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

Louis Savary* Simon Rokicki Steven Derrien

Univ Rennes, Inria, ENS Rennes, IRISA

Abstract

State-of-the-art hardware countermeasures against fault attacks are based, among others, on control flow and code integrity checking. These integrities can be asserted by Generalized Path Signature Analysis and Continuous Signature Monitoring. However, supporting such mechanisms requires a dedicated compiler flow and does not support indirect jumps. In this work we propose a technique based on a hardware/software runtime to generate those signatures while executing unmodified COTS RISC-V binaries. The proposed approach has been implemented on a pipelined rv32i processor, and experimental results show an average slowdown of $\times 1.82$ compared to unprotected implementations while being completely compiler independent.

Introduction

Because of their nature, embedded systems are prone to physical attacks. Several works have demonstrated that a well-designed cryptographic application, whose implementation is considered safe, can be compromised with fault injection attacks (eg., laser, EM, clock or power glitch), which induce an incorrect behavior of the victim processor or a data leak [1].

Countermeasures against such faults can be implemented both in software or in hardware. Software countermeasures, often inserted at compile time, consist of duplicating part of the instructions to detect and counter fault injections. This type of countermeasures has reached its limits with the emergence of attacker models allowing for several faults happening in a single execution. On the other hand, hardware countermeasures rely on a modified processor microarchitecture which ensures some form of Control Flow Integrity (CFI) and code integrity. Among the numerous existing techniques, Generalized Path Signature Analysis (GPSA) and Continuous Signature Monitoring (CSM) [3] happen to provide the best trade-off between sensitivity and area/performance overhead.

GPSA/CSM relies on cryptographic signatures to ensure integrity. Throughout the execution, the processor computes a signature based on previously executed instructions. The dynamic signature is verified against a reference signature at each branch and patches are used to correct the signature when executing branches. Additional instructions are therefore needed to load signatures during the execution. Besides, patches and reference signatures must be computed ahead of time and inserted in the executable.

In GPSA/CSM, the processor datapath is considered as being protected against faults, for example through error-detecting codes in both pipeline stage registers and data/code memory.

This technique is implemented in The SCI-FI RISC-V core [4], along with an additional mechanism that protects pipeline control signals through some form of redundancy.

SCI-FI and other existing approaches share common limitations: i) the target application needs a custom compilation flow to embed signature and patches; ii) indirect branches cannot be handled without strong assumptions on the possible targets; iii) function calls, returns, and interrupts require to store/restore signatures which increases attack surface.

Previous work [2] overcomes these limitations with a runtime environment for the generation of GPSA values, relying on an interrupt mechanism to handle the GPSA values. This solution comes with a high cost, in both time and area.

In this paper, we present a method to mitigate the high overheads induced by the method of Savary *et al.* [2]. Our approach relies not only on a runtime environment, but also on a GPSA value generation when deploying the program. Our runtime also transparently handles indirect branches, function calls, interrupts, and context switches.

We have designed a proof of concept implementation based on the Comet RISC-V processor [5]. In our implementation, the pipeline is modified to check signatures on control flow instructions and trigger an interrupt to update patches and signatures whenever an indirect jump with missing signature is executed.

Our approach has been validated through fault injection simulations to ensure that protection was effective. The experimental study also shows that the average performance slowdown factor due to dynamic analysis is $\times 1.82$.

