A CFI Verification System based on the RISC-V Instruction Trace Encoder

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

Authors' version
DSD 2022
https://doi.org/10.1109/DSD57027.2022.00067

A CFI Verification System based on the RISC-V Instruction Trace Encoder

Anthony ZGHEIB, Olivier POTIN, Jean-Baptiste RIGAUD, Jean-Max DUTERTRE
Mines Saint-Etienne, CEA, Leti, Centre CMP, F - 13541 Gardanne, France
zgheib@emse.fr, olivier.potin@emse.fr, rigaud@emse.fr, dutertre@emse.fr

***Abstract*—Control-Flow Integrity (CFI) is used to check a program execution flow and detect whether it is correctly executed and not altered by software or physical attacks. This paper presents a CFI verification system for programs executed on RISC-V cores. Our solution is based on the RISC-V instruction Trace Encoder (TE). The TE provides information about the execution path of the user program. Two approaches are proposed. One is consistent with the RISC-V TE standard. It permits to detect instruction skip attacks on function calls, on their returns and on branch instructions. The second implies an evolution of the RISC-V TE specifications to detect more complex fault models as the corruption of any discontinuity instruction. We implemented both approaches on a RISC-V core and simulated their efficiency against Fault Injection Attacks (FIA). Compared to existing CFI solutions, our methodology does not modify the user application code nor the RISC-V compiler.**

***Index Terms*—RISC-V, CFI, Trace Encoder, FIA, FPGA**

I. Introduction

Physical attacks are particularly effective threats to strike confidentiality, integrity or authenticity of a system. These attacks were firstly introduced by Boneh et al. in 1997 [1] where they showed how to attack RSA and Rabin signatures implementations. Fault Injection Attacks (FIA) are physical attacks injecting faults into a system in order to alter its intended behavior. The most common FIA techniques are described in [2]. These attacks could lead to skip or corrupt a vulnerable instruction in the user code to bypass system security features [3], extract a cryptographic key [4], bypass a PIN code [5] or have a privilege escalation [6]. Against FIA, a Control-Flow Integrity (CFI) [7] scheme verifies 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). The CFG is generated by statically analyzing the source code of the program. It represents the valid control flow changes in a normal program execution [8]. Most of existing approaches for CFI address Code-Reuse Attacks (CRA) such as Return-Oriented Programming (ROP) [9] and Jump-Oriented Programming (JOP) attacks [10] [11]. Hence, they only need to verify CFG integrity such as in [7] where their method checks both source and destination of indirect jumps. To avoid stateless approaches (approaches that do not associate call / forward jump to return / backward jump), stack canaries and shadow stacks are often used in combination for detecting bad return addresses [12]. CFI countermeasures are also used to detect FIA [13] as presented in section II. Compared to the state-of-the-art solutions, our approach ensures a CFI verification without modifying the user application code nor the compiler toolchain. It is designed for open source Instruction Set Architecture (ISA) RISC-V cores [14] [15]. In our study, the RV32I base integer instruction set is used [16]. This means that the data are represented on 32 bits and only instructions manipulating integer values are used. Because our solution is compatible with all RISC-V cores having the TE feature, it is illustrated in this paper on two cores implementation. Our CFI verification system is based on the standardized RISC-V Trace Encoder (TE) [17]. To the best of our knowledge, this is the first solution that uses the RISC-V TE for CFI verification. Our paper is divided as follows: Section II provides insights on existing CFI solutions. Sections III and IV describe our CFI verification methodology. Section V illustrates a FIA detection on a VerifyPin use case. Furthermore, Section VI details the hardware requirements for both approaches. Finally, we discuss and conclude on our proposed solution in Sections VII and VIII.

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

II. CFI Verification techniques

Software, hardware and co-design CFI verification systems exist. In our study, we are only interested in hardware / co-design approaches. A classification in two categories of these state-of-the-art CFI solutions is observed:

- Countermeasures extending the processor and / or its ISA.
- CFI monitoring modules connected to the processor.

A. Countermeasures extending the processor and / or its ISA.

a) Processor extension: In several works, an extension to the processor is made for CFI as in SOFIA [18] where it covers code injection, CRA such as JOP and ROP. It protects the software integrity, performs CFI, prevents execution of tampered code and enforces copyright protection. HAPEI [19] is inspired by SOFIA solution. It covers code injection, code reuse and fault injection attacks on instructions. These countermeasures involve a modification of the processor architecture without modifying the compiler. They also ensure the confidentiality of the user code by encrypting the code instructions and decrypting it before execution.

