SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks

Thomas Chamelot
Univ. Grenoble Alpes, CEA, List,
F-38000 Grenoble, France
thomas.chamelot@cea.fr

Damien Couroussé
Univ. Grenoble Alpes, CEA, List,
F-38000 Grenoble, France
damien.courousse@cea.fr

Karine Heydemann
Sorbonne Université, CNRS, LIP6,
F-75005 Paris, France
karine.heydemann@lip6.fr

***Abstract*—Fault injection attacks have become a serious threat against embedded systems. Recently, Laurent et al. have reported that some faults inside the microarchitecture escape all typical software fault models and so software counter-measures. Moreover, state-of-the-art counter-measures, hardware-only or with hardware support, do not consider the integrity of microarchitectural control signals that are the target of these faults.**

We present SCI-FI, a counter-measure for Control Signal, Code, and Control-Flow Integrity against Fault Injection attacks. SCI-FI combines the protection of pipeline control signals with a fine-grained code and control-flow integrity mechanism, and can additionally provide code authentication. We evaluate SCI-FI by extending a RISC-V core. The average hardware area overheads range from 6.5% to 23.8%, and the average code size and execution time increase by 25.4% and 17.5% respectively.

***Index Terms*—fault injection attacks, code integrity, control-flow integrity, execution integrity, control logic, counter-measures**

I. Introduction

Context. Fault injection attacks are known to be an important threat to the security of embedded systems. The fault injection targets the hardware, using various means such as power or clock glitches, electromagnetic disturbances, or laser illumination. It may result into various effects at the logical level, and nowadays powerful attacker models consider the possibility to alter selectively one or few bit values [1]. The attacker aims at inducing computation errors or modifying values in the circuit under attack in order to leverage fault injection for many attack objectives such as extract confidential data, leverage software vulnerabilities, or escalate privileges.

State-of-the-art protections against fault injection attacks enforce three security properties: data integrity, code integrity, and control-flow integrity. Code integrity ensures that program instructions are not modified before execution. Control-flow integrity ensures that control-flow transfers, such as branches and calls, are correct with respect to a reference control-flow graph. A full control-flow integrity also ensures the correct order of the executed instructions. Data integrity is related to both data stored in external memory such as Flash or RAM and data manipulated by the processor. On the one hand, the correctness of the program output depends on the paths taken during the execution of a program. On the other hand, the computations involved in the evaluation of control-flow conditions depend on the data values being processed [2]. Hence, data, code, and control-flow integrity are all required to ensure the correct processing of a computation.

This work was partially funded by the French National Research Agency (ANR) under grant agreement ANR-18-CE39-0003.

Several works study code and control-flow integrity hardware mechanisms based on the computation of an integrity signature. In [3], a hardware monitor, associated to the processor, computes a code integrity signature and uses additional metadata to validate the code and control-flow integrity in separate verification mechanisms. In [4], a single signature mechanism enforces both code and control-flow integrity. Finally, recent counter-measures for code and control-flow integrity are based on the authenticated decryption of program instructions [5]–[7]. They ensure code authenticity and code confidentiality in addition to code and control-flow integrity.

Problem. Recently, Laurent et al. reported that some faults inside the microarchitecture escape all typical software fault models [8], e.g., when the processing of an instruction is altered after the decoding of the binary instruction. Therefore, state-of-the-art counter-measures fail to catch such fault injection attacks, and ensuring both code integrity and control-flow integrity is not sufficient to protect the program execution. We call *execution integrity* the integrity of the control logic in the hardware, and we argue that this new security property is required to protect against powerful fault injection attacks.

In the context of safety, Kim and Somani compute an execution integrity signature from a selection of control signals [9]. However, this work targets soft errors, and assumes that the first program execution is correct, which is not a valid assumption in the context of fault injection attacks.

Goal & Challenges. Our goal is to design a counter-measure against fault injection attacks supporting simultaneously execution integrity, code integrity, and control-flow integrity. Execution integrity is achieved by enforcing the integrity of the processor's control signals, hence protecting the whole instruction path of the processor microarchitecture against fault injection attacks. The first challenge is to implement an efficient signature-based mechanism that ensures execution integrity. We are looking for a signature mechanism that does not impact the processor's critical path, and that minimizes the silicon area overheads. The second challenge is to combine the execution integrity signature with code and control-flow integrity.

