, Steven Derrien. Ahead of Time Generation for GPSA Protection in RISC-V Embedded Cores. ASAP 2025 - 36th IEEE International Conference on Application-specific Systems, Architectures and Processors, Jul 2025, Vancouver (BC), Canada. pp.1-7. ⟨hal-05100014v2⟩

HAL Id: hal-05100014
https://hal.science/hal-05100014v2

Submitted on 25 Aug 2025

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 CC BY 4.0 - Attribution - International License

Ahead of Time Generation 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, many implementations of such mechanisms require a dedicated compiler flow and do not support indirect jumps, while others have prohibitive overheads. This work proposes a technique based on a ahead-of-time analysis to generate those signatures, associated with a hardware/software runtime handling indirect jumps while executing unmodified off-the-shelf RISC-V binaries. The proposed approach has been implemented on a pipelined processor, and experimental results show an average slowdown of $\times 1.82$ and an area overhead of at least $\times 1.3$ compared to unprotected implementations.**

I. Introduction

Because embedded systems are frequently exposed to physical threats, they are particularly susceptible to fault injection attacks. Numerous studies have demonstrated that even cryptographic applications considered secure in their design can be compromised through fault injection. These attacks can be carried out through various techniques (laser, electromagnetic interference, clock glitches, power glitches, etc.), aiming to induce processor malfunctions or leak sensitive information [1].

Countermeasures against fault injection attacks generally focus on preserving certain types of integrity. Mechanisms can be devised to protect:

- Data integrity, ensuring the accuracy of values used during computations;
- Code integrity, ensuring that executed instructions are not tampered with;
- Control-flow integrity (CFI), ensuring that the program follows its intended execution path;
- Control integrity, preserving the consistency of microarchitectural control signals.

These mechanisms may be implemented in software—often by duplicating program code at compilation time to detect faults—or in hardware, modifying the processor's microarchitecture to identify erroneous signals directly.

Among the many approaches to CFI, Generalized Path Signature Analysis (GPSA) and Continuous Signature Monitoring (CSM) [2] strike a compelling balance between sensitivity to faults and performance/area overhead. GPSA+CSM uses cryptographic signatures to verify control-flow conformance: during execution, the processor maintains a dynamic signature reflecting the instructions already executed. The signature is updated at each control-flow instruction (e.g., jumps or branches) to ensure consistency between the expected and actual instruction. The dynamic signature is compared against a reference to detect any anomaly at predefined checkpoints.

However, current implementations of GPSA+CSM face several significant limitations:

- They require a specialized compilation process to embed signature references and patches into the application;
- Indirect branches are difficult to handle without strong assumptions regarding their potential targets;
- Managing function calls, returns, and interrupts requires saving and restoring the signature, thereby expanding the system's attack surface.

Recent work by Savary *et al.* [3], hereinafter referred to as PS-Mem, proposes to address these issues by performing a dynamic binary analysis that reconstructs signatures and patches on demand whenever a control-flow instruction is executed. While this approach resolves the previously mentioned shortcomings, it also leads to significant performance overhead and increases the processor's area requirements.

In this paper, we introduce a method that addresses these limitations while keeping the performance overhead acceptable. Rather than relying on a compile-time analysis, our approach uses an ahead-of-time binary analysis in charge of generating most of the GPSA information, along with a runtime mechanism in charge of dynamically crafting the missing 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 and has a lower impact on performance than previous approaches.

In summary, the key contributions of this paper are:

- An ahead-of-time binary analysis in charge of building the control-flow graph and most of the signature reference and patch values needed for signature-based control-flow integrity on unmodified binaries;
- A runtime mechanism that handles efficiently all the direct control-flow instructions combined with a software routine in charge of handling indirect control-flow operations;
- An implementation of the proposed approach on the Comet processor [4], 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 the ahead-of-time

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-000

analysis and the runtime behavior. 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

This subsection describes the signature-based approach, in which each instruction is assigned a distinct signature computed by a cryptographic function, denoted f. This function takes as input the control signals corresponding to the current instruction and the signature produced by the preceding instruction.

