Authors' version
Cosade 2023
https://doi.org/10.1007/978-3-031-29497-6_3

A CCFI verification scheme based on the RISC-V Trace Encoder

Anthony Zgheib, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Max Dutertre

Mines Saint-Etienne, CEA-Tech, Centre CMP. Gardanne, France
{zgheib, olivier.potin, rigaud, dutertre}@emse.fr

Abstract. Control-Flow Integrity (CFI) is used to check at runtime that a program's execution path follows its corresponding Control-Flow Graph (CFG) and is not altered by software or physical attacks. In addition to the CFI's features, the Code and Control-Flow Integrity (CCFI) verifies the integrity of the executed program code. This paper presents a CCFI verification system for programs executed on RISC-V cores. Our solution is built upon the RISC-V Trace Encoder (TE) that provides information about the execution path of the user's program. An evolution of the TE specifications and additional logic have made it possible to monitor the integrity of a program control flow and of all the executed instructions. We implemented this approach on a RISC-V core and simulated its efficiency against Fault Injection Attacks. Its average hardware area overheads range from 26.1% to 44.5%. Compared to existing CCFI solutions, our methodology does not modify the user code, the RISC-V compiler or the core's pipeline.

Keywords: RISC-V · Trace Encoder · CFI · CCFI · FIA.

1 Introduction

Fault Injection Attacks (FIA) are effective threats that can alter the intended behavior of a program running on a processor. The most common FIA techniques are described in [3]. These attacks could lead to skipping or corrupting a vulnerable instruction in the user application code, in order to bypass system security features [18] (e.g. bypassing a PIN code [12]). Against these attacks, Control-Flow Integrity (CFI) [1] verification schemes are used to verify that a program is correctly executed during runtime. It checks that its execution follows a path known to be correct in the application Control Flow Graph (CFG). This CFG can be drawn by statically analyzing the source code of the program (if all destinations can be computed during the compilation process). Note that indirect jump destinations in a program may not be predicted at compilation time, in this case the generation of the graph is difficult. The CFG represents the valid control flow changes in a normal program execution [8]. However, attacks made on instructions/operations within the user code are not always detected by CFI solutions, such as changing an addition of two values into a subtraction (if no violation of the CFG is induced). Code and Control-Flow Integrity

(CCFI) countermeasures are designed to verify at runtime, in addition to CFI, the integrity of the executed user code. With this verification, the entire code is protected against FIA not only the discontinuity instructions (monitored by CFI).

Contributions Our work contributes to the CCFI state-of-the-art by adding a solution that does not require any code or compiler modification. In addition, no core nor Instruction Set Architecture (ISA) extension are made. Our solution consists in adding an additional verification system to the RISC-V core [2]. It detects software or physical attacks that derive the program CFG from its normal behavior. This graph is formed from all known destinations of the binary code. Therefore, our solution does not cover forward edge attacks (faults on indirect jump destinations that are not precisely known at binary level) nor attacks injected on data (register or memory). Our CCFI verification system is based on the RISC-V Trace Encoder (TE) [17]. To the best of our knowledge, this is the first solution that uses the RISC-V TE for CCFI verification.

Organization Our paper is divided as follows: Section 2 provides insights on existing CCFI solutions. Sections 3 and 4 describe our CCFI methodology and countermeasure. Section 5 shows its effectiveness against simulated FIA. Sections 6 and 7 report the hardware requirements and the discussion about our solution. Finally, we conclude our paper in the last Section.

2 Related work

In this section, the most relevant CCFI verification solutions are presented. Some countermeasures ensure both the integrity and confidentiality of the user code by encrypting the code instructions and deciphering it at runtime. From this category, we can cite SOFIA [7]. It is a hardware-based security architecture that protects the software integrity, performs CFI, prevents execution of tampered code and enforces copyright protection. This countermeasure is added by extending the processor. In another perspective, other countermeasures modify the user code and compiler to insert dedicated CCFI instructions. Werner et al. [20] designed SCFP, a solution that guarantees the confidentiality of a software IP and its authentic execution on a microcontroller. It covers code reuse, code injection and fault attacks on the code and control flow. The SCI-FI [4] solution belongs also to this category. It is designed for control signal, code and CFI verification. It protects against FIA. The verification process is triggered by dedicated and customized instructions added by extending the RISC-V ISA. Another approach is to connect external blocks to the processor without extending the ISA to verify the program CCFI like the solution presented in CCFI-Cache [6] and ATRIUM [21]. Danger et al., in [6], developed a hardware based solution that verifies code and CFG, ensures protection against cyber and physical attacks. It covers backward edges and forward edges in certain cases —

when the indirect jump targets a destination address not pointing to a beginning of a Basic Block (BB), detected by checking the `StartBB` label—, code and fault injection. ATRIUM is a runtime attestation scheme targeting "bare metal" embedded systems software that works in parallel to the processor. It ensures CFI and instruction integrity. This solution covers code injection, code reuse, hardware fault attacks on instructions and TOCTOU (Time Of Check Time Of Use) attacks [19]. The previous countermeasures have an impact on the runtime of the programs. Except ATRIUM, these solutions require user code modification to ensure CCFI verification. Table 1 summarizes the average overhead costs of these countermeasures in terms of code size, performance and hardware area compared to our solution whose overhead is detailed in Section 6. In the following sections, we describe a new CCFI scheme that keeps unchanged the user code, compilation process and core design.

Table 1. State-of-the-art solutions average overhead costs.

Solution	*SOFIA [7]*	*SCFP [20]*	*SCI-FI [4]*	*CCFI-Cache [6]*	*ATRIUM [21]*	**This Work**
Code Size (%)	141	19.8	25.4	<30	0	**0**
Performance (%)	110	9.1	17.5	32	<22.7	**0**
Hardware Area (%)	28.2	N/A	<23.8	10	<20	**<44.5**

3 CCFI Methodology

Our CCFI verification methodology is divided into 3 steps:

1. The static analysis of the binary code to obtain its CFG.
2. The generation of metadata related to the discontinuity instructions and Basic Blocks (BB, cf. Section 3.2 for definition).
3. The addition of an external hardware module — the Trace Verifier (TV) — to proceed to the CCFI verification.