*Corresponding author: louis.savary@inria.fr

[0] 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

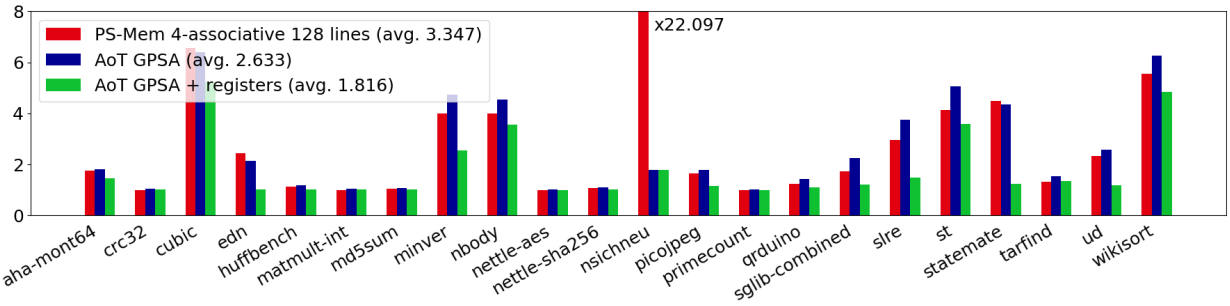

Figure 1: *Runime overhead on Embench-IoT. The first column corresponds to [2] with a ps-mem main memory of 128 lines, 4-associative.* `GPSA deploy` *corresponds to the solution presented in this paper.* `registers` *correspond to two sets of 16 128bits register for indirect jumps, and other instructions, gpsa values.*

Ahead of Time Analysis for GPSA

In order to apply GPSA protection on COTS binaries, but with less overheads than existing approaches, we propose to compute the GPSA values ahead of time. These values are then stored in data memory.

During execution, we need the hardware to easily access the GPSA values of the executed instruction. To do this, these values are stored with the following structure: a list of tuple, each corresponding to a control flow instruction, sorted by PC. To ease the data cache fetching in memory, the tuples addresses are align to the data cache line size. A register is also added to the core, pointing to the values of the next control flow instruction to be executed.

To browse the list of tuple in constant time, it is sorted by PC and a fourth value is contained in the tuples: the address offset. In the tuple corresponding to an instruction a, the offset is the difference between the address of this tuple and the address of the tuple corresponding to the control flow instruction following the target of the instruction a. With this structure, when the CCFI component processes a control flow instruction, it loads the data cache line containing the values of this instruction. With these values, it verifies the dynamic signature. If the branch is taken, the signature is updated and the address of the tuple corresponding to the next instruction is obtained by adding the offset to the current tuple address. Otherwise, the corresponding tuple is the following tuple in the list, because it is sorted by PC.

Concerning patches for indirect jumps, as their targets cannot be known ahead of time, their computation is left to an interrupt mechanism, similar to the one from Savary *et al.* [2].

Experimental study

We implemented our solution on the Comet RISC-V processor. The overall area overhead has been evaluated thanks to an HLS tool and is presented in the Table 1.

Core	area(μm^2)	overhead
PS-Mem [2]	150311	126.6%
AoT GPSA	74856	12.9%
AoT GPSA + registers	86130	29.9%

Table 1: *Area overhead of different solutions. PS-Mem refers to solution from [2] with a 4-associative 128 lines main memory. AoT GPSA is the solution presented in this paper. Registers represents two sets of 16 128bits registers for GPSA values.*

The figure 1 shows the slowdown between the previous solution from Savary *et al.* [2] and our solution on the Embench-IoT benchmarks [6], normalized on the performances of an unmodified Comet. Results show a slow-down factor between 1.0 and 5.23, with an average of $\times 1.82$.

Conclusion

In this paper, we propose a method to apply GPSA and CSM protections on unmodified binaries, with an average runtime overhead of $\times 1.82$ and area overhead of 30%. As far as we know, this is the best hardware/software implementation for GPSA without compiler dependence and allowing integrity properties to hold while handling indirect jumps, function calls, interrupts as well as context switches.

References

[1] J. Laurent et al. "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures". In: *DATE'19*.

[2] L. Savary et al. "Hardware/Software Runtime for GPSA Protection in RISC-V Embedded Cores". In: *DATE'25*.

[3] M. Werner et al. "Protecting the Control Flow of Embedded Processors against Fault Attacks". In: *Smart Card Research and Advanced Applications*. Springer International, 2016.

[4] T. Chamelot et al. "SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks". In: *DATE'22*. IEEE.

[5] S. Rokicki et al. "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications". In: *ICCAD 2019*. IEEE.

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