Proposal. We propose SCI-FI, a counter-measure against fault injection attacks. The approach is based on a function signature used to compute a runtime signature from control signals emitted by the decode pipeline stage, similarly to [4], which in SCI-FI provides control-flow integrity, code integrity and potentially code authentication, and execution integrity for the frontend stages of the processor pipeline. The control signals in the following stages are protected by a redundancy scheme, which completes the coverage of the processor pipeline in our execution integrity approach. The runtime signatures, computed in the hardware, are checked against reference signatures. These reference signatures are computed at compile-time by a signal-accurate model of the target processor, and are inserted into the program by a dedicated compiler toolchain.

Contributions & Outline. The contributions of this paper are: the description of SCI-FI, a counter-measure ensuring code, control-flow and execution integrity in the context of fault injection attacks, supported by a processor extension and a compiler toolchain (Section II); the description of two different implementations on the RISC-V CV32E40P processor and the evaluation of hardware and software overheads (Section III).

II. SCI-FI Concepts

A. Threat Model

We consider an attacker that only has physical access to the device under attack. The attacker is supposed to use fault injection on the device. They can inject two kinds of faults in the memory or in the processor logic: either a fault with full control over a few bits (typically less than 8 bits), or a fault altering many bits but without any control on the faulted value. They can inject multiple faults at different time locations. We consider fault injections targeting the instruction path only; faults targeting the data path are assumed to be covered by a complementary dedicated mechanism, typically, error detection code in internal data registers. The attacker does not have logical access to the device, and therefore cannot perform common software attacks, nor cannot modify the memory contents through logical access, e.g., by reprogramming it. Moreover, side-channel analysis and invasive attacks such as micro-probing are out of scope.

B. Background

A program can be decomposed in maximal instruction sequences with a single entry instruction and a single exit instruction, commonly called basic blocks. A standard technique to ensure code integrity is to compute a signature for each basic block from the binary encoding of its instructions. The signature S_i associated to a basic block B_i composed of instructions $I_0, \dots, I_n$ is computed using a signature function f and an initialization vector IV_i (1):

$$s_{i_0} = f(IV_i, I_0), \quad s_{i_n} = f(s_{i_{n-1}}, I_n), \quad S_i = s_{i_n} \tag{1}$$

At runtime, the signatures are computed and checked against reference signatures during each control-flow transfer. Reference signatures are precomputed offline, they are either stored in a dedicated memory or embedded in the instruction memory, e.g., at the end of basic blocks.

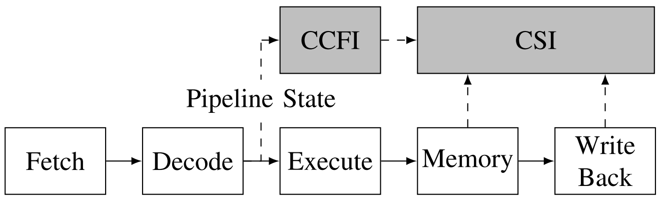

Fig. 1. Illustration of a 5-stage processor extended with SCI-FI (grey modules)

Generalized path signature analysis (GPSA) enforces control-flow integrity by computing signatures that depend on the control-flow graph [10]. Typically, the signature of the basic block B_{i-1} is used as the initialization vector IV_i of the successor basic block B_i. If several execution paths merge into a basic block, patch values are applied to the signature of each predecessor basic blocks $B_j, B_k, \dots$: an update function u generates a unique initialization vector IV_i for every tuple of signatures $S_j, S_k, \dots$ and patch values $P_j, P_k, \dots$ (2):

$$IV_i = u(S_j, P_j) = u(S_k, P_k) = \dots \tag{2}$$

GPSA requires that reference signatures are accessible to the signature verification mechanism. Similarly to code integrity presented above, such signatures are intertwined with program instructions, or stored in a separate data section. Additionally, GPSA requires to instrument the program with patch values.