Each step is described in detail in the next sections.

3.1 Static Analysis

A custom program has been developed to analyse statically the user's binary code in order to derive its CFG. This graph shows all the legitimate paths that a program could follow. From this analysis, all discontinuity instructions are reported.

Algorithm 1 CFG Generation

Require: *Binary Code*
Ensure: *Discontinuity instructions with their Basic Blocks (cf. Section 3.2 for definition)*

```
for i ← program.begin to program.end do
    #Discontinuity on Branch, Jump, or Return
    if discontinuity instruction then
        #Default Report
        report address at i;
        report instruction at i;
        if branch instruction then
            report next discontinuity's address when branch taken;
            report next discontinuity's address when branch non taken;
        else if jump instruction then
            report next discontinuity's address when call;
            report return address # equal to jump address + 4
        else if return instruction then
            report address and instruction;
        end if
    end if
end for
```

They refer to direct jumps —Jump (J) and Jump And Link (JAL) instructions—, branch and return instructions. Algorithm 1 illustrates the pseudo-code of the static analysis process used to build the program CFG. The application reports for each discontinuity instruction the address(es) of the next attempted discontinuity instruction(s). For a branch instruction, two addresses are reported: the first when the branch is taken and the second when the branch is not taken. In our strategy, indirect jumps represented by the Jump And Link Register (JALR) instructions are not considered. These instructions involve the code's program counter (PC) to jump to a destination whose address is calculated according to the content of a register and a value contained in its binary instruction. This register's content is not known at the time of the static analysis. However, the possible addresses can be guessed by parsing the code to form an array of possible destinations. This strategy complicates the control flow schemes to check for a correct destination. To allow easy and accurate extraction of the CFG, Gonzalvez et al. [11] proposed two modified ISAs by removing indirect jumps from a program. In our case, we dedicate our approach to cover the CCFI for programs that do not contain indirect jumps. We further discuss in Section 7 how these jumps could be treated as a further work.

3.2 Metadata Generation

From the discontinuity instructions of the CFG, metadata are generated constituting the CFG's map. Each data element contains the discontinuity instruction, its address and the index (address in the memory) of the following discontinuity.

These instructions delimit a Basic Block (BB): a set of successive instructions for which execution is done consecutively and in order. A BB starts with the first instruction following a discontinuity instruction until the next discontinuity. In addition to the information stored in the data elements, hash signatures of BBs are calculated. A hash signature computation is made on the binary value of all the 32-bit length instructions of a BB using a Multiple-Input Signature Register (MISR) mechanism [9]. It starts from the BB first instruction until the end of the BB for which the signature is generated. This signature is stored along with the discontinuity instruction pointing to the address of the BB first instruction. A runtime verification of this signature is bound to check the integrity of the executed instructions. Fig. 1 illustrates an example of four BBs delimited by `jump (j)`, `branch if not equal to zero (bnez)`, `jump and link (jal)` and `ret (return)` instructions with their metadata. For example, the hash signature of "Basic Block 2", delimited by the addresses `0x374` and `0x380`, is stored with the discontinuity instruction pointing to the starting address of this block — instruction `j 374` at address `0x328`. For a function call using `JAL` instruction, its return address reported by Algorithm 1 is used to delimit the BB formed by the instructions after its return till the next discontinuity. The instruction `jal ra,308` at address `0x3a8` in Fig. 1 illustrates this case. Its corresponding metadata contains (see index `40`), its address, the instruction and the index `3B` of the next discontinuity instruction, the expected signature of "Basic Block 1" (BB starting with the instruction after the jump) and the expected signature of "Basic Block 4" (representing the BB after the return starting at address `0x3ac`). A stored signature is a prediction of the correct BB signature when executed on core. A recalculation of the BB signature is done at runtime and an additional hardware module — the Trace Verifier (TV) — is in charge of comparing it with the metadata (stored in a memory).

3.3 Trace Verifier

The TV is an additional hardware module (cf. Fig. 2, bottom). It receives, at runtime, information about the execution path followed by the program. These information are reported by the RISC-V Trace Encoder (TE) [17]. Based on the CFG metadata (cf. Section 3.2), the TV checks that the execution path of the program is included in its CFG. It also ensures the integrity of the user application code as a security propriety by verifying the BB signatures. An alarm is raised if a CFG derivation has been detected. The following section describes in more details our countermeasure.

4 Proposed CCFI Solution

4.1 Trace Encoder

Overview The TE is a RISC-V hardware module [17]. It is an execution flow tracer that compresses at runtime the sequence of discontinuity instructions

