CIFER: Code Integrity and control Flow verification for programs Executed on a RISC-V core

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

Authors' version
HOST 23
https://doi.org/10.1109/HOST55118.2023.10133542

CIFER: Code Integrity and control Flow verification for programs Executed on a RISC-V core

Anthony Zgheib*†‡, Olivier Potin*, Jean-Baptiste Rigaud*, Jean-Max Dutertre*

*Mines Saint-Etienne, CEA Tech, Centre CMP, F - 13541 Gardanne, France
{zgheib, olivier.potin, rigaud, dutertre}@emse.fr
†*CEA Tech, Centre CMP, Equipe Commune CEA Tech - Mines Saint-Etienne, F-13541 Gardanne, France*
‡*Univ. Grenoble Alpes, CEA, Leti, F-38000 Grenoble, France*
anthony.zgheib@cea.fr

***Abstract*—Fault Injection Attacks (FIA) are powerful threats that can modify the intended behavior of a program running on a processor. 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 these attacks. Recent works have stated that developers do not sufficiently consider hardware specifications while designing software countermeasures. Moreover, most hardware and co-design CFI solutions do not cover the integrity of the processor microarchitecture. This paper presents CIFER, a Code Integrity and control Flow verification system for programs Executed on a RISC-V core. It ensures instruction execution in the core while checking the microarchitectural control signals. This is known as a Control-Flow and Execution Integrity (CFEI) approach. Our solution is built upon the RISC-V Trace Encoder (TE) which provides information about the execution path of the user's program. CIFER proposes an evolution of the TE standard and an analysis of the targeted core's architecture to monitor the pipeline control signals. The average hardware area overheads of our solution range from 35.1% to 55%. Compared to existing CFI and CFEI countermeasures, CIFER presents no performance costs. It does not modify the RISC-V Instruction Set Architecture (ISA), the compilation process nor the user code.**

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

I. Introduction

Fault injection attacks (FIA) are effective threats capable of modifying the behavior of the program running on a processor. The most common FIA techniques are described in [1]. These attacks could lead to skip or corrupt a vulnerable instruction in the user application code, in order to bypass system security features (e.g bypassing a PIN code [2]). Against these attacks, Control-Flow Integrity (CFI) [3] 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 evolutions in a normal program execution [4]. Surveys of hardware [4], [5], [6] and software solutions [7], [8] illustrate the relevant CFI solutions. Most of the state-of-the-art CFI solutions check the executed instructions at the fetch stage (first stage in a core's pipeline). In this case, FIA on memory containing these instructions or at the fetch stage could be detected. Yuce et al. [9] reported steps and mechanisms to inject faults on an embedded system. A fault can be injected into any micro-architectural block to affect the execution of an instruction in the code. Therefore, most of the existing countermeasures fail to catch such FIA on the core's mircoarchitecture. Thus, it is not enough to only verify the code integrity and/or the CFI of a program to guarantee its correct execution. The main challenge of our work is the detection of these FIA on binary instructions by protecting their execution from memory to the last stage of a RISC-V core.

Recently, Laurent et al. [10] discussed the effect of physical fault attacks on the microprocessor and also noted that developers use software countermeasures against these attacks without considering and analyzing the microarchitecture of the processor. Without taking this into account, the designed countermeasures have low effectiveness or overprotection and additional costs. As discussed in [10], the failure to consider both hardware and software specifications can lead to successful attacks, even in a protected system.

In this paper, we present CIFER, a Code Integrity and control Flow verification system that checks the correctness of all Executed instructions until the last stage of a RISC-V core's pipeline. CIFER follows a Control-Flow and Execution Integrity (CFEI) approach. Our solution is based on the RISC-V Trace Encoder (TE) [11]. This module is mainly designed for debug purposes. In our reseach, an exploitation of the TE functionality is done with the addition of a verification system in order to protect the execution integrity of a program.

Our paper is divided as follows: Section II provides insights on existing CFI and CFEI solutions. Sections III and IV describe our CIFER methodology and countermeasure. Section V shows its effectiveness against simulated FIA. Sections VI and VII report the hardware requirements of our solution and discussion. Finally, we conclude our paper in the last Section.

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

II. Related work

In this section, the most relevant CFI and CFEI verification solutions are presented.