For control-flow instructions, patch values are applied to align the signature with that of the target instruction. This mechanism is particularly important in convergent control flows, where multiple paths lead to the same instruction. Without patch values, the signature would differ depending on the path taken, possibly causing a mismatch even if the control flow is correct. By applying these patches, the dynamic signature is changed according to the target reference signature.

The function f exhibits several critical properties, as documented in prior works [2, 10]:

- **Reliability** Ensures that any control-flow deviation causes the signature to differ from its expected value.
- **Error preservation** Prevents a faulty signature from reverting to the correct one, even if subsequent instructions are executed correctly.
- **Non-associativity** Ensures that rearranging the same sequence of instructions yields a different signature.
- **Invertibility** Simplifies the generation of patch values and the computation of signatures.

A cyclic redundancy check (CRC) function is often used for f, given that it satisfies these properties.

Although it would be possible to verify the signature after each instruction, such a method would introduce substantial overhead. Instead, the error preservation property of f allows verification of the dynamic signature only on control-flow instructions where deviations in execution are more probable.

When a control-flow instruction is encountered during runtime, the system compares the dynamically computed signature with the reference signature associated with the target instruction. If they match, execution proceeds normally. Otherwise, a control-flow error indicates potential tampering or corruption in the program flow.

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 faults do not compromise store instructions. The COMPAS compiler [11] can be used 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 [7] 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 instead of executed instructions, to more easily assert the execution integrity. However, because the signature references are statically generated, MAFIA compiler has to remove the indirect jumps.

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.* [12] and Fetzer *et al.* [13] 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.* [9] 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.

Fault injection can also target other parts of the embedded device, such as the memory where the program is stored, as shown by Dutertre *et al.* [14]. Previous works also studied

TABLE I: Comparison of the previous approach on control-flow integrity, code protection and data protection. COTS binaries indicates whether the approach protects unmodified binaries. IJ indicates whether the approach supports indirect jumps.

	Support for:				Attacker model
	COTS binaries	**IJ**	**Datapath protec.**	**Code protec.**	
SWIFT [5]	✘	✔	✔	✘	Single occurrence of single-bit fault
On-line monitoring [6]	✔	✔	Out-of-Scope	✘	No faults on first execution
MAFIA [7]	✘	✘	Out-of-Scope	Integrity	Multiple occurrences of multi-bit faults
Dual-core Lockstep [8]	✔	✔	✔	✘	Single occurrence of single-bit fault
CONFIDAENT [9]	✔	✘	✔	Confidentiality	Data/Instr. corruption outside of CPU
PS-Mem [3]	✔	✔	Out-of-Scope	✘	Multiple occurrences of multi-bit faults
Our approach	✔	✔	Out-of-Scope	Integrity	Multiple occurrences of multi-bit faults

the problem of code protection. For example, code integrity ensures that the code stored in memory remains unmodified, code authenticity verifies that the code originates from a trusted source, and code confidentiality prevents unauthorized disclosure of the code's contents.

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. Attacker 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 model, like modular redundancy, are too weak to handle this attacker model.

III. Proposed approach

The objective of the proposed approach is to protect COTS binaries with a reduced runtime overhead compared with existing work. This is achieved through i) an ahead-of-time binary analysis in charge of building signature references and patches and ii) a runtime in charge of verifying the dynamic signature and handling events that cannot be handled statically (e.g., indirect jumps, context switches).

A. Ahead-of-time binary analysis

The proposed ahead-of-time analysis is in charge of generating most of the reference signature and patch values which will be used to ensure the control-flow integrity at runtime. As the analysis is performed at install time (i.e., when the device is flashed with the compiled software), the analysis only has access to the binaries.

As illustrated in Figure 1, the first step is to go through the different instructions in order to identify every control-flow instruction and their target when it is known statically. Using this information, the analysis applies the signature function f to build the reference signature for each instruction of the binaries, storing only the one related to the control flow instructions. The control-flow graph is also used to compute the patch values, by xoring the signature references of the source and destination of each control-flow instruction.