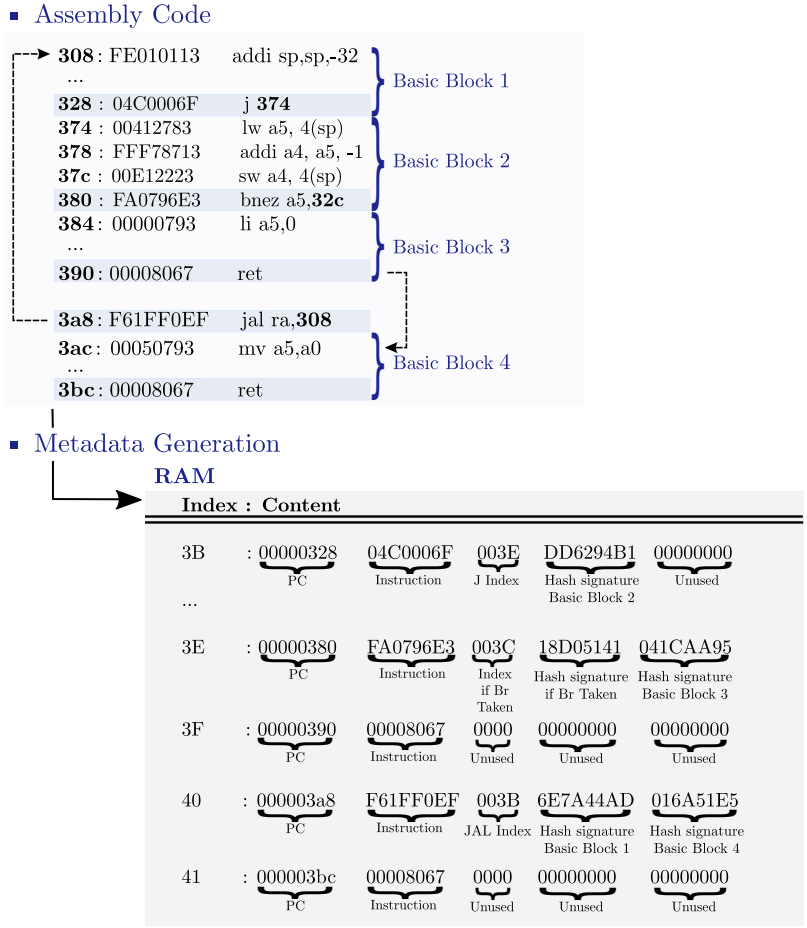

Fig. 1. Generated metadata content.

executed by the RISC-V core into trace packets. These packets sent to a debug tool allow developers to check the path followed by the program. By having access to the program binary, developers can reconstruct the program flow as depicted in Fig. 2 (top). This module alone is used for debugging purposes and allows neither CFI nor CCFI verification. The TE has a 3-stage pipeline to store the current (I), previous (I-1) and next (I+1) instructions [17]. Based on these three instructions, a packet defined by the TE standard [17] containing information about the path followed by the program since the last sent packet is emitted to an external debugging tool. It is emitted after fulfilling one of the seven conditions described in the TE specifications [17]. These conditions are related to the state of the core (context, privilege, exception) or to the instructions executed (first executed, discontinuity instructions, etc). We briefly describe three of these conditions below that led in the verification of CCFI.

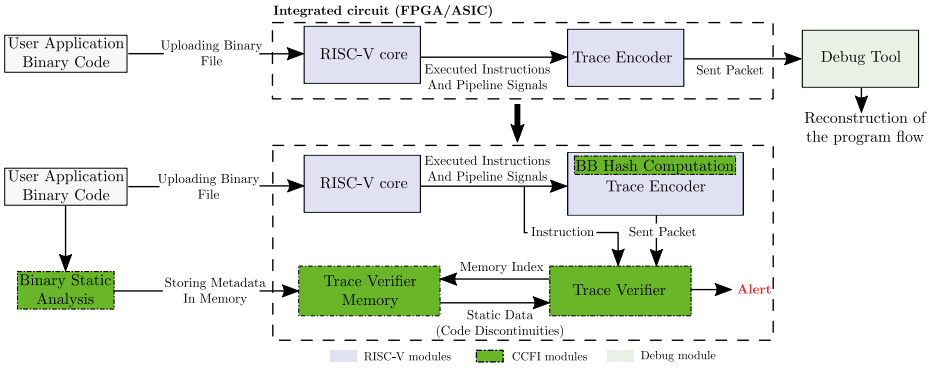

Fig. 2. A schematic of the RISC-V + TE (top), and its extension to ensure CCFI verification (bottom).

The other four conditions involve reporting the state of the core (as defined above), which does not cover the verification of CCFI but can be used to handle interruptions or core's exceptions. A packet is sent:

- Based on the previous instruction (I-1):
  a) An instruction with an unpredictable PC discontinuity is executed. This type refers to instructions applying a change to the PC whose offset could not be determined from the compiled code such as return instructions. To be able to follow the program path, the TE reports these discontinuities in form of trace packets. Hence, in its current configuration, it does not send a packet after each discontinuity instruction.
- Based on the current executed instruction (I):
  b) A first qualified instruction which refers to the first instrution executed in a program's code.
  c) The TE branch map is full (number of branches=31, a packet is issued to clear its branch map) or it has a misprediction case (when branch predictor enabled).

Depending on these conditions, a packet is sent with a specific format identifier [17]. Referring to this standard, four packet formats are defined:

- **Format 0** is used to send optional efficiency extensions (such as the number of correctly predicted branches) when the core's branch prediction module is enabled. In our research, we based our CCFI solution while this module is disabled. A discussion about CCFI verification with this module active could be found in Section 7.
- **Format 1** reports a branch information when the TE branch counter reaches its maximum value (31 branches). Or, when an address needs to be reported and there has been at least one branch since the previous packet. This format only contains branch information.

- **Format 2** reports only the address of an instruction when no branch information need to be reported (for example, executing only a return instruction after the last packet sent).
- **Format 3** is used for synchronization, reporting context and supporting information.

An example of a Format 1 packet is illustrated below.

- **PACKET 1:**
  - branches: n
  - branch_map: n_map
  - absolute_address: PC

This packet is sent after executing a discontinuity instruction satisfying the first condition `a`. The unpredictability imposes the sending of a packet in order not to lose the thread of the program executed flow. This packet indicates that `n` branches have been executed since the last sent packet. It also mentions the `branch_map` (bit vector where taken/not taken status of each branch is stored chronologically) and address of the executed instruction after the discontinuity.

CFI A prior work [22] exploits the TE in order to verify the CFI of a program executed on a RISC-V core. This design allows the detection of CFG integrity violations. Two CFI verification approaches were proposed. The first approach is consistent with the TE standard [17]. With this approach, only instruction skip on discontinuity instructions and backward edge attacks are detected. The second approach suggests an enrichment of the standard in order to detect more threat models. A packet is sent after each executed discontinuity instruction and not just after the unpredictable ones as in the first approach. This packet contains the address of the following executed instruction and more information depending on (I-1), (I) and (I+1) instructions. This permits to detect in addition to the previous threats, any corruption of a discontinuity instruction.

CCFI Our paper contributes in extending the work of [22], by additionally adding a new functionality to the TE — thanks to its open-source specifications — to work in the CCFI verification mode. A hash signature computation is done on each BB in order to protect the entire code against FIA and not only the discontinuity instructions as in the CFI mode. The end of a BB is identified when a discontinuity instruction is executed. As for the CFI enhanced mode, a packet is sent after each discontinuity instruction. The BB generated signature is included in this packet with the information it sends originally. As an example, a Format 1 packet in CCFI configuration is enhanced as shown below.