A. CFI solutions

Some of these solutions extend the processors Instruction Set Architecture (ISA) with CFI dedicated instructions. FIXER [12] and NILE [13] take part of this approach. FIXER is a solution implementing a co-processor to a RISC-V Rocket Chip core [14]. It detects code injection [15] and Code Reuse Attacks (CRA) such as buffer overflow and Return-Oriented Programming (ROP) attacks [16]. Delshadtehrani et al. implemented NILE, a co-processor to a RISC-V core that detects stack buffer overflow. The newly introduced instructions are used to communicate with the dedicated co-processor for CFI verification.

Some countermeasures guarantee, in addition to the integrity, the confidentiality of the user code by encrypting the code instructions and deciphering it at runtime. From this category, we can cite SOFIA [17] and SCFP [18]. SOFIA 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. Werner et al. 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. A modification to the user code and compiler is required to insert dedicated CFI instructions.

Savry et al. implemented CONFIDAENT [19]. This solution consists in protecting both the data and instructions executed in the processor by encrypting them using a light masking scheme — ASCON [20]. CONFIDAENT detects FIA such as rowhammers [21] and glitches at the code or data level. This solution ensures the confidentiality and integrity of inputs and data during execution against CRA and stack overflow attacks. An extension of the ISA is needed to support this mechanism and a modification to the RISC-V compiler is required to insert these instructions.

Another approach consists of connecting external blocks to the processor without extending the ISA to verify the program's CFI like the solutions presented in CCFI-Cache [22] and ATRIUM [23]. Danger et al., in [22], 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, 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.

The listed countermeasures verify the control flow and/or code integrity of the program. In this case, all the executed instructions are verified and the integrity of the control flow is checked at the memory or first stage of the targeted core. Faults injected at the mircroarchitecture level are not detected. This forms a limitation to the CFI solutions and is solved by following a CFEI approach.

B. CFEI solution

Chamelot et al. developed SCI-FI [24], a countermeasure for control signal, code and control flow integrity against FIA. This solution was implemented on the CV32E40P processor [25]. It protects the execution of the program's instructions by checking the pipeline control signals with a fine-grained code and CFI mechanism. Its hardware system is composed of two modules: The CCFI and CSI.

The CCFI module provides CFI, code integrity and execution integrity for the front-end stages of the processor pipeline. This is possible by computing a runtime signature of the control signals at the core's decode stage. Verification instructions are added in the program code containing reference signatures. SCI-FI dedicated instructions are used to trigger signature verification or to load patch values supporting branch instructions.

The CSI module checks the execution integrity for the pipeline stages following the decode stage. It detects any change in the control signals constituting the pipeline state from their emission to their consumption stage. This 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. SCI-FI requires an extension of the RISC-V ISA in order to add the verification instructions.

Table I 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 VI.

C. Our contribution

Our verification solution CIFER checks the CFI, code and execution integrity of a program executed on a RISC-V core.

TABLE I
State-of-the-art solutions average overhead costs.

Solution	Code Size (%)	Performance (%)	Hardware Area (%)
FIXER [12]	N/A	1.5	2.9
NILE [13]	N/A	<3	15
SOFIA [17]	141	110	28.2
SCFP [18]	19.8	9.1	N/A
SCI-FI [24]	25.4	17.5	<23.8
CONFIDAENT [19]	<36	<36	N/A
CCFI-Cache [22]	<30	32	10
ATRIUM [23]	0	<22.7	<20
TE-based CFI [26]	0	0	<17
This Work	**0**	**0**	**<55**

Our solution consists in adding an additional verification system to the core. It detects software or physical attacks that derive the program execution from its normal behavior. CIFER is based on the program's CFG. 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). CIFER also detects alteration of the microarchitectural control signals, during the execution of instructions, that are the target of FIA. Our CFEI verification system is based on the RISC-V TE [11]. To the best of our knowledge, this is the first solution that uses the RISC-V TE for CFEI verification.

Compared to the SCI-FI solution which requires modification of the compiler back-end to insert the signature checks and patch values, our solution does not modify the compiler, the RISC-V ISA nor the user code. The following section details our solution methodology and architecture.

III. Our methodology

CIFER could be implemented on a RISC-V core, compatible with the TE [11], by applying the following three steps:

- A synthesis of the core pipeline: this step allows us to have a view on all internal modules and associated signals contained in its microarchitecture (to be protected against FIA).
- The generation of metadata related to the processors control signals following the intended execution of the program's instructions.
- The addition of an external hardware module — the Trace Verifier (TV) — to proceed to CFEI metadata verification.

Each step is described in detail in the next sections.