b) Custom ISA extension: This category presents solutions extending the processor's ISA with CFI dedicated instructions. HAFIX [20] takes part of this approach. It covers CRA exploiting Backward Edge Attacks (BEA). This solution

was tested on "bare metal" codes with Intel Siskiyou Peak core and LEON3 processor. Werner et al. [21] designed SCFP, a solution that ensures the confidentiality of a software IP and its authentic execution on Internet of Things (IoT) devices. It covers code-reuse, code injection and fault attacks on the code and control flow. Based on [21], Werner et al. [22] also designed a protection for the conditional branches by using encoded comparisons. De et al. [23] proposed FIXER. It's a solution implementing a co-processor to a RISC-V Rocket Chip core [24]. It detects code injection [25] and CRA such as buffer overflow and ROP attacks. Delshadtehrani et al. [26] implemented NILE, a co-processor to a RISC-V core that detects stack buffer overflow. Attacks skipping FIXER and NILE custom instructions prevent CFI verification as in HAFIX solution. Abdul Wahab et al. [27] developed a DIFT (Dynamic Information Flow Tracking) coprocessor. Their verification process is based on the ARM CoreSight debug component. It protects against buffer overflows, format-string attacks, SQL injection, cross-site scripting or data leakage. SCI-FI [28] solution is designed for control signal, code and CFI verification. It protects against FIA. From this category, all solutions modify the user code and compiler to insert the dedicated CFI instructions except for FIXER where its custom instructions are provided in a binary format before compiling the code (supported by the toolchain).

B. CFI *monitoring modules connected to the processor.*

This approach implies solutions connecting external blocks to the processor without ISA extension to verify the program CFI. CCFI-Cache countermeasure [13] is designed to check simultaneously Code and Control-Flow Integrity (CCFI). It verifies code and CFG (inter / intraprocedures), ensures protection against cyber and physical attacks. CCFI-Cache covers backward edge (ROP, buffer overflow), forward edge, code and fault injection. ATRIUM [30] 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 attacks, CRA, hardware fault attacks on instructions and TOCTOU (Time Of Check Time Of Use) attacks [31]. HCIC [32] is a hardware-based solution covering CRA such as JOP and ROP. It performs CFI checking on call, return and jump operations. All solutions in this category do not modify the processor's pipeline. Table I summarizes the average overhead costs of these countermeasures in terms of code size, performance, hardware area and power.

TABLE I
STATE-OF-THE-ART SOLUTIONS AVERAGE OVERHEADS.

Solution	Code Size (%)	Performance (%)	Hardware Area (%)	Power (%)
SOFIA [18]	141	110	28.2	N/A
HAPEI [19]	N/A	N/A	N/A	N/A
HAFIX [20]	N/A	2	N/A	N/A
SCFP [21]	19.8	9.1	N/A	N/A
FIXER [23]	N/A	1.5	2.9	N/A
NILE [26]	N/A	<3	15	26
DIFT [27]	<10	<335	<1	16.2
SCI-FI [28]	25.4	17.5	<23.8	N/A
CCFI-Cache [13]	<30	32	10	N/A
ATRIUM [30]	0	<22.7	<20	N/A
HCIC [32]	<0.8	<1	N/A	N/A

III. TE-BASED CFI VERIFICATION METHODOLOGY

A. Proposed TE-based CFI solution for RISC-V cores

This section presents a new CFI verification scheme based on the RISC-V Trace Encoder. The TE is an instruction tracer that compresses, at runtime, the sequence of instructions executed by the RISC-V core into trace packets. It is mainly used by designers for code debugging purposes. By having access to the program binary, they can reconstruct the program flow. Fig. 1 illustrates the existent solution using the TE and our CFI verification system. The use of a debug tool with the TE alone allows to dynamically reconstitute the program followed flow but it does not allow CFI verification. It constitutes a basis to perform it. Our work exploits the TE functionality by adding external blocks reading the TE packets in order to verify at runtime the program's CFI. A static analysis is done on the binary code where CFG metadata are generated. This information is stored in a memory connected to the CFI verification module: the Trace Verifier (TV). At runtime, the TV receives TE packets and refers to the static data to check the CFI of the program. The TV is connected to the TE. Hence, the RISC-V core remains intact. Our approach does not bring modifications to the user code nor to the RISC-V compiler. A description of our solution modules is provided in the following sections.

B. Trace Encoder

The so-called uninferable instructions are those applying a change in the Program Counter (PC) whose offset could not be determined from the compiled binary code like unpredictable jumps (e.g. return instructions). To follow the graph of the executed program, the TE reports the uninferable discontinuities in its control flow in forms of trace packets. The TE has a 3-stage pipeline to have a visibility of the current (J), previous (J-1) and next (J+1) instructions. Based on these instructions, a packet defined by the TE standard algorithm [17] is sent. The TE could be configured to activate or deactivate packets generation. It also has a filter to choose its operating address range by selecting the lowest and highest PC. A packet contains information about the path followed by the program since the last packet sent. Based on the instructions in the TE pipeline, packets are sent according to seven conditions:

- Based on the previous executed instruction (J-1):
  a) In case of an exception or a context change within the code with a discontinuity.
  b) An instruction with an uninferable PC discontinuity.