- **PACKET 1:**
  - branches: n
  - branch_map: n_map
  - absolute_address: PC

- **Signature_sent: Computed_Hash_Signature**

Fig. 2 (bottom) illustrates the circuit for CCFI verification. It is composed of the TV, its memory containing the static metadata and the signature computation module (BB hash computation) in the TE.

4.2 Trace Verifier Hardware Description

As depicted in the bottom part of Fig. 2, our verification system is constituted by a memory (Trace Verifier Memory) and its core part (Trace Verifier).

TV Memory The produced metadata are stored in a dedicated memory — Random Access Memory (RAM) — as illustrated in Fig. 2. Referring to Fig. 1, at index `3B`, the **32-bit** jump instruction `j 374` is stored with its **32-bit** address `0x328`, the **16-bit** index of next discontinuity and the 32-bit signature `0xDD6294B1` of the following BB. In case of a branch instruction (e.g. at index `3E`), **two 32-bit** hash signature values are stored referring to the two possible branches. In total, each discontinuity instruction requires **144-bit** of metadata.

TV Architecture Fig. 3 shows the architecture of our TV. It is composed of configurables modules (FIFO and LIFO), a Finite State Machine (FSM) and several processes. The verification process starts when it receives a packet from the TE that activates its FSM (1). Meanwhile, the packet is stored in a FIFO and will be acquired by the FSM (2). Subsequently, it is decoded in order to extract the reported address and signature (3). In case of a packet reporting the execution of a branch instruction, the branch and branch map are also extracted. Having the packet information, a navigation through the RAM metadata is done to constitute the path followed by the program and the expected hash signature (4). The last step of the FSM is to check the address stored in the packet against the static address computed from the navigation process and also compare the reported hash signature to the calculated signature (5). If the addresses and/or signatures are not equal, an error flag is raised. The process of resilience is not discussed here. This error could be treated as a software exception or hardware interruption with a dedicated process or as a message sent to the user.

TV FSM The five steps listed in the **TV Architecture** are explicitly represented in Fig. 4. The TV is in an idle state until it receives a packet from the TE. Then, this packet is decoded to check its format — this refers to step 2 in Fig. 3. Step 3 — `Packet extraction process` — is divided into 2 sequential FSM states (which requires 2 clock cycles). In the first state, the packet format is read. Then in the second state, the format-related fields are extracted (branch, address, signature). For instance, referring to Fig. 1 at index `3E` and after the execution of the branch instruction `0xFA0796E3`, a Format 1 packet is reported indicating if the branch is taken or not. In case of a taken branch, the BRAM index will point to `3C` and the branch address is extracted from the metadata

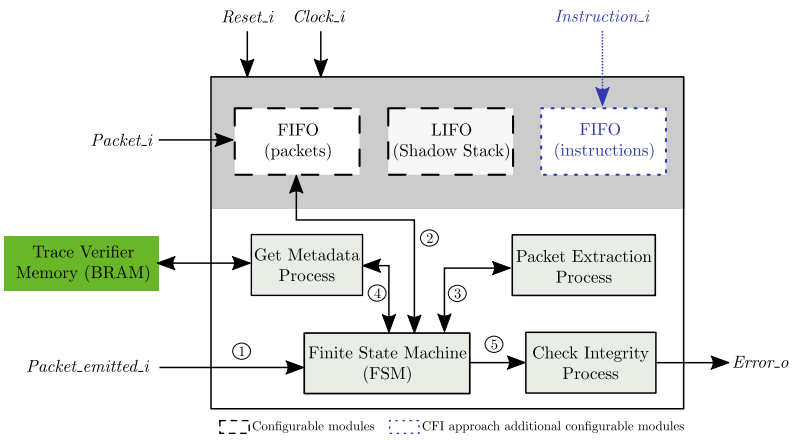

Fig. 3. Architecture of the TV.

binary instruction. In the other case, the index will be incremented by 1 to reach the index `3F` which refers to the next planned discontinuity in the code. This corresponds to step 4. In the step 5, the TE content is verified by comparing the metadata extracted address to the TE reported address. The expected signature for the actual BB (contained within the previous metadata instruction) is also compared to the hash signature reported by the TE. The communication with

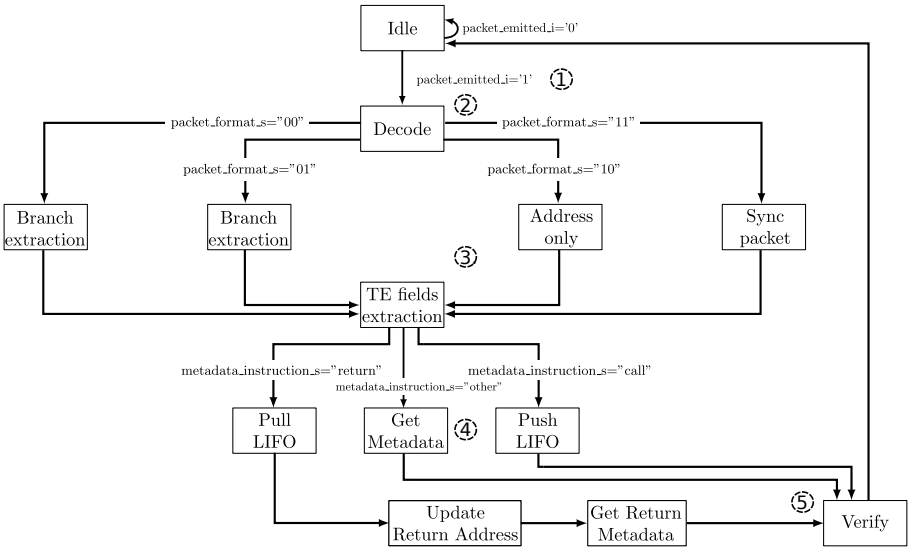

Fig. 4. FSM of the TV.