A. Core analysis

As an illustration, our CFEI solution is implemented on an IBEX core [27]. It is a 32-bit open-source RISC-V, low power core with a 2-stage pipeline suitable for IoT applications. Its architecture is illustrated in Fig. 1. A precise analysis has been performed on this core in order to select the microarchitectural signals linked to RISC-V instructions. These signals contribute to the verification of the program's execution integrity. Our research is dedicated for covering instructions respecting the RISC-V ISA RV32IM (32-bit instructions manipulating integers including multiplication and division operations). The compressed instructions (represented on 16 bits) were out of our scope. However, a 16-bit instruction is transformed to a 32-bit instruction at the fetch stage of the core but needs more signals to be analyzed. The six 32-bit RISC-V base instruction formats [28] are taken into consideration in our analysis. The decoding of a 32-bit RISC-V instruction leads to assigning values for dedicated signals in the core to ensure its execution. Our analysis has identified control signals to be relevant to the CFEI verification by following these 3 steps:

- Perform a synthesis of the core pipeline detecting all internal modules and signals.
- Evaluate the control signals linked to the instructions execution.
- Define the required unique signals for identifying each executed instruction.

As a result, a maximum of 22 control signals was deemed necessary to guarantee the CFEI of a program executed on an IBEX core. Fig. 1 shows the IBEX architecture with the control signals identified in its 2-stage pipeline. The size of these signals is equal to 47 bits. For other RISC-V architecture, a further analysis is required to elaborate the list of the relevant control signals.

B. Metadata generation

A custom program has been developed to parse automatically and statically the binary code in order to derive its CFG and constitute static metadata to be verified on runtime. This program is independent and does not take part of the RISC-V toolchain backend. It only requires the binary file containing the program instructions. The CFG shows all the legitimate paths that a program could follow. Our program reports all discontinuity instructions. They refer to calls, branch and return instructions. For each discontinuity instruction, metadata are generated. Each data element contains the PC (Program Counter) address, the discontinuity instruction with the indexes (addresses in the memory) of the following discontinuities. 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. For each instruction in the BB, our custom program extracts the values of the related control signals. With each discontinuity instruction in the metadata, a hash signature of instructions' control signals in a BB is computed. Algorithm 1 illustrates the pseudo-code of the static analysis process to build the program CFG and Control Signals Signature (CSS). A hash signature computation is made on the binary value of all the control signals of a BB using a Multiple-Input Signature Register (MISR) mechanism [29]. It starts from the BB first instruction's control signals until the end of the BB where the signature is generated. Fig. 2 illustrates an example of five BBs in a user code. For each instruction in a BB, its control signal values are generated and an MISR computation is done as described in Algorithm 1. Table II shows the signature calculation for the five BBs of Fig. 2. In this example, the Initialization Vector (IV) of the MISR is fixed to zero. The control signals of the first instruction of the BB will be xored to zero and will therefore generate a signature equal to the value of the signals of the instruction. The generated signature of a BB is stored within the discontinuity instruction pointing to the address of the first BB instruction. A runtime verification of this signature is bound to check the execution integrity of the instructions. The hash signature of `Basic Block 3` control signals starting at the address `0x3b4` is stored with the discontinuity instruction pointing to this address the `bne a4,a5,3b4` instruction at the address `0x3a4` when branch is taken. The signature of `Basic Block 2` is also stored

Algorithm 1 CFG Generation and CSS* calculation

Require: *Binary Code*
Ensure: *Discontinuity instructions metadata*
  for $i \leftarrow$ *program.begin* to *program.end* **do**
    #Discontinuity on Branch, Call or Return
    if *discontinuity instruction* **then**
      #Default Report
      report address at i;
      report instruction at i;
      calculate the CSS of the pointed BB*
      if *branch instruction* **then**
        report next discontinuity's address when branch taken and not taken;
        calculate the CSS of the two pointed BBs*
      else if *call instruction* **then**
        report next discontinuity's address;
        report return address and instruction;
        calculate the CSS of the pointed BB*
        calculate the CSS from return adress till next discontinuity;*
      else if *return instruction* **then**
        report return instruction and PC;
      end if
    end if
  end for

*CSS: Control Signals Signature

within this instruction as shown in Fig. 3 — representing the executed BB when the branch is not taken. The application reports for each discontinuity instruction, its address, its 32-bit instruction, the index (address in the memory) of the next attempted discontinuity instruction and hash signature of the pointed BB. For a branch instruction, two signatures are reported: the first when the branch is taken and the second when the branch is not taken. The stored signature is a prediction of the correct BB CSS when executed on core. A recalculation of the BB instructions' control signals is done at runtime and an additional hardware module the Trace Verifier (TV) is in charge of comparing it with the metadata (stored