The information computed by this analysis is used at runtime to monitor that the dynamic signature is always coherent with the one computed ahead of time. Consequently, we carefully choose the structure of the control flow graph (CFG) in memory to limit the impact on the performance of runtime monitoring. The entry for each control-flow instruction is stored according to its PC value, along with an offset used to retrieve the entry corresponding to the destination of the branch instruction. Then, for each entry, the signature reference and patch value are stored. Each entry in the control-flow graph needs 16 bytes in memory. Note that patch values and offset for indirect branches cannot be determined statically and are left blank in memory.

As shown by Schilling *et al.* [15], the outcome of a conditional branch is a single-bit signal that can easily be faulted. As this signal drives both the next instruction executed and the next dynamic signature, it will not be seen by the signature monitoring system. Consequently, the comparator for conditional branches returns a hashed value representing the outcome of the conditional branch, and this value is integrated in the dynamic signature, as shown in Figure 3. During the ahead-of-time analysis, this additional modification of the signature for conditional branches is taken into account to construct the signature references.

B. Runtime signature monitoring

The runtime component in the proposed approach is in charge of monitoring the dynamic signature and handling dynamic events through the construction of a correct patch. The logic of the runtime component is illustrated in the Figure 3.

1) Signature monitoring: During the program execution, control signals of each executed instruction are used to update the dynamic signature through the signature function f. A dedicated register (labeled `Pointer Register` on Figure 3) is used to easily retrieve the GPSA information computed by the ahead-of-time analysis. This register always contains the address of the entry containing the GPSA information for the next control-flow being executed.

Every time a control-flow instruction is executed, its corresponding GPSA values are loaded from the address stored in `Pointer Register`. The dynamic signature is compared against the signature reference to ensure the control-flow

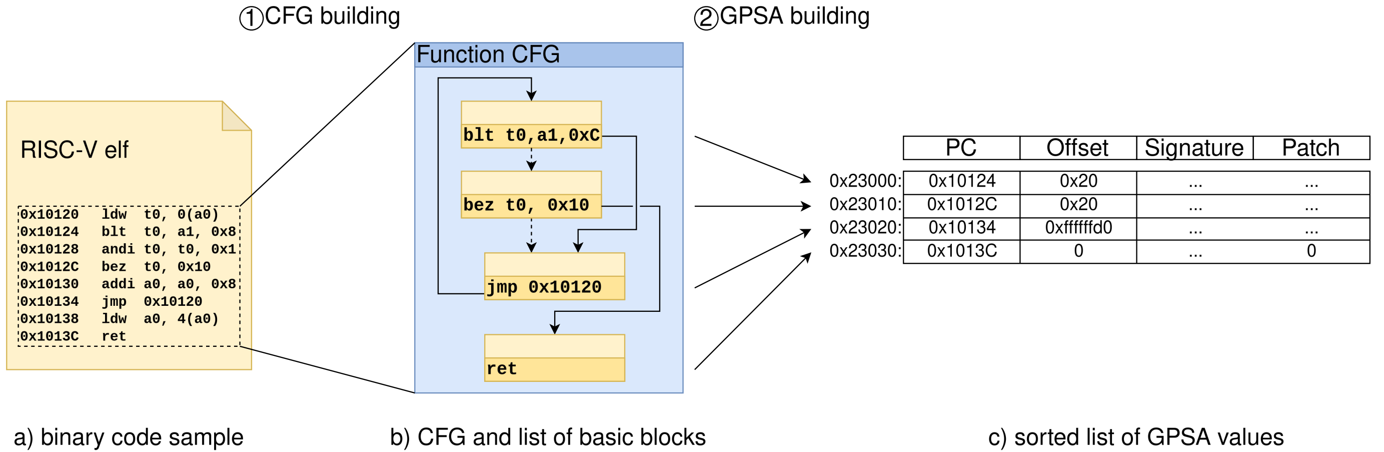


Fig. 1: Ahead-of-Time GPSA Analysis

integrity. Then, if the branch is taken, the patch is applied to the newly computed signature. The `Pointer Register` is updated to point to the next branch being executed: if the branch is taken, we add the offset stored in the CFG; if the branch is not taken, we simply move to the next entry in the CFG. After the execution of the control-flow instruction, the value of the dynamic signature is correct and the `Pointer Register` is ready for the next control-flow instruction.