the LIFO representing the "Shadow Stack" of our TV (cf. Fig. 3) is done when the BRAM navigation process points to a call or return instruction. After the FSM state `TE fields extraction`, the next state will depend on the type of the pointed instruction in the BRAM. We can distinguish 3 categories:

- **Function call (JAL instruction):** In this case, the next FSM state `Push LIFO` stores the instruction index in the LIFO module. Additionally, the call address and expected signature are extracted.
- **Return instruction:** The last call index stored in the LIFO is retrieved via the state `Pull LIFO`. The TV adds four to the retrieved call address in order to get the return address of the called function. It also increments the BRAM index by one to point to the discontinuity instruction following the call via steps `Update Return Address` and `Get Return Metadata`. The second signature stored at the call index corresponds to the expected signature for the BB executed after the return (as illustrated in Section 3.2). This signature is also extracted to be verified in the `Verify` state.
- **Branch or Jump (J) instruction:** The address and signature from the metadata are extracted in the state `Get Metadata`.

As an example, we refer to a function call in Fig. 1. The call instruction `jal ra,308` pushes the index `40` into the LIFO. After the execution of the return instruction at address `0x390`, the index is extracted from the LIFO, and then the return address is calculated by adding four to the call address `0x3a8`+4, which is equal to `0x3ac`. In addition, the BRAM index points to the call's index `40` incremented by one referring to the next planned discontinuity at index `41`. At the verification step, the calculated address `0x3ac` is compared to the TE reported address in addition to the signatures. The expected signature for the actual BB is extracted from the previous discontinuity metadata (the branch instruction at address `0x380`). It is equal to `0x18D05141` if the branch is taken (`0x041CAA95` if not). The verification process requires six clock cycles to verify the content of a packet and eight cycles when the instruction fetched from the BRAM is a return instruction. The two additional cycles are needed to calculate the return address and the following discontinuity index based on the call index stored in the LIFO via steps `Update Return Address` and `Get Return Metadata`.

5 FIA on a Memcmp application code

In this work, we address the protection of programs executed on a RISC-V core. Our CCFI solution detects attacks that divert the program CFG from its normal behavior. We consider that the attacker is able to alter the contents of the instruction memory by physical means (e.g. by laser injection at the reading of instructions from the memory [5]). We assume that the metadata stored in the memory of the TV cannot be modified by the attacker in order to defeat our countermeasure (this requires changing the hash signature in addition to the fault injection at the instruction memory level, which is difficult). As illustration, this section demonstrates the detection of a FIA (physical attack) targeting a

vulnerable instruction of a non protected comparison function `Memcmp`. It also outlines how our CCFI verification solution detects this attack.

Memcmp The function compares the values of two arrays. Its C function is shown in Fig. 5. The parameter `n` specifies the number of values to compare from

```
int memcmp(const void *src1,
           const void *src2,
           unsigned int n) {
 unsigned char *s1 = (unsigned char *)src1;
 unsigned char *s2 = (unsigned char *)src2;
 while (n--) {
  if (*s1 != *s2) {
    return *s1 - *s2; }
  s1++;
  s2++; }
 return 0; }
```

Fig. 5. Memcmp C Code.

`src1` and `src2` arrays. If no difference is reported between their elements, a value of `0` is returned. Otherwise, the difference of the first two different elements is returned.

FIA Scenario An attacker might be interested in altering the result of this function. A fault could be injected to point that there is no difference while two different arrays are compared (e.g. a hash signature checking in an authentication process). This is possible by faulting the `n` value in the `while` condition (cf. Fig. 5). This condition checks that the number of values to compare `n` is greater than `0`. It allows to enter the loop and to compare values from both arrays. Then, the value of `n` is decremented. The assembly code in charge of this operation is shown in Fig. 6. The instruction at address `0x374` retrieves the `n` value from the processor stack and store it in the `a5` register. At address `0x380`, a comparison of the `a5` content with zero is done to decide if the program enters the `while` loop. If `a5` content is different than zero, a comparison of array values is done.

FIA setup The code analysis of the unprotected `Memcmp` function leads to vulnerabilities, one of which is found at the instruction `lw a5,4(sp)` (cf. Fig. 6). A fault transforming this instruction to `li a5,0 (0x00000793)` writes in the `a5` register the value `0`. This is a complex fault that requires 4 bit flips. However, the FIA state-of-the-art proves that it is possible [5]. Based on this attack, the correct value of `n` is not retrieved from the stack. A comparison of the `a5` content with zero is done at address `0x380`. In this case, the `branch if not equal to`

```
32c:   01c12783          lw      a5,28(sp)
                 .........
374:   00412783          lw      a5,4(sp)
378:   fff78713          addi    a4,a5,-1
37c:   00e12223          sw      a4,4(sp)
380:   fa0796e3          bnez    a5,32c
384:   00000793          li      a5,0
388:   00078513          mv      a0,a5
38c:   02010113          addi    sp,sp,32
390:   00008067          ret
```

Fig. 6. Memcmp Assembly Code

`zero (bnez)` condition is not fulfilled and the branch is not taken because `n=0`. Therefore, no comparison of values is done. The `Memcmp` function returns `zero` reporting that there are no differences between the two arrays even though they are different. This attack could not be detected by pure CFI solutions, but rather by CCFI countermeasures checking the integrity of the code.