TABLE II
SIGNATURE GENERATION

	Instructions	Control signals values	MISR calculation (Initial Vector=0)
Basic Block 1	01a14703 ... 00f71863	513802001c1c ... 073de0002010	513802001c1c ... **67c237716200**
Basic Block 2	faa00793 00f10da3 00c0006f	103c023f5410 013c0240361c 300168000302	103c023f5410 2144063e9e3c **7289647d3f7a**
Basic Block 3	05500793 00f10da3 01b14783 ... 03010113 00008067	103c0200aa10 003c0200aa10 513c02001e1c ... 110802006010 300168200002	103c0200aa10 21440641623c 13b40e82da64 ... 70007a163742 **523271b45da6**
Basic Block 4	01b14783 ... 03010113 00008067	513c02001e1c ... 110802006010 300168200002	513c02001e1c ... 6c8fa8e0d802 **6b2dd4598326**
Basic Block 5	00050793 ... 00f71e63	153c02000010 ... 073DE0003810	153c02000010 ... **7e0f859a1122**

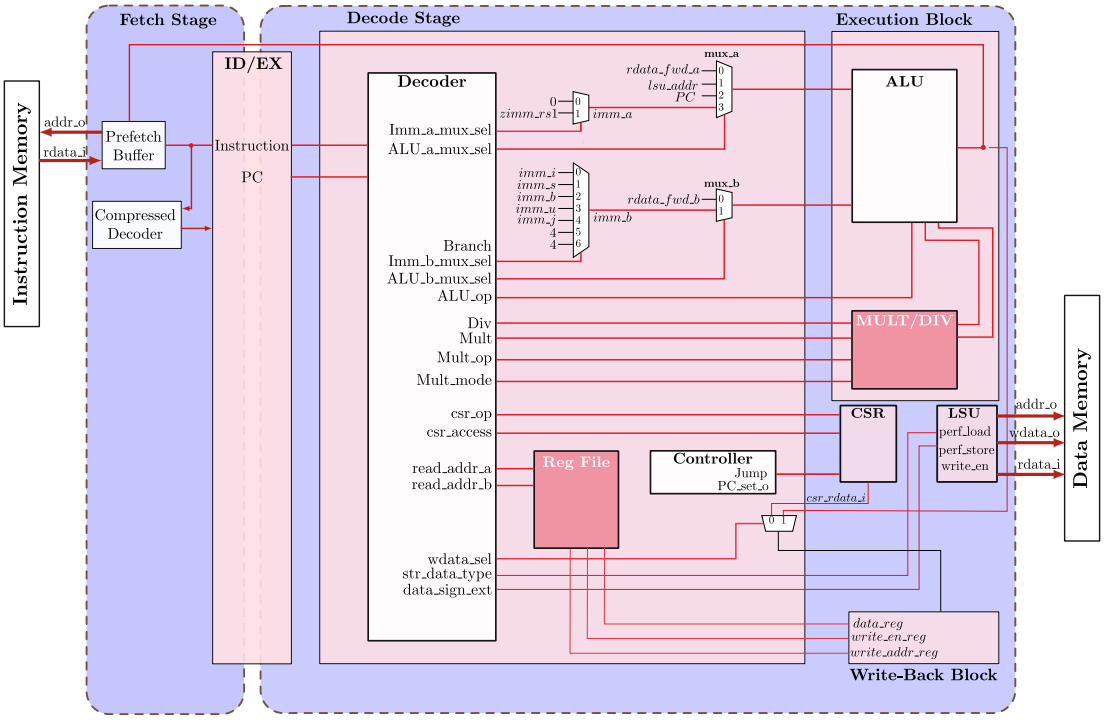

Fig. 1. IBEX core internal signals

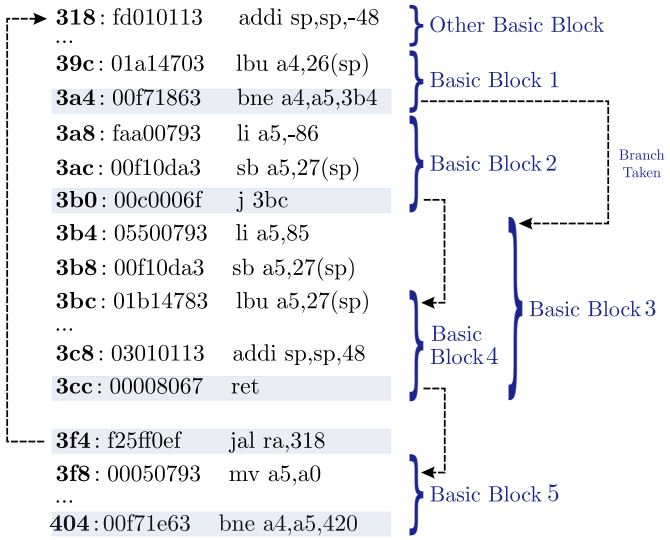

Fig. 2. Five BBs delimited by discontinuity instructions

in a memory).