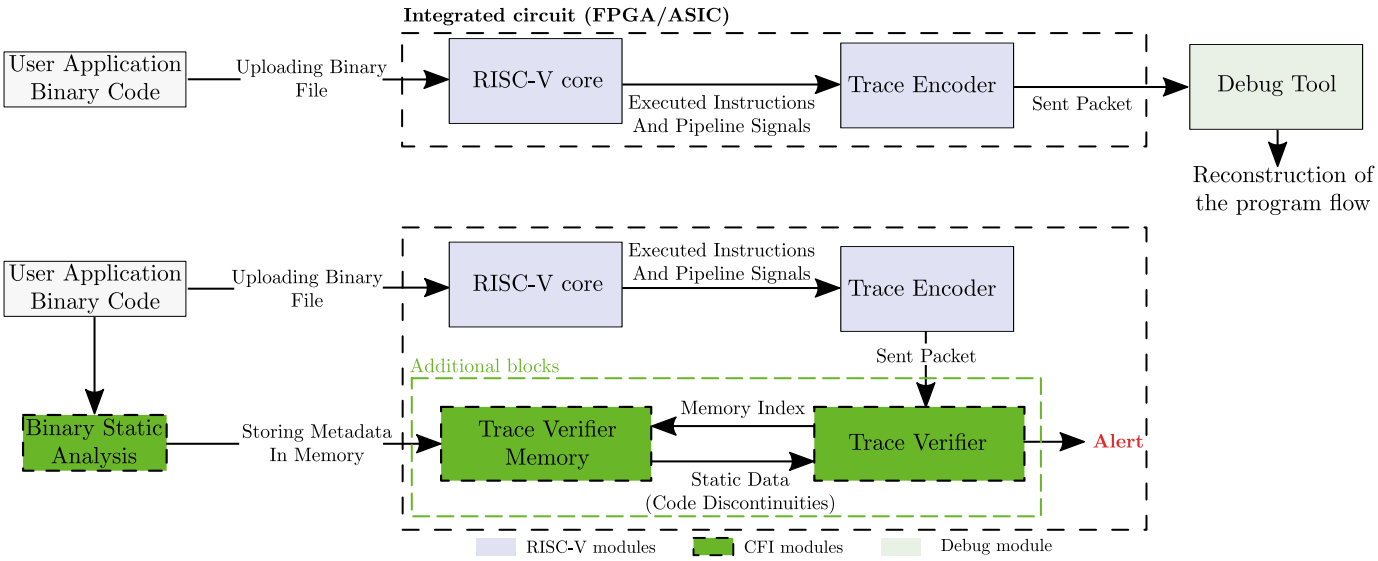

Fig. 1. A schematic of the RISC-V + TE (top), and the CFI verification system (bottom).

- Based on the current executed instruction (J):
  c) A first qualified instruction, a privilege / precise context change or the resync counter reaching its max resynchronisation value.
  d) Full branch map (number of branches=31) or mis-prediction case (when branch predictor enabled).
  e) An imprecise context change.
- Based on the next instruction (J+1):
  f) A simultaneous exception and retirement, or context change with discontinuity, or notify or resync counter has reached the maximum value and branch map not empty (need to be reported).
  g) An exception without retirement, a privilege change or precise context change and branch map not empty or unqualified instruction.

According to these conditions, packets are sent with a specific format (identifier). Referring to the specification [17], four packet formats are defined:

- **Format 0** is intended for optional efficiency extensions (like the counts of correctly predicted branches).
- **Format 1** reports a branch information when the TE branch counter reaches its maximum value. 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.
- **Format 3** is used for synchronization, reporting context and supporting information.

An example of a Format 1 packet is described below. It is sent after fulfilling one of the seven mentioned conditions before.

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

For instance, it can be sent after the execution of an uninferable discontinuity (cf. condition b). This packet indicates that ”n” branch instructions 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 the address of the next executed instruction following the uninferable discontinuity.

C. Static Analysis

A static analysis of the binary code by a custom program produces metadata. The objective of this analysis is to obtain a CFG description at disposal of the TV in order to verify the integrity of the execution path. These metadata concern all discontinuity instructions (Calls, Branch and Return Instructions), the 32-bit Program Counter (PC) and memory indexes corresponding to the upcoming discontinuities. These information are stored in a dedicated Random Access Memory (RAM). Its structure is shown in fig. 2. The monitoring of the control flow is done by following a sequence of addresses corresponding to the discontinuity instructions. Each of these instructions requires 96-bit memory place: 32-bit for its address, 32-bit for the instruction and 2*16-bit for memory indexes. In case of an unconditional discontinuity instruction, only one memory index is used to refer to the next discontinuity instruction. The second index is unused and fixed to ”FFFF”. In case of a conditional branch instruction, 2 memory indexes are stored (one referring to the next discontinuity instruction if the branch is taken and the other one in case of a non taken branch). As an example detailed in fig. 2, a jump instruction is stored at index ”1F” and points to index ”79” as the next discontinuity instruction. At this index, the branch instruction (BGE) is listed with its corresponding branch indexes.

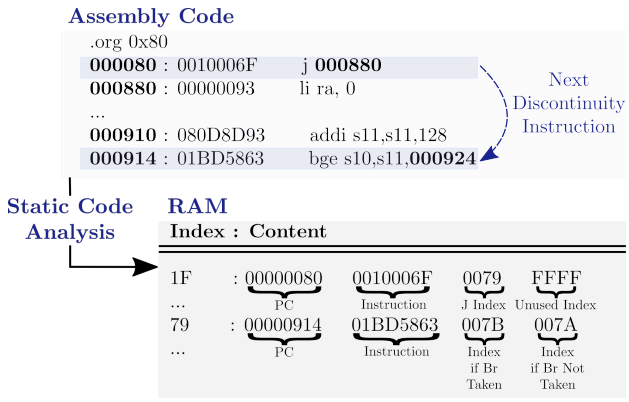

Fig. 2. Metadata content stored in RAM.