2) Handling dynamic events: During the program execution, several events that are difficult to predict at deploy time can occur: indirect jumps toward unknown locations, context switch to execute another process or handling an interrupt.

Similarly to previous work, we choose to forbid the direct write of an arbitrary value in the register storing the dynamic signature [3]. Indeed, an instruction capable of writing this register would increase the attack surface. Instead, as illustrated in Figure 3, this register can only be updated with two values:

- the new signature, calculated by applying the signature function f to the previous value of the dynamic signature, and xored with the currently executed instruction;
- a constant, hardcoded value when an interrupt is triggered.

All other modifications of the dynamic signature have to be performed through the application of a well-crafted patch value.

When an indirect jump is executed, the runtime component is in charge of building a patch value that transforms the reference signature of the source instruction into the reference signature of the destination instruction.

More precisely, Figure 2 depicts the different steps occurring in such a situation. When an indirect jump `src` is executed toward an address `target`, the CFI component detects that the `offset` and the `patch` are null (step 1 in Figure 2), and an interruption is raised. The current `pc` value, the destination address, and the dynamic signature are stored in the dedicated Control and State Registers (CSRs) of the processor. At the same time, the dynamic signature register is set at the constant value, and the PC jumps to the interrupt handler (step 2). By performing binary searches in the sorted list of GPSA values, we find the CFG entry corresponding to the `src` instruction (step 3) and the entry which is right after target, which we call `nxtbr` (step 4). If `nxtbr` is different to `target`, we load the signature reference of `nxtbr`, and apply the reversed signature function f^{-1} to determine the reference signature of `target`. Once we know both the reference signature for `src` and for `target`, we can craft the desired patch by xoring those two values. The offset for `src` is determined by the difference between the adresses of the CFG lines of the `nxtbr` and `src` ($@2-@1$, using names from Figure 2). These patch and offset are then stored in the entry corresponding to the indirect jump (step 5).

When the patch is properly crafted, the execution goes to the interrupt terminator, which is in charge of restoring register values and returning the program execution with a correct dynamic signature (step 6). As the runtime code is known at compile time, it is possible to determine the reference signature of the last instruction of the interrupt terminator (i.e., the instruction `MRET` for returning from an interrupt). Knowing this reference, we can craft a patch that transforms this statically known signature into the reference signature that was stored in a CSR when the instruction was triggered. By executing the `MRET` instruction, the execution resumes to the indirect branch that triggered the interrupt with the same dynamic signature as before. But this time, a patch value is available for handling the indirect jump correctly. When reading this value in the data cache, a special flag is set to nullify the patch so that the next execution of the indirect branch triggers an interrupt.

It is important to note that during the execution of this software analysis, the dynamic signature is always updated using the pseudo-cryptographic function and verified whenever a branch is executed. Every reference signature for this part has been computed at compile time, using the hardcoded signature value of the interrupt handler as a starting point.

Handling a context switch is similar to handling an indirect jump, but the values for returning from interrupt are loaded from the memory.