CCFI verification After executing the discontinuity instruction at address `0x380`, a packet is sent by the TE. Fig. 7 illustrates the faultless packet and the packet content if the FIA is done on the `lw` instruction. Referring to the attack

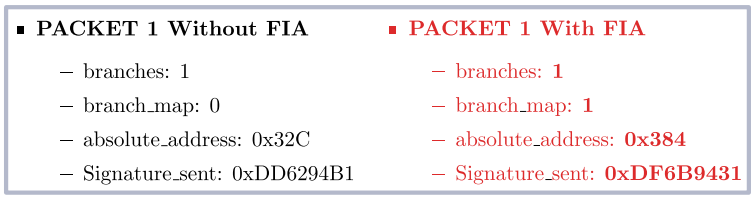

Fig. 7. FIA impact on the sent packet content.

scenario described in the **FIA setup**, corrupting the `lw` instruction generates a different hash signature at the end of the BB. We have simulated this fault attack by modifying the binary instruction at memory level. Fig. 8 shows a simulation of the packet emission due to execution of the `bnez` instruction. The TE awaits the execution of two more instructions after the branch in order to have a visibility on the last three executed instructions to be able to generate a packet as discussed in Section 4.1. We present in Fig. 9 a simulation of the verification of the concerned packet. The enumerated FSM steps (as described in Fig. 4) lead to the integrity verification of the BB. As the signature of the faulted execution `0xDF6B9431` is different from the signature `0xDD6294B1` expected by the TV (cf. the correct one also in Fig. 1), the TV raises an error flag. Therefore,

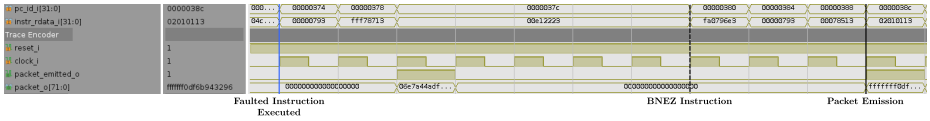

Fig. 8. Simulation of a packet emission due to the execution of the bnez instruction.

the FIA is detected. A comparison of our solution with the CCFI state-of-art

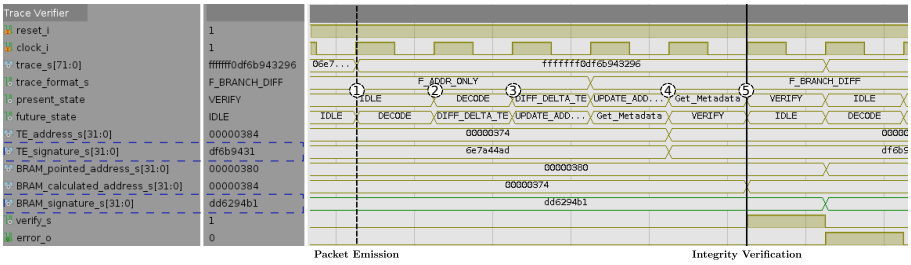

Fig. 9. Simulation of a packet's verification by the TV.

solutions and the CFI solution of [22] could be found in table 2. Compared to [22], our solution covers the integrity of the code. ATRIUM [21] has similar CCFI characteristics. However, it locks the processor if the hash of the current instruction block is not completed and a new block arrives (28 cycles are required to hash a block). The generated signature is sent at the end of the code region chosen by the trusted verifier `vrf` for remote CCFI verification. Our hash module requires only one cycle to compute the signature of an instruction and does not interfere with the core's activity. Compared to the CCFI related works, our solution does not impact the execution runtime of the user application code. Table 1 shows the overheads of these countermeasures compared to our solution which has no impact on the user code (size or execution time). It is a hardware verification method that neither modifies the RISC-V pipeline nor the compiler.

6 HARDWARE METRICS

All our simulations target the Artix-7 Field-Programmable Gate Array (FPGA) embedded on a Nexys video board. This FPGA contains 33,650 logic slices. Each slice is composed of four 6-input LUTs, 8 flip-flops, multiplexers and carry units. A description of the hardware requirements of our system in terms of slice is provided in the following parts.

6.1 Target Core

Our CCFI solution is implemented on an IBEX core [13]. It is a 32-bit open source RISC-V, low power core with a 2-stage pipeline suitable for IOT applications. Its area cost is equal to **645** slices. As our solution is independent of the chosen RISC-V core, it can be implemented on any core compatible with the TE. For instance, our TV could also be applied to the CV32E40P, a 32-bit 4-stage RISC-V core [14]. Its core implementation requires **1171** slices.

6.2 Trace Encoder

The TE module is extracted from the pulp-platform project [16]. Its implementation needs **239** slices. To verify the CCFI of a program, we made an enhancement to the standard in a way to send a packet after each discontinuity instruction including the signature of the executed BB. This enhancement and the additional signature module cost **62** slices while respecting the retro-compatibility of the TE. It can run in a normal [17] or CFI/CCFI mode. Note that, the 32-bit hash signature computation module is designed to compute the signature of an instruction in one cycle. In total, the TE requires **301** slices.

6.3 Trace Verifier components

The TV is divided in 3 parts: its memory to store the static metadata — implemented as a Block Random Access Memory (BRAM) on FPGA, its core and its configurable block (FIFO and LIFO modules).

BRAM Our CCFI solution was tested on several benchmarks from Pulpino project [15], Embench-IOT benchmarks [10] and some classic ones. These benchmarks were compiled with the RV32IM base instruction set and into 3 compilation optimizations level: **O1** for the basic level, **O2** for the advanced level and

Table 2. Comparison of our solution with related works

Solution	*SOFIA [7]*	*SCFP [20]*	*SCI-FI [4]*	*CCFI-Cache [6]*	*ATRIUM [21]*	*TE-CFI [22]*	**This Work**
No User Code Modification	✗	✗	✗	✗	✓	✓	✔
No Compiler Modification	✓	✗	✗	✗	✓	✓	✔
No Pipeline Modification	✗	✗	✗	✓	✓	✓	✔
No Performance Overhead	✗	✗	✗	✗	✗	✓	✔
Backward Edge Protection	✓	✓	✓	✓	✓	✓	✔
Forward Edge Protection	✗	✓	✗	(✗)	✗	✗	✘
Code Integrity	✓	✓	✓	✓	✓	✗	✔
Code Confidentiality	✓	✓	✗	✗	✗	✗	✘

O3 for the highest possible optimization level. As an illustration, Fig. 10 shows the ratio between the generated metadata and code size for the 3 optimization levels. The Metadata-Code size ratio ranges from 15% and 55%. The small benchmark codes have the highest ratio (e.g. the `Memcpy` and `Memcmp` codes). The BRAM is designed with a single read port model. The writing of the metadata is done upstream of the code execution. We have chosen the index width of the BRAM memory to be 16. This value delimit the depth of the metadata memory to $2^{16} = 65536$ lines. To give an order of magnitude, the "nshichneu" code from Embench-IOT [10] contained the most discontinuity instructions. In total, 1188 lines (discontinuities) were needed for a optimization in O1. An index size of 11 was sufficient to address these lines. With this configuration, the memory implementation requires only 8 BRAM blocks without additional slices (this is also the case for indexes of less than 12 bits). This represents the maximum hardware utilization for the simulated benchmarks. The choice of having a large memory is to give the user an order of magnitude for his implementation in case his code contains less than 65536 discontinuity instructions. With a 16-bit index width, the TV memory's implementation requires 256 BRAM blocks and **148** slices (BRAM uses internal register stages). For a complex user code, the BRAM size is sufficient to contain the metadata related to its discontinuity instructions.