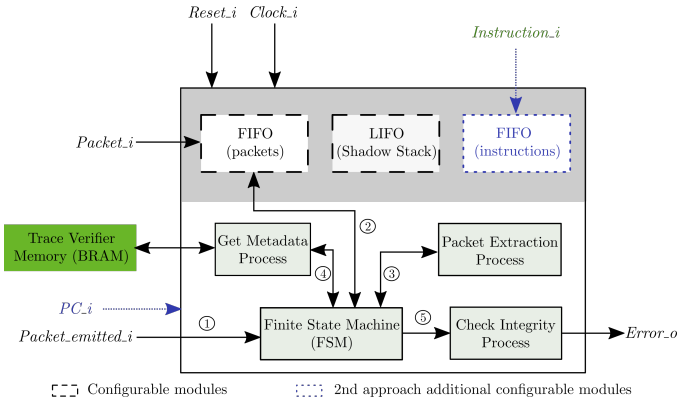

Fig. 3. Architecture of the TV.

D. Trace Verifier

The Trace Verifier is the core of our designed verification system. It wakes up as soon as a TE packet is emitted. As shown earlier, a sent packet contains the destination address (e.g. the address of the next valid executed instruction) and data depending on the packet format. It may contains the number of executed branch instructions with a branch map in case of Format 0 or 1. The TV navigates through the static data according to the packet content and checks the consistency with a faultless path. In case of a path inconsistency, having a difference between the TV obtained information and the packet content, an error flag will be raised reporting a CFG violation. According to fault model detection, we declined our methodology in two approaches: a first one strictly compatible with the TE standard and a second one with an enhancement of the specifications.

IV. CFI Solutions Description

A. Verification based on the RISC-V TE Standard [17]

In this approach, a packet is sent according to the conditions explained in section III-B. Fig. 3 shows the architecture of the TV module. The verification process starts when the TV receives a packet which activates its Finite State Machine (FSM) (1). The TV core retrieves the packet from its FIFO memory (2). Subsequently, in case of a Format 2 or 3 packet, the packet is decoded in order to extract the reported address (3). In case of Format 0 or 1, the branch number and branch map are also extracted. Having a starting address and packet information, a navigation through the RAM metadata is done to constitute the path followed by the program (4). The last step of the FSM is to check the address stored in the packet and the static address computed from the navigation process (5). If the addresses are not equal, an error flag will be raised.

a) TV behavior on BEA and on instruction skip attacks on function calls: With this approach, attacks changing return addresses (BEA) and instruction skip attacks on function calls are detected. A return instruction induces a packet containing the address of the next executed instruction. In the normal behavior, it must point out to the PC+4 of the function call. In case of a BEA, the sent packet will report a different address as shown in fig. 4. However, based on this packet, the TV will navigate the metadata, meet a function call and store its return index in a LIFO used as shadow stack (cf. fig. 3). In the verification process, the LIFO return address is compared with the packet address which reflects the FIA. In case of an instruction skip attack on a function call (cf. fig. 5), the next called function will send a packet containing branch number and / or the destination address different from the expected ones. Based on these information, the TV navigation process will lead to a different destination address and an error could be detected. The main advantage of the solution is that it is independent from the RISC-V core implementation as soon as a TE is present. The drawback is that the navigation through the RAM and verification process will only initiate after receiving a packet. Hence, if a FIA was done between two sent packets, a detection of this fault will be extremely late (3 clock cycles — more in case of a multicycle instructions — related to the packet retrieval + the extraction and check integrity process time + the navigation time). The navigation time is not negligible. It is the time required by the TV to reach the destination address. This time depends on the number of discontinuity instructions encountered in the static data between two instructions inducing packets to be sent.

b) Decrease latency by accessing the PC: A way to reduce the latency between the packet emission and verification step is to navigate through the metadata and constitute the path followed by the program before the TV receives the packet. For this improvement, the TV is adapted and connected to the PC of the RISC-V core (the PC is already

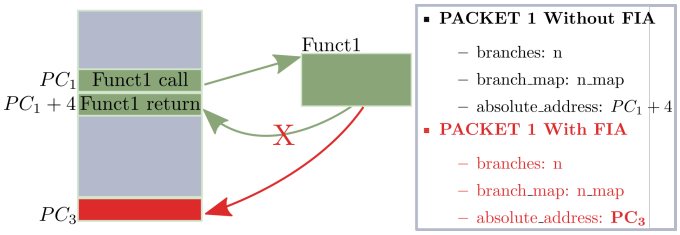

Fig. 4. BEA with a faultless packet and the packet sent due to the FIA.