C. SCI-FI Overview

SCI-FI combines GPSA with a redundancy-based mechanism to ensure code, control-flow and execution integrity in the microarchitecture. An example of the extension of a 5-stage pipeline with the two SCI-FI modules is shown in Fig. 1. The Code and Control-Flow Integrity module (CCFI) implements the hardware support for GPSA and enforces execution integrity up to the decode stage. The Control Signal Integrity module (CSI) completes the coverage of execution integrity through a redundancy-based mechanism. On the software side, SCI-FI requires modifications of the compiler backend to insert signature checks and patch values.

Instead of using binary encodings of program instructions to compute a signature, CCFI uses signals coming from the decode pipeline stage, called the *pipeline state*. CSI checks that signals from the pipeline state are correctly propagated up to their consumption in the subsequent pipeline stages. The selected signals are duplicated into CSI at the output of the decode stage. Then, for each subsequent pipeline stage, CSI checks the original control signals against their duplicates. Therefore, the CSI module can detect any fault on control signals included in the pipeline state after the decode stage up to the pipeline end. Execution integrity of the whole instruction path is ensured by the combination of the CCFI and CSI modules: CCFI ensures the integrity of the pipeline state, and CSI then ensures the integrity of control signals up to their consumption stage.

D. Pipeline State

The construction of a pipeline state requires identifying the control signals that directly map to the binary instruction in order to provide code integrity. Also, GPSA requires that the signature is computed by static analysis, which constrains the selection of signals monitored by the signature. Control signals that only depend on the instruction currently in the decode stage are integrated in the pipeline state, e.g., operands selection signals, ALU control signals and immediates. Some control signals are also part of the pipeline state as they only depend on statically known operand dependencies, e.g., signals controlling forwarding mechanisms.

E. CCFI – Code and Control-Flow Integrity Module

The CCFI module implements GPSA. It requires two functions, for the signature computation and for the application of patch values, with specific properties summarized in this section. Cf. Werner et al. [4] for a detailed discussion. Note that most cryptographic functions intrinsically support all these properties.

1) The signature function f: is the core of the CCFI module, and the GPSA fault detection capabilities depend on f's properties. i) *Collision resistance*: prevents an attacker from forging a faulted basic block presenting the same signature as the signature of the original basic block, or to introduce a second fault reverting a signature change. ii) *Error preservation*: signature changes due to an error are not cancelled by any following error-free sequences. This property, in combination with collision resistance, allows for the arbitrary placement of signature checks. iii) *Non associativity*: sequences of instructions with different orderings produce different signatures. This property ensures control-flow integrity at the level of machine instructions. iv) *Invertibility*: also introduced by [4], is not required by our approach because patch values are applied on the basic block signature instead of being applied on intermediate signatures (see below).

2) The update function u: has the following requirements. i) *Full control*: given a signature, there exists a patch value for any target IV. ii) *Error preservation*: any fault previously introduced in the signature cannot be reverted by applying an error-free update. iii) *Invertibility*: a patch value can be computed from an initialization vector and a signature.

The update mechanism is triggered at each control-flow transfer. The runtime signature is updated using function u and the current patch value. The patch value is stored in a *patch register* in CCFI, and can be updated by a dedicated instruction that loads a patch value from memory. Additionally, the patch register is reset to a default, constant patch value after each control-flow transfer. This default patch value must be known at compile time to compute the reference signatures, and the identity element of u, if it exists, can be used as the default patch value.

When several basic blocks $B_i, B_k, \ldots$ have the same successor B_s, there is at most a single basic block B_f falling into B_s (i.e., the basic block immediately preceding B_s in the memory layout). If B_f exists, its signature S_f is used as the initialization vector IV_s of B_s: $IV_s = S_f$. Otherwise, IV_s is chosen randomly among the signatures of $B_i, B_k, \ldots$. Knowing IV_s and u^{-1}, a patch value is computed for all the other predecessors of B_s.