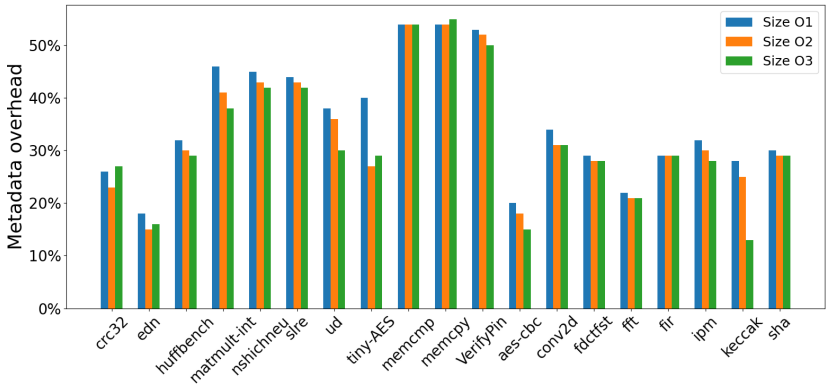

Fig. 10. Ratio between the metadata vs code size.

TV core It is composed of the FSM and processes. It requires **170** slices and could run in CFI or CCFI mode. A TE configuration packet initiates, at startup, the verification mode.

TV Configurable block It represents the LIFO (shadow stack) and FIFOs for storing the packets and discontinuity instructions. Their sizes are dependent of the running application and could be configured to a specific application.

However, in the evaluation phase of our approach, we have chosen sufficiently large sizes to simulate all the targeted benchmarks for an FPGA or even ASIC implementation. Note that the instructions FIFO (cf. Fig. 3) is only used in the CFI mode. As a consequence, a comparison between discontinuity and metadata instructions is performed for a given address to only check the integrity of the executed discontinuity instruction. In the CCFI mode, each identified BB containing a set of instructions ends with a discontinuity instruction. Therefore, a corruption of the discontinuity affects the computed hash signature. A FIA on the analyzed BB is detected by comparing the computed and metadata signatures. In this mode, the instruction FIFO is useless. As illustration of the configurable block requirements, Fig. 11 shows the FIFO and LIFO depths to store data for the compiled benchmarks with the O2 optimization on a log scale. The maximum FIFO and LIFO depths are respectively **153** and **9**. Each application requires a different depth depending on the number of discontinuities/packets sent and the size of the BBs. The verification latency becomes important when packets are sent simultaneously and could not be verified by the TV on the fly. We discuss in Section 7 the reason behind this accumulation of packets. In order to have a generic solution compatible with more complex benchmarks and to avoid the overflow phenomenon, the FIFO and LIFO are designed to store 512 and 16 values respectively. Their implementations require respectively **12** and **3** slices. The overall area cost of the TE optimization (**62** slices, cf. Section 6.2) and TV (**333** slices for 16-bit LIFO index) is equal to **395** slices. When working with an 11-bit index, the TV only requires **185** slices. We have simulated the benchmarks used by [22] and an 11-bit index was enough to point all their discontinuity instructions. Therefore, compared to the TE-based CFI solution of [22] which only adds 17% in terms of slices over the IBEX + TE requirements, our solution requires 26.1% (for an index less than 12-bit) and 44.5% (for a 16-bit index). The area overhead is due to the additional slices required by our TV to work in the CFI or CCFI mode. It is also related to store 2*32 extra signature bits for each discontinuity instruction and their verification process.

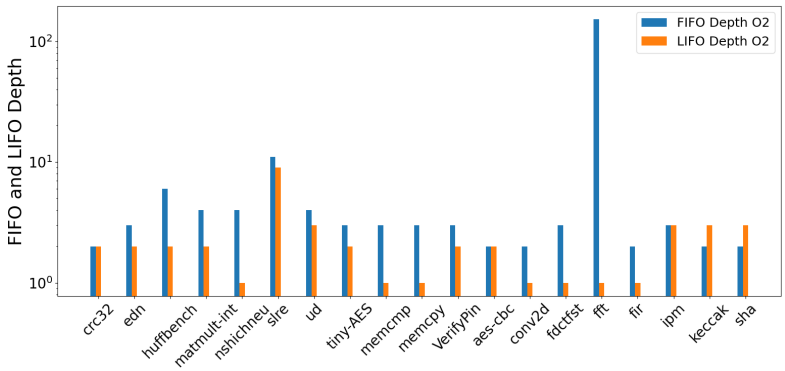

Fig. 11. FIFO and LIFO depths for programs compiled with O2 optimization.