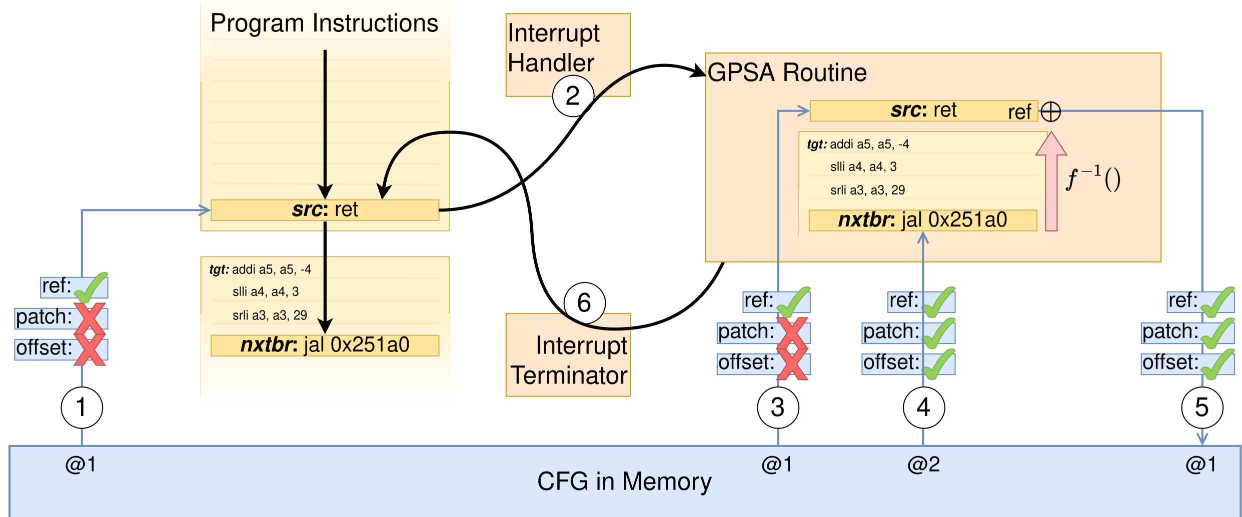


Fig. 2: GPSA routine execution scheme. Black arrows represent the execution path through the executed program and the GPSA routine. Blue arrows represent the CFG entries loaded/stored at different addresses. The different steps are labelled between 1 and 6.

To reduce the performance overhead due to indirect branches, we added a set of registers in the micro-architecture to store the last `src`, `dest` values seen, along with their reference signature and patch values. When an indirect jump is executed, if the correct value is stored in one of these registers, the execution can continue without going through the entire runtime software.

3) Hardware modifications: All the above-mentioned mechanisms require to modify the processor implementation. The CFI component has been implemented as described in Figure 3. The component in capable of raising an interrupt if needed, and to detect invalid signatures. The mechanism for protecting conditional branch has been implemented as described by Schilling *et al.* [15]. The cache memory has been modified in order to allow reading 16 bytes at a time in order to load the full CFG information in a single cycle. Consequently the CFI component can load its data in a single cycle. Note that access conflicts to the data cache are possible. In such a case, the memory operation from the Mem pipeline stage is executed first, stalling the first stages of the pipeline. Then the memory load initiated by the CFI component is executed.

The chosen signature function f is a CRC32 function. The choice of the CRC32 polynome is based on the work of Chamelot *et al.* [7]. In order to reduce the time required to craft a signature, special instructions are added in the ISA to compute f and f^-1, as done in the implementation by Savary et al. [3]. These instruction reduce the time for the ahead-of-time analysis and for handling dynamic events.

Finaly, several registers have been added in the CFI component to store recently used CFG entries and reducing the cache pressure. Similarly, several registers are used to store the CFG information computed for an indirect jump. If the same jump is executed, and if the destination is the same, the information can be re-used transparently. The impact of those registers on the performance overhead is evaluated in Section IV.

IV. Experimental study

To be able to compare our approach to SotA solutions, we implemented it on Comet, a RISC-V pipelined core simulator [4]. The simulated processor is a 5-stages `rv32i` pipeline with two 4-associative caches of 64 lines, for data and instructions. This implementation can simulate the execution of RISC-V programs, cycle accurately, to measure time overhead compared to base core. Thanks to the C++ implementation of the core, the area overhead of the solution can be estimated through High-Level Synthesis. Because our approach stores additional data in memory, the memory overhead must be compared to these of SotA solutions.

In addition, to assert the security of our solution, a fault injection simulation campaign has been run.

A. Performance overhead

The proposed approach has been implemented on the RISC-V pipeline simulator Comet [4], and the Embench-IOT benchmark [16] has been run. The Comet simulator is cycle-accurate, such that the performance cost of our solution on the benchmarks can be compared against the SotA solution.

In Figure 4a, the slowdown factors are shown for multiple solutions. These slowdowns are measured in additional execution cycles after a warm-up pass. The first columns represent the slowdowns of the solution from Savary *et al.* [3]. They proposed multiple sizes for the GPSA values memory (PS-Mem); we used the one with the best results, that is, a 4-associative memory of 128 lines. The second and third columns represent the slowdowns for the proposed approach, Ahead of Time GPSA, the last one benefiting from the additional registers that store the recently used GPSA values.