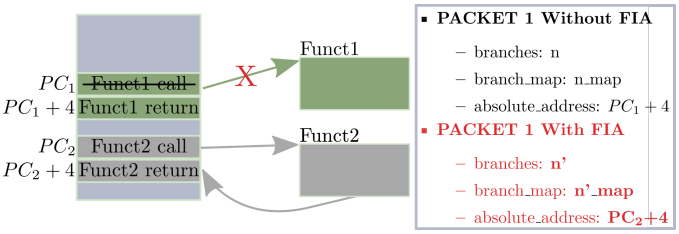

Fig. 5. Instruction Skip Attack with a faultless packet and the packet sent due to the FIA.

TABLE II
THREATS DETECTED BY THE SOLUTIONS:
SFC: SKIP ON FUNCTION CALLS
BEA: BACKWARD EDGE ATTACK
SBI: SKIP ON BRANCH INSTRUCTIONS
CDI: CORRUPTION OF A DISCONTINUITY INSTRUCTION
L: LATENCY.

	SFC	BEA	SBI	CDI	L
TV	✓	✓	(✗)	✗	--
TV with PC	✓	✓	✓	✗	-
TV for CFI	✓	✓	✓	✓	+

connected to the TE, no modification is made inside the core). Having the instruction address at each clock cycles allows an access to the metadata and a faster constitution of the path the program followed. This TV instance gives us the destination address and eventually the number of branches and branch map encountered before the packet emission. As soon as it is received, 4 clock cycles are needed to check the CFG. In addition, this improvement makes it possible to detect any instruction skip targeting a branch instruction because the next TE packet will report a number of branch instructions different from that given by the metadata. It has indeed no effect on the ability to detect the skip of call instructions or BEA.

c) Fault Model Limitation: With this approach, discontinuity instructions corruption is not covered which is a major advantage for an attacker. For instance, modifying the bit indexes [14:12] of a branch instruction - the funct3 field - from "000" to "001", changes the functionality of the instruction from Branch if Equal (BEQ) to Branch if Not Equal (BNE). This issue is resolved in the second approach of our TV.

B. Extending the RISC-V TE Standard

We also propose an extension of the TE — thanks to the open-source specification — to cover more fault models and to reduce the CFI verification time. This extension is compatible with the current TE. It consists in adding an option to send a packet after each (J-1) executed discontinuity instruction. In this version, the TE emits more packets for a same program compared to the first approach. The 32-bit executed instruction (already connected to the TE) is also retrieved where the discontinuity ones are stored in an additional FIFO as illustrated in fig. 3. An adjustment in the TV FSM is done in order to verify the stored instruction which permits to detect any discontinuity instruction corruption with the static data stored in the RAM (cf. fig. 2). Once a packet is received, 4 clock cycles are needed on average to verify the executed discontinuity instruction and destination address. The CFI verification is faster and more accurate since it is done after each discontinuity instruction. Table II sums up the threats covered by each solution. We refer respectively to "TV" the solution respecting the TE specifications and "TV with PC" the one respecting the standard and accessing the PC. Additionally, we refer to "TV for CFI" as the countermeasure detecting corruption of any discontinuity instructions. A qualitative comparison of our solution with the state-of-the-art CFI solutions is given in table III. Our approaches have zero impact on the user code and compilation process. Moreover, no modification inside the RISC-V pipeline is made. Compared to ATRIUM [30] which has similar CFI features, our CFI verification is performed on chip. Unlike our solutions that do not interact with the core, ATRIUM may stall the processor if the hash of the current instruction block is not completed and a new block arrives (28 cycles are needed to hash a block). The generated signature is sent at the end of the code region chosen by the trust verifier "vrf" for CFI verification. In contrast, "TV" and "TV with PC" do CFI verification after each uninferable instruction and "TV for CFI" after each discontinuity one. This reduces the verification latency.

V. EXPERIMENTAL STUDY ON A VERIFYPIN USE CASE

As an illustration of our CFI verification mechanisms, this section deals with an example of FIA detection on a VerifyPin use case from the FISSC collection [33]. This application aims to authenticate an user by comparing an user PIN to a card PIN code. L. Dureuil et al. [33] demonstrated that the VerifyPin version using hardened booleans and fixed-time loop as countermeasures has vulnerabilities to FIA. 4 attacks scenarios against authentication have been found. Two

TABLE III
COMPARISON OF OUR SOLUTION WITH RELATED WORKS.

Solution	No User Code Modification	No Compiler Modification	No Pipeline Modification	External Blocks Addition	No Execution Time Penalty	Backward Edge Protection	Forward Edge Protection	Code Confidentiality
SOFIA [18]	✗	✓	✗	✗	✗	✓	✗	✓
HAPEI [19]	✗	✓	✗	✗	N/A	✓	✗	✓
HAFIX [20]	✗	✗	✗	✗	✗	✓	✗	✗
SCFP [21]	✗	✗	✗	✗	✗	✓	✓	✓
FIXER [23]	✗	✓	✓	✓	✗	✓	✓	✗
NILE [26]	✗	✗	✓	✓	✗	✓	✗	✗
DIFT [27]	✗	✗	✓	✓	✗	✓	✗	✗
SCI-FI [28]	✗	✗	✗	✗	✗	✓	✗	✗
CCFI-Cache [13]	✗	✗	✓	✓	✗	✓	✗	✗
ATRIUM [30]	✓	✓	✓	✓	✗	✓	✗	✗
HCIC [32]	✗	✓	✓	✓	✗	✓	✓	✗
This Work	✔	✔	✔	✔	✔	✔	✘	✘