Currently, SCI-FI does not fully support indirect branches; backward edges, also known as function returns, are the only indirect branches currently handled. The idea is to assume a constant signature at the exit points of a callee for any call site. To complete the control-flow integrity in presence of returns, a shadow stack is required.

3) Signature verification: A runtime signature is computed for every instruction in the program. Thanks to the properties of the functions f and u, any fault captured in the signature will be forwarded into the next ones (cf. II-E1). Therefore, it is possible to insert verifications anywhere in the program.

SCI-FI uses custom control-flow transfer instructions, thereafter called *verification instructions*, which have the same semantics as their original counterpart. Verification instructions load a reference signature immediately following in the program memory, and trigger the signature verification. Then, they proceed similarly to other control-flow instructions: if the branch is taken, the runtime signature is updated with the current patch value, and the current patch value is reset to its default value. When the verification fails, it triggers an exception that calls a software user-designed fault handler.

F. CSI – Control Signal Integrity Module

The CSI module enforces execution integrity for the pipeline stages following the decode stage. The principle is to use a redundancy scheme to detect any change in the control signals constituting the pipeline state, from their emission to their consumption stage. This approach is lightweight because it involves only a small part of the pipeline's control logic. The CSI module duplicates the propagation of selected signals between the different pipeline stages. In each pipeline stage, the duplicated signals are checked against the original ones. The duplication can use any redundancy scheme, potentially with several duplicates, e.g., a simple copy, a complementary copy or the initial value `xor`ed with an arbitrary value.

III. Implementation and Experimental Evaluation

A. Implementation

We integrate SCI-FI to the CV32E40P processor [11], a 32-bit, in-order RISC-V core with a 4-stage pipeline implementing the RV32I base instruction set version 2.1. We select the CV32E40P because such small in-order core is representative of typical fault injection targets, and because a 4-stage pipeline is representative of the main challenges of microarchitectural design due to control and data hazards such as forwarding mechanisms. More complex processors are left for future work.

The pipeline state is constructed manually from the signals described in Section II-D. The control signals outputted by the decode stage and that will go through subsequent stages are duplicated in the CSI module. We use a simple duplication scheme to implement the CSI redundancy.

SCI-FI is implemented with two different single cycle signature functions for the CCFI module: i) a CRC32 designed to detect up to 8 bit-flips per basic block; ii) a CBC-MAC based

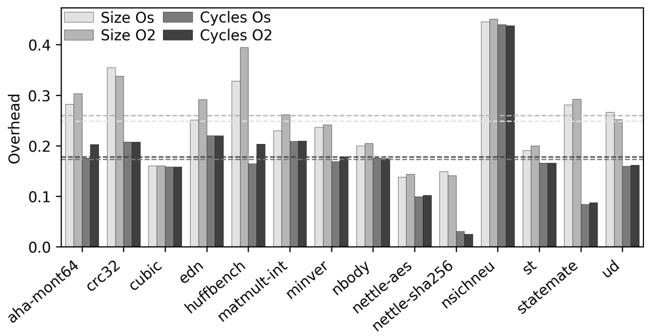

Fig. 2. Execution time and code size overheads for the Embench-IoT benchmarks, with the `-Os` and `-O2` compiler optimization levels.

on a fully unrolled hardware implementation of the Prince block cipher, which is selected for its small silicon area. Thanks to the CBC-MAC signature function, SCI-FI additionally provides code authenticity. We select the exclusive or (XOR) for the update function in the two implementations.

The CV32E40P is modified as follows. All the control-flow instructions update the runtime signature with the current patch value if the branch is taken (Section II-E2). The core is also extended with custom instructions to trigger the signature verifications, or to load the patch values. We implement a verification instruction (Section II-E3) for each control-flow instruction in the RV32I instruction set, and a load patch instruction that fetches a patch value from memory using a new Control Status Register (CSR) as the address base and an immediate value as the offset. The CSR is set during the core bootstrap to point to the `.patches` section of the binary program that gathers all the patches.

The RISC-V backend of the LLVM-12 toolchain is modified to emit the dedicated instructions required by SCI-FI. The reference signatures and the patch values are generated from the final binary program by a static analysis tool and a signal-accurate model of CV32E40P's decode stage.