The slowdown factors of the PS-Mem approach go from $\times 1.0$ to $\times 5.2$, to a mean of $\times 1.8$. In some cases, the proposed approach without registers achieves greater slowdowns

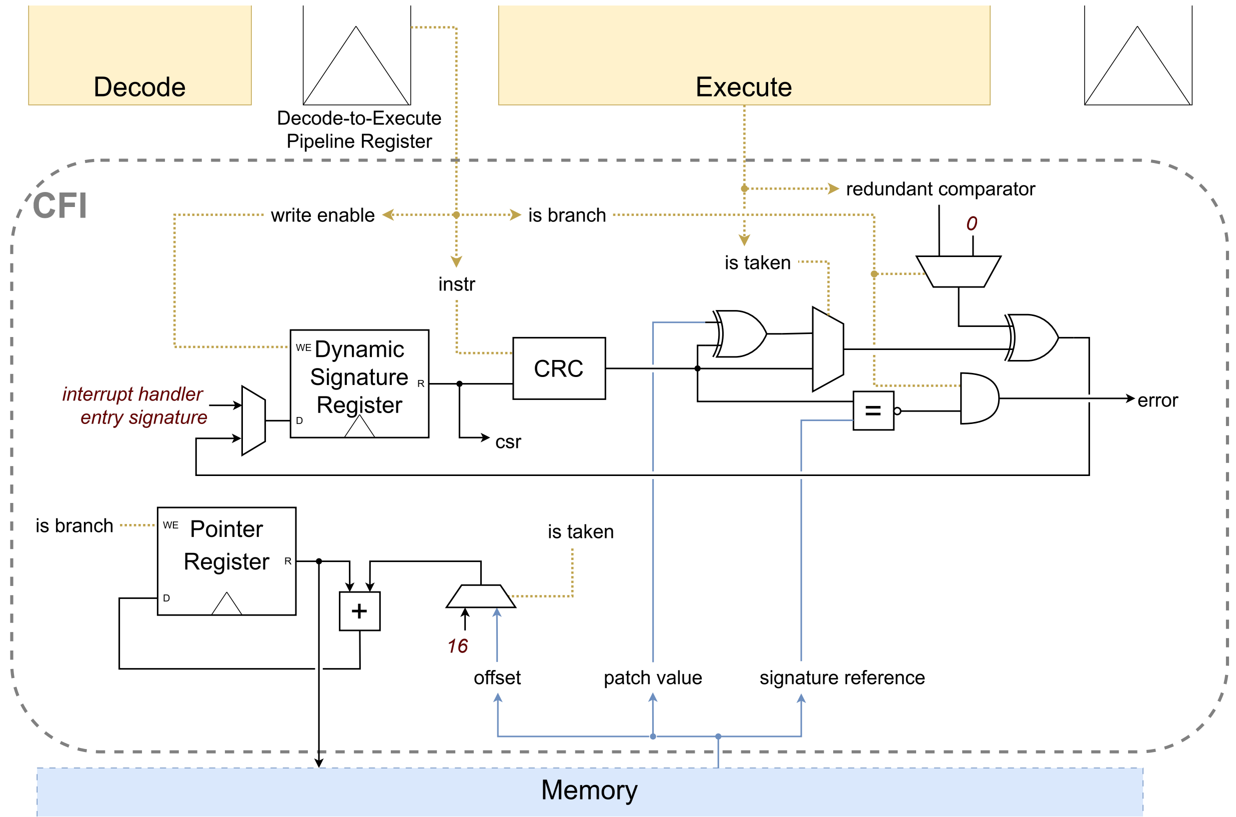
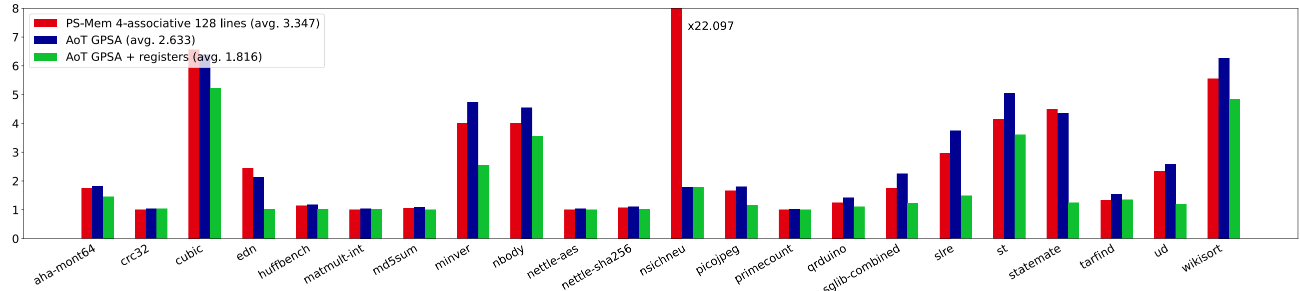
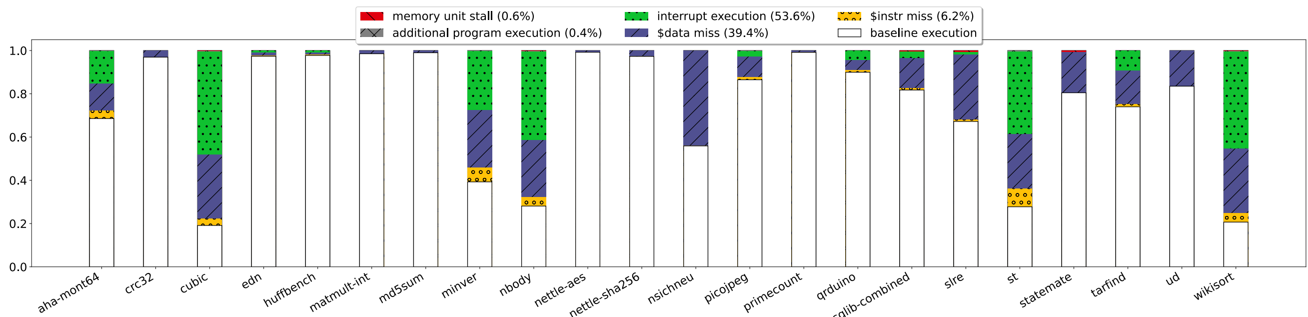