7 Discussion

Our TV does CCFI verification at runtime after receiving a packet from the TE. It needs 6 to 8 clock cycles to process a packet (cf. Section 4.2). The verification latency becomes significant when discontinuity instructions follow each other with fewer clock cycles than is required to process successive packets by the TV (6 clock cycles). Therefore, an increase in the FIFO depth is mandatory when a packet is issued after a BB execution containing less than 6 non-multi-cycle instructions. These executions send packets simultaneously. To process all of these packets, the FIFO is required to store them. Fig. 11 illustrates the packet accumulations which induces a latency to verify all of them. The FIFO is used in order not to stall the processor while a packet is being verified. The depth of the FIFO can be calculated statically by analyzing the binary code of an application. This can be done by counting the instructions formed by all BB and comparing the count to the number of FSM states. In the case of a parameter-conditioned loop containing fewer clock cycles than the FSM check cycles, the user can predict the number for that loop by analyzing the code to correctly increment the FIFO depth. Additionally, our verification is based on the static metadata. Indirect calls destinations are not known from the static analysis and then not covered in our solution. However, these calls emit a packet after their execution. To treat these instructions, we can wait for the packet sent after the one related to the indirect call. Having instruction and packet information, a navigation through the metadata could be done in order to find the discontinuity address and resume verification. Another perspective is to avoid these calls by modifying the user code or the compiler. In addition, our experiments covered the CCFI of all the user code. Referring to Section 5, an illustration of how the `Memcmp` code integrity has been protected against FIA. This code represents relatively simple control flows. For a complex firmware, the designer may need to cover just a sensitive section of the code (e.g. authentication function). It could be done by using the TE filter (cf. Chapter 5 of the TE standard [17]). It allows to specify the lower and higher addresses where packets need to be generated. Activating this functionality reduces drastically static data size and TV configurable modules area cost. This is due to the fact that there is less metadata in the TV's memory related just to this function. In our core implementation, the branch prediction feature was disabled. Enabling this option emits a specific packet after a discontinuity instruction with content defined by the TE standard. The TV could be configured to operate in this mode for CCFI verification. This is a perspective of our work.

8 Conclusion

In this paper, we propose a CCFI verification system based on the RISC-V Trace Encoder, a debug feature that allows to capture the execution path of a program. An additional feature is added to the TE mechanism in order to work in the CCFI mode. We demonstrate how discontinuity instructions and

BB are protected against FIA. The comparison of a computed signature of a BB at runtime with a pre-calculated signature guarantees the integrity property of a program's code. Compared to state-of-the-art solutions, our countermeasure does not generate performance overheads. Only hardware overheads are reported. Its implementation modifies neither the RISC-V compiler nor the user code nor the core's architecture. It is a modular, non-invasive and does not depend of the RISC-V core. In our future work, we aim to verify that the executed BB instructions are also unaltered within the core's pipeline. This is known as verifying the Control Flow and Execution Integrity (CFEI) of the program.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) **13**(1), 1–40 (2009)
2. Asanović, K., Patterson, D.A.: Instruction sets should be free: The case for risc-v. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146 (2014)
3. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proc. IEEE **100**(11), 3056–3076 (2012)
4. Chamelot, T., Couroussé, D., Heydemann, K.: Sci-fi: control signal code and control flow integrity against fault injection attacks. In: 2022 Design, Automation & Test in Europe Conference & Exhibition. pp. 556–559. IEEE (Aug 2022)
5. Colombier, B., Grandamme, P., Vernay, J., Chanavat, É., Bossuet, L., Laulanié, L.d., Chassagne, B.: Multi-spot laser fault injection setup: new possibilities for fault injection attacks. In: International Conference on Smart Card Research and Advanced Applications. pp. 151–166. Springer (2021)
6. Danger, J.L., Facon, A., Guilley, S., Heydemann, K., Kühne, U., Merabet, A.S., Timbert, M.: Ccfi-cache: A transparent and flexible hardware protection for code and control-flow integrity. In: 2018 21st Euromicro Conference on Digital System Design (DSD). pp. 529–536. IEEE (2018)
7. De Clercq, R., Götzfried, J., Übler, D., Maene, P., Verbauwhede, I.: Sofia: software and control flow integrity architecture. Computers & Security **68**, 16–35 (2017)
8. De Clercq, R., Verbauwhede, I.: A survey of hardware-based control flow integrity (cfi). arXiv preprint arXiv:1706.07257 (2017)
9. Elguibaly, F., El-Kharashi, M.: Multiple-input signature registers: an improved design. In: 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997. vol. 2, pp. 519–522 vol.2. https://doi.org/10.1109/PACRIM.1997.620315
10. Embench™: Open benchmarks for embedded platforms (Jan 2021), https://github.com/embench/embench-iot
11. Gonzalvez, A., Lashermes, R.: A case against indirect jumps for secure programs. In: Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering - SSPREW9 '19. pp. 1–10. ACM Press. https://doi.org/10.1145/3371307.3371314, http://dl.acm.org/citation.cfm?doid=3371307.3371314

12. Kiaei, P., Breunesse, C.B., Ahmadi, M., Schaumont, P., Van Woudenberg, J.: Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 319–324. IEEE (2021)
13. Lowrisc: Ibex documentation (Dec 2021), https://ibex-core.readthedocs.io/en/latest
14. OpenHW Group: Cv32e40p (Mar 2022), https://github.com/openhwgroup/cv32e40p
15. Pulp-platform: Pulpino: A small single-core risc-v soc (May 2019), https://github.com/pulp-platform/pulpino
16. Pulp-platform: Trace debugger for risc-v core (Nov 2020), https://github.com/pulp-platform/trace_debugger
17. RISC-V International: Efficient trace for risc-v (Nov 2020), https://github.com/riscv/riscv-trace-spec
18. Timmers, N., Spruyt, A., Witteman, M.: Controlling pc on arm using fault injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 25–35. IEEE (2016)
19. Wei, J., Pu, C.: Tocttou vulnerabilities in unix-style file systems: An anatomical study. In: FAST. vol. 5, pp. 12–12 (2005)
20. Werner, M., Unterluggauer, T., Schaffenrath, D., Mangard, S.: Sponge-based control-flow protection for iot devices. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 214–226. IEEE (2018)
21. Zeitouni, S., Dessouky, G., Arias, O., Sullivan, D., Ibrahim, A., Jin, Y., Sadeghi, A.R.: Atrium: Runtime attestation resilient under memory attacks. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 384–391. IEEE (2017)
22. Zgheib, A., Potin, O., Rigaud, J.B., Dutertre, J.M.: A cfi verification system based on the risc-v instruction trace encoder. In: 2022 25th Euromicro Conference on Digital System Design (DSD). IEEE (2022)