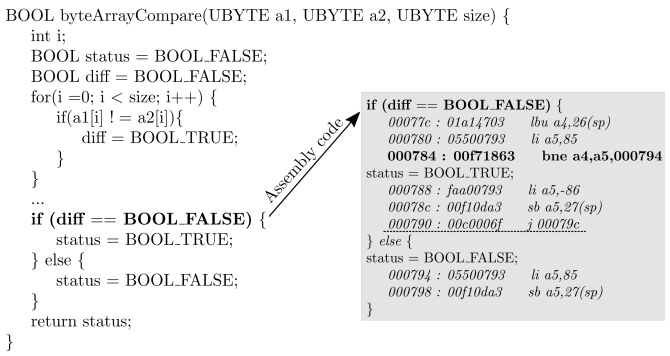

Fig. 6. ByteArrayCompare function in the VerifyPin Code.

of these attacks consist in inverting the condition of a sensitive branch instruction via a single fault injection. To illustrate one of these vulnerabilities, a FIA is simulated on one of the identified branch conditions as shown in fig. 6. This condition compares a variable "diff" to "BOOL_FALSE". If the condition is true then the variable "status" will indicate that there is no difference between the user and card PIN. Authentication could be granted. Inverting or skipping this instruction will affect "status" to "BOOL_TRUE" regardless of the user PIN. Hence, authentication could be granted with a wrong code PIN.

A. FIA Detection with the "TV"

Fig. 6 illustrates the "byteArrayCompare" function contained in the VerifyPin code [33]. Its return instruction (an uninferable instruction) induces a packet to be sent. Fig. 7 illustrates the content of the packet sent in a faultless behavior and the one sent when a skip attack is made on the identified branch instruction discussed in the previous section. Our TV expects 18 branches to be reported in a faultless code execution. Nevertheless, only 17 are reported in the Format 1 packet sent if the BNE is skipped (cf. fig. 7). The TV navigation process computes the expected address from the static data by following the branch map of the sent packet. In case of a faulted behavior, the navigation process will stop computing after the 17th branch and reports the address of the faulted instruction "0x000784". The address returned by the navigation process differs from the packet's address which leads to detect a CFI violation. An error flag is raised.

B. FIA Detection with the "TV for CFI"

In this approach, a packet is sent after each discontinuity instruction. For instance, a branch instruction will cause a packet emission. In case of the VerifyPin program, the "TV for CFI" expects a packet to be sent after the identified BNE instruction. Skipping this instruction will not cause an emission of a packet. Referring to fig. 6, the TV is expected to receive a packet related to the branch instruction that points on the address of "0x000784" with "0x00f71863" as instruction and "0x000788" or "0x000794" as destination address depending on the branch condition. In case of FIA, a packet will be sent at the next discontinuity instruction encountered by the core, the "j 00079C" instruction at "0x000790" address (cf. fig. 8). The fault can be detected by comparing the packet information and the TV expected metadata. If a corruption occurred within this instruction, for instance by changing the branch condition (e.g. opcode or immediate value), the comparison of this instruction with the one stored in the RAM also raises an error flag.

VI. Hardware Metrics

All our simulations target the Nexys Artix-7 FPGA board. This Integrated Circuit (IC) contains 33,650 logic slices. Each slice is composed of four 6-input LUTs, 8 flip-flops, multiplexers and carry units. In the following parts, a description of the hardware requirements of our system is provided.

A. Target Core

Our CFI solutions were implemented to a RISC-V IBEX core [34]. IBEX is a 32-bit open source RISC-V Central Processing Unit (CPU) core. It is a low power and small processor suitable for IOT applications. It has a 2-stage pipeline: Instruction Fetch (IF) and Instruction Decode / Execute (ID / EX). The implementation of this core requires **640** Slices. As our solutions are independent of the chosen RISC-V core, they can be implemented on any core compatible with the TE. For example, we can mention the CV32E40P (RI5CY) RISC-V core [29]. It is a 32-bit in-order 4-stage core. The pipeline is composed of an: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (EX) and Writeback (WB) stage. The RI5CY implementation requires **1171** Slices.

B. Trace Encoder

The TE module is extracted from the pulp-platform project [35]. Its software model could be found in [36]. Its implementation requires **184** slices. For the "TV for CFI", we made an enhancement to the standard in a way to send a packet after each discontinuity instruction. This improvement adds 2 extra slices in the TE module which therefore requires **186** slices.

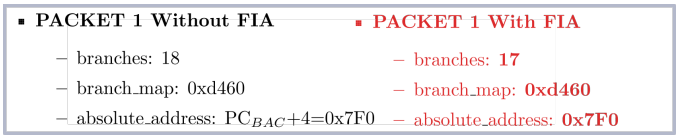

Fig. 7. Packet received by "**TV**" in case of an instruction skip attack on the VerifyPin identified branch instruction.