Fig. 3: CFI component scheme



(a) Embench-IOT slowdowns, normalized over baseline Comet



(b) Embench-IOT performance overhead details

Fig. 4: Performance slowdown analysis

TABLE II: Area and Memory Overhead of SotA solutions

	Area overhead	Code size overhead	Performance overhead
SWIFT [5]	0	123%	-
On-line monitoring [6]	-	-	0
MAFIA [7]	6.5%	29.4%	18.4%
Dual-core Lockstep [8]	169%	0	0
CONFIDAENT [9]	-	24%	167%
PS-Mem [3]	124%	-	235%
AoT GPSA	12.9%	26.0%	163%
AoT GPSA + registers	29.9%	26.0%	81.6%

than the PS-Mem. These benchmarks are considered to have numerous indirect jumps, which is the weaker spot of our approach regarding performance cost.

Figure 4b shows the details of the execution with the *AoT GPSA + register* solution. The white part represents the cost of the baseline execution, while each overhead is divided into categories: the overhead due to the routine handling dynamic events (labeled `interrupt execution` on Figure 4b); the increased number of data or instruction cache miss due to the use of GPSA values or from the execution of the above-mentioned routine (labeled `$instr miss` and `$data miss`); the stalls due to access conflicts between a memory operation and a control flow instruction requiring a GPSA value (labeled `memory unit stall`); the additional execution cycles due to the pipeline flush occurring when an interrupt occurs (labeled `additionnal program execution`).

We can observe in Figure 4b that most of the important performance degradation is due to the execution of the GPSA routine. Because benchmarks like *cubic* or *wikisort* have a lot of function calls, and so `return` instruction, they have an important performance degradation. We can also observe that the instruction cache miss number increases compared to the baseline execution. This is due to the cache pollution caused by the instructions of the software routine. Instruction cache misses, as well as instruction re-executions, are directly linked to the indirect jumps handling mechanism.