B. Experimental evaluation

To evaluate the hardware overhead due to SCI-FI, we synthesize the modified CV32E40P into an Application Specific Integrated Circuit (ASIC). The ASIC is designed for a frequency of 400MHz, in the GF-22FDX FDSOI technology, and the target frequency is not impacted by the addition of SCI-FI. The core occupies 55 kGE with CRC32 and 64 kGE with CBC-MAC/Prince, which represents an area overhead wrt. the unmodified core of 6.5% and 23.8% respectively.

The software evaluation is carried out through HDL cycle-accurate simulations of the modified CV32E40P with CRC32. We benchmark our implementation with the Embench-IoT [12] test suite, which targets embedded systems without operating system. 4 tests (picojpeg, qrduino, sglib-combined, and wikisort) are not evaluated because SCI-FI does not support indirect branches yet. All the test programs are compiled with the SCI-FI toolchain, with optimization levels `-Os` and `-O2`, linked with the Newlib C-library and the LLVM soft float library, and signature verifications are inserted in the benchmarked functions only.

Fig. 2 reports the execution time (measured in CPU cycles) and the code size overheads. Note that the code size evaluation considers only the sections impacted by SCI-FI (`.text` and `.patches`), which provides a pessimistic, upper bound of the overall code size overheads for a complete firmware image. Execution time overheads range between 2.5% and 44.0% (geometric average 17.5%), and the code size overheads range between 13.8% and 45.1% (geometric average 25.4%). When compared to existing counter-measures [3]–[7], SCI-FI has comparable overheads but offers in addition execution integrity.

IV. Conclusion

This paper presents SCI-FI, a counter-measure against fault injection attacks. SCI-FI articulates two protection mechanisms to provide full coverage of the control logic of the processor. A first module, implementing generalized path signature analysis (GPSA), builds a signature from control signals to provide simultaneously code integrity, code authenticity, control-flow integrity and execution integrity from the fetch stage to the end of the decode stage. A second module, implementing a redundancy-based mechanism, enforces the integrity of the same control signals in the subsequent pipeline stages. SCI-FI provides comparable overheads to related works, and extends the state of the art of counter-measures against fault injection attacks by combining execution integrity with code integrity, code authenticity and control-flow integrity.

Acknowledgements

We thank Mikael Le Coadou and Juan Suzano Da Fonseca for their contributions to the hardware evaluation.

References

[1] B. Yuce, P. Schaumont, and M. Witteman, “Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation,” *Journal of Hardware and Systems Security*, 2018.
[2] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, “Compiler-Assisted Loop Hardening Against Fault Attacks,” *ACM Transactions on Architecture and Code Optimization*, 2017.
[3] J.-L. Danger *et al.*, “Processor Anchor to Increase the Robustness Against Fault Injection and Cyber Attacks,” in *COSADE*, 2020.
[4] M. Werner, E. Wenger, and S. Mangard, “Protecting the Control Flow of Embedded Processors against Fault Attacks,” in *CARDIS*, 2015.
[5] R. de Clercq *et al.*, “SOFIA: Software and control flow integrity architecture,” in *DATE*, 2016.
[6] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, “Sponge-Based Control-Flow Protection for IoT Devices,” in *EuroS&P*, 2018.
[7] O. Savry, M. El-Majihi, and T. Hiscock, “Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption,” in *DSD*, 2020.
[8] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, “Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor,” *Microprocessors and Microsystems*, 2019.
[9] S. Kim and A. K. Somani, “On-line integrity monitoring of microprocessor control logic,” *Microelectronics Journal*, 2001.
[10] K. Wilken and J. P. Shen, “Continuous signature monitoring: Low-cost concurrent detection of processor control errors,” *IEEE TCAD*, 1990.
[11] OpenHW Group, “CV32E40P,” https://github.com/openhwgroup/cv32e40p, 2021.
[12] “Embench™: Open Benchmarks for Embedded Platforms,” https://github.com/embench/embench-iot, 2021.