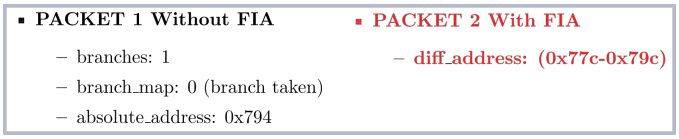

Fig. 8. Packet received by "**TV for CFI**" in case of an instruction skip attack on the VerifyPin identified branch instruction.

TABLE IV
TE AND TV CORE AREA (SLICES)

Solution	TE	TV core
TV	184	95
TV with PC	184	107
TV for CFI	186	100

C. Trace Verifier Components

The TV is composed of its core (FSM and processes), configurable modules (FIFO, LIFO) and a Block Random Access Memory (BRAM) as a Field-Programmable Gate Array (FPGA) implementation to store the static metadata. Table IV provides the area of the TE and TV core in slices. The "TV with PC" solution is the most demanding in terms of slices. This is due to the fact of receiving continuously the PC, computing the navigated branch number and dedicated branch map. The BRAM size depends on the user application code (i.e. the number of discontinuity instructions). Our CFI solutions were tested on several classic benchmarks like the tiny-AES, CRC32 [37], Memcmp, Memcpy and VerifyPin [33]. All the programs are compiled with the RISC-V GCC toolchain. The chosen architecture in the compilation process is the RV32I - 32bit RISC-V core. Two different compilations were done with no optimization "O0" and "O3" for code size and execution time optimization. Fig. 9 shows a diagram of the ratio between their code size and generated metadata on a log scale. The metadata represent **10%** to **26%** of the size of the application code. The total slice area for a TV could be represented as the sum of:

$$TV\_area = TV\_core + TV\_configurable\_blocks \quad (1)$$

where "TV_core" is the FSM and processes. The "TV_configurable_blocks" is the FIFO and LIFO modules. A memory block is a set of slices. Each slice can store a byte. 2 slices by return index, 4 slices by RISC-V instruction and 9 slices by TV packet are required. The following equation allows to size the TV configurable blocks area:

$$\begin{aligned} TV\_configurable\_blocks = (LIFO\_Depth * 2) + \\ (Trace\_FIFO\_Depth * 9) + \\ (Instr\_FIFO\_Depth * 4) \end{aligned} \quad (2)$$

In (2), "_Depth" represents the number of data the FIFO or LIFO needs to store. This parameter depends on the application user code. Fig. 10 illustrates the slice requirements of the configurable blocks for benchmarks compiled with "O3". The "TV" and "TV for CFI" solutions are presented. All benchmarks with "TV for CFI" are bigger because the discontinuity information are stored in the extra FIFO. As mentioned earlier, our TV could be implemented to any RISC-V core connected to the TE. In our simulations, the TE is connected to an IBEX core. In this case, the TV represents approximately **17%** in terms of slices with respect to the IBEX + TE requirements. On the CV32E40P, the TV represents **10%** of the (CV32E40P + TE) area.

VII. Approach Discussion

A. Trace Encoder

Our experiments covered the CFI of all the program. Nevertheless, the designer may need to cover just a sensible section of the code (e.g. authentication function). This could be done by using the filter of the TE. It allows us to specify the lower and higher addresses where we need packets to be generated. Activating this functionality reduces the static data and TV configurable modules area cost.

B. Trace Verifier

In the "TV" version, instruction skip attacks on branch instructions are detected under certain circumstances. It can be detected if no other branch instruction is executed instead in the user code, leading to the same destination address. In this case, the packet total number of branches is different and the TV is able to check that the path has been altered. Our "TV with PC" version could also be improved by detecting a FIA

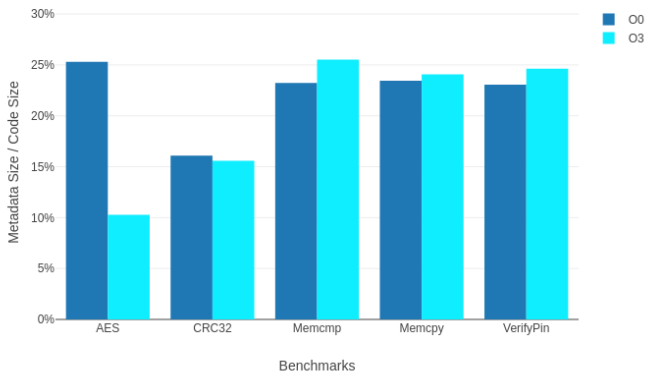

Fig. 9. Ratio between the metadata vs code size.

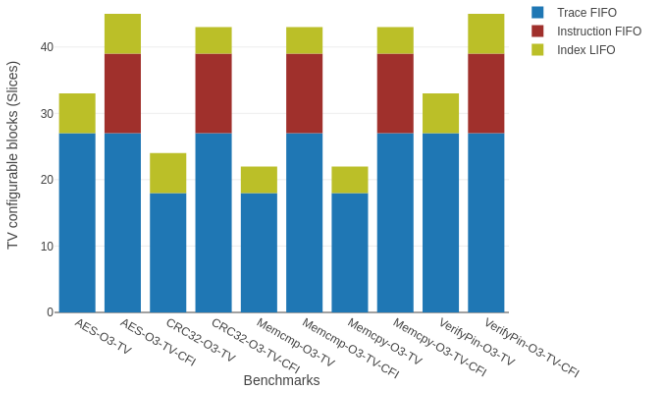

Fig. 10. TV configurable block slice requirements for "TV" and "TV-CFI" solutions.

before receiving a packet. This could be done by decoding the BRAM instruction corresponding to the pulled PC. This operation gives the address of the next expected instruction address. A comparison between this address and the pulled PC detects a FIA before the emission of a packet.