The second most important overhead source is the data cache misses. As the protection mechanism requires data cache lines to verify and update the dynamic signature, it adds stress to this cache. In addition, the benchmarks with a slowdown factor close to 1 in Figure 4a show no or few cycles from other sources than data cache misses. We can conclude that the handling of dynamic events is the most expensive part of the proposed approach.

In general, we can see that our approach achieves better performance than the approach of Savary *et al.* [3]. However, our solution can still be more expensive when executing programs with many indirect jumps.

B. Area overhead

Since our approach requires microarchitecture modifications, we evaluate the cost of these additions. To obtain surface figures, we used the Mentor Catapult HLS tool on the Comet processor [4]. By its conception, using an HLS tool on the Comet processor results in a circuit corresponding to the simulated core, which surface can be estimated.

In the Table II are gathered overheads from different SotA solutions. The area overhead of our solution represents an increase of 29.9% of the surface of the core compared to the baseline Comet core.

The area overhead of our solution comes from multiples sources. To ensure control flow and execution integrities, the CFI component is added to the core. This mechanism needs a lot of access to the data memory. Hence, cache registers are also added to the CFI component in order to gain from temporal locality. A CRC operator is added to the arithmetic and logic unit (ALU) to reduce the performance cost of the GPSA routine.

C. Memory overhead

As our approach computes GPSA values before the execution, they need to be stored somewhere. Since we store these values in the program data section, we need to measure the overhead in the code size of the file, in order to be able to compare our approach to other solutions.

In our approach, the memory overhead causes are twofold. One is the GPSA values for the program, depending on the program code itself. It corresponds to four 32 bits values, i.e. 16 bytes, for each control flow instruction in the executable section of the program. In the Embench-IOT benchmarks, it represents an average overhead of 20.4%.

The other source is the routine code and its GPSA values. This cost is constant and unique in the whole system. It represents 2060 bytes for the routine code, and 21 GPSA entries for 336 bytes for a total of 2396 bytes.

Our approach has an average memory overhead of 26.0% over the Embench-IOT benchmarks.

D. Security analysis

In order to assert the security of our approach, a fault injection campaign has been run. According to the attacker model, the faults being injected are of two sorts: either randomizing a value up to 32 bits, or writing a specific value of up to 8 bits, both in the processor logic, outside of the datapath. However, only the 32 bits random value injection has been tested here.

Since no physical implementation of our approach has been made yet, the faults were injected into simulations of the processor. To inject faults, injection points are created with an execution cycle number, a register of the pipeline logic and a xor mask. Then, a benchmark is run for each injection point. During its execution, the xor mask is applied to the named register at the given cycle.

To conduct the security analysis, a million injection points have been generated and simulated. The location, cycle and mask of the injection points are chosen randomly. The results show that no injection went effective and undetected. However, 5.56% of the faults crashed the simulator, by causing a load outside of the memory for example. These injected faults could have been detected or not, but the device crashing is assumed not to be the objective of the attacker. 13.95% of the faults were detected, regardless of whether the run

crashed afterward or not, or the effectiveness of the fault. The 80.49% remaining were ineffective undetected non-crashing faults, tempering with canceled instructions, for example.

This fault injection campaign shows that all the effective faults have been detected by the proposed approach. The multiple faults model, as well as specific value overwriting, have yet to be tested.

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 the computation of most reference signatures ahead of time and an interruption mechanism for dynamic events such as indirect jumps. The protection comes at the price of a $\times 1.82$ slowdown for a $\times 1.30$ area increase. Future work will be focused on reducing the overhead of the GPSA interruption. A deeper fault injection campaign is also needed, with a more effective way to generate random injection points, an adversarial and multiple faults in a test.

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] Louis Savary, Simon Rokicki, and Steven Derrien. "Hardware/Software Runtime for GPSA Protection in RISC-V Embedded Cores". In: *DATE 2025*. Lyon, France, Mar. 2025.

[4] 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.

[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] 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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Jean-Max Dutertre et al. "Experimental Analysis of the Laser-Induced Instruction Skip Fault Model". In: *The 24th Nordic Conference on Secure IT Systems, Nordsec 2019*. Aalborg, Denmark, Nov. 2019, pp. 221–237. DOI: 10.1007/978-3-030-35055-0_14.

[15] 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.

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