VIII. Conclusion

In this paper, we propose solutions to verify the CFI of application codes executed on RISC-V cores. Our verification modules are based on the RISC-V TE. Two approaches were developed where each one achieves a certain granularity and fault coverage for CFI protection. We demonstrated how discontinuity instructions within a code are protected against FIA. All our solutions do not generate software overheads. Only hardware overheads are reported. Their implementations do not make any changes to the RISC-V compiler nor the user code nor the core's architecture. They are modular, non-invasive and do not depend of the RISC-V core. Our perspectives are to upgrade the "TV for CFI" solution in order to ensure the verification of all executed instructions (Code Integrity) by checking the basic blocks. In a further step, we aim to verify that these instructions are unaltered within the core's pipeline. This is known as verifying the Control Flow and Execution Integrity (CFEI) of the program.

References

[1] D Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[2] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice,and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[3] N. Timmers, A. Spruyt, and M. Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[4] J. Breier, D. Jap, and C. Chen. Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103, 2015.

[5] P. Kiaei, C. Breunesse, M. Ahmadi, P. Schaumont, and J. Van Woudenberg. Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 319–324. IEEE, 2021.

[6] N. Timmers, and C. Mune. Escalating privileges in linux using voltage fault injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13, 2009.

[8] R. De Clercq, and I. Verbauwhede. A survey of hardware-based control flow integrity (cfi). *arXiv preprint arXiv:1706.07257*, 2017.

[9] H. Shacham. The geometry of innocent flesh on the bone. In *14th Conference on Computer and communications security*. ACM Press, oct 2007.

[10] M. Payer, A. Barresi, et al. Fine-grained control-flow integrity through binary hardening. In *12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[11] W. Arthur, B. Mehne et al. Getting in control of your control flow with control-data isolation. In *13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015.

[12] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Symposium on Security and Privacy, SP*, pages 48–62. IEEE Computer Society, 2013.

[13] J. Danger, et al. Ccfi-cache: A transparent and flexible hardware protection for code and control-flow integrity. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 529–536, Aug 2018.

[14] D. Patterson and A. Waterman, "The RISC-V Reader: An Open Architecture Atlas," Strawberry Canyon, 1st ed., 2017.

[15] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[16] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, Volume I User level Isa. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[17] RISC-V International, Nov 2020 https://github.com/riscv/riscv-trace-spec.

[18] R. d. Clercq, et al. Sofia: Software and control flow integrity architecture. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1172–1177, March 2016.

[19] G. Thomas, R. Lashermes, and H. Le Bouder. Hardware-assisted program execution integrity: Hapei. *23rd Nordic Conference on Secure IT Systems, Nov 2018, Oslo, Norway*, November 2018.

[20] L. Davi, et al. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[21] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard. Sponge-based control-flow protection for iot devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 214–226, April 2018.

[22] M. Werner, R. Schilling, T. Unterluggauer, and S. Mangard. Protecting risc-v processors against physical attacks. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, March 2019.

[23] A. De, A. Basu, S. Ghosh, and T. Jaeger. Fixer: Flow integrity extensions for embedded risc-v. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 348–353, March 2019.

[24] K. Asanović, et al. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[25] C. Hee Kim and J. Quisquater. Faults, injection methods, and fault attacks. *IEEE Design Test of Computers*, 24(6):544–545, 2007.

[26] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters*, 17(1):92–95, Jan 2018.

[27] M. Wahab, et al. A small and adaptive coprocessor for information flow tracking in arm socs. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018.

[28] T. Chamelot, D. Couroussé, and K. Heydemann. SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 564-567. IEEE, 2022.

[29] OpenHW Group, OpenHW Group CORE-V CV32E40P RISC-V IP, open-source hardware project, https://github.com/openhwgroup/cv32e40p

[30] S. Zeitouni, et al. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391, 2017.

[31] J. Wei, and C. Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005.

[32] J. Zhang, B. Qi, Z. Qin, and G. Qu. Hcic: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal*, 6(1):458–471, Feb 2019.

[33] L. Dureuil, et al. Fissc: A fault injection and simulation secure collection. In *International Conference on Computer Safety, Reliability, and Security*, pages 3–11. Springer, 2016.

[34] Lowrisc, IBEX documentation, Oct 2020, https://ibex-core.readthedocs.io/en/latest.

[35] Pulp-platform, Trace Debugger For RISC-V Core, Nov 2020 https://github.com/pulp-platform/trace_debugger

[36] Pulp-platform, RISC-V processor tracing tools and library, Nov 2020 https://github.com/pulp-platform/trdb

[37] J. Pallister, S. Hollis, and J. Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, 2013.