C. Trace Verifier

The TV is an additional hardware module (cf. Fig. 4, 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) [11]. Based on the CFEI metadata (cf. Section III-B), the TV checks that the execution path of the program is included in its CFG . It also ensures the execution integrity of the user application code by verifying the BB CSS. An alarm is raised if an execution derivation has been detected. The following section describes in more details our countermeasure.

IV. CIFER

Our solution is composed of four modules: the RISC-V Trace Encoder, the signature computation module, the Trace Verifier and its memory to store the generated metadata. The verification system is depicted in the bottom part of Fig. 4.

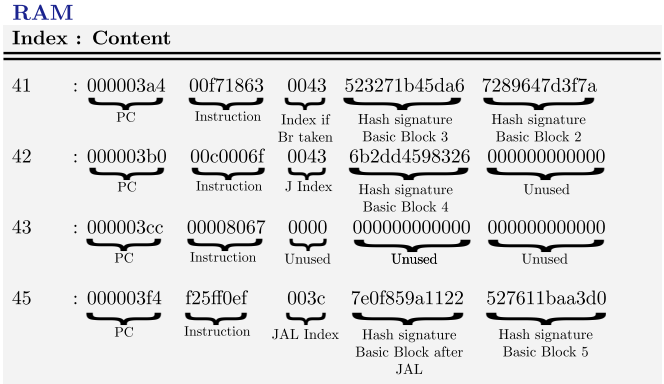

Fig. 3. Generated metadata content.

A. Trace Encoder

1) Overview: The TE is a RISC-V hardware module [11]. It is an execution flow tracer that compresses at runtime the sequence of discontinuity instructions 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. 4 (top). This module alone is used for debugging purposes and allows neither CFI nor CFEI verification. The TE has a 3-stage pipeline to store the current (I), previous (I-1) and next (I+1) instructions [11]. Based on these three instructions, a packet defined by the TE standard [11] 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 [11]. 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 CFEI. The other four conditions involve reporting the state of the core (as defined above), which does not cover the verification of CFEI 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.
- 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 is enabled).

The unpredictability imposes the sending of a packet in order not to lose the thread of the program executed flow.

2) TE-based CFI: A prior work [26] 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 [11]. 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. We found that the TE enrichment is interesting for a CFEI verification. CIFER is based on the TE sending a packet after each executed discontinuity. Compared to the TE-based solution, we do not require the 32-bit instruction and address buses for a CFEI verification.

B. MISR module

A hash signature computation with a MISR mechanism [29] is made on the 47-bit length control signals value of all the 32-bit instructions of a BB. This signature is then included in the packet transmitted to the Trace Verifier. An example of a Format 1 packet containing this signature is depicted below:

- **Packet 1:**
  - branches: n
  - branch_map: n_map
  - absolute_address: destination_address
  - **additional_field: control_signals_signature**

C. Trace Verifier Hardware Modules

As depicted in the bottom part of Fig. 4, our verification system is constituted by a memory (Trace Verifier Memory) and its core part (Trace Verifier).

1) TV Memory: The produced metadata are stored in a dedicated memory — Random Access Memory (RAM) — as illustrated in Fig. 4. Referring to Fig. 3, at index `42`, the 32-bit jump instruction `j 3bc` is stored with its 32-bit address `0x3b0`, the 16-bit index of next discontinuity and the 47-bit signature `0x6b2dd4598326` of the following BB. In case of a branch instruction (e.g. at index `41`), two 47-bit hash signature values are stored referring to the two possible branches. In total, each discontinuity instruction requires **174-bit** of metadata.

2) TV Architecture: Fig. 5 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.

3) TV FSM: The five steps listed in the **TV Architecture** are explicitly represented in Fig. 6. 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. 5. 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. 3 at index `41` and after the execution of the branch instruction `0x00f17863`, 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 `43` and the branch address is extracted from the metadata binary instruction. In the other case, the index will be incremented by 1 to reach the index `42` 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 CSS BB (contained within the previous metadata instruction) is also compared to the hash signature reported by the TE. The communication with the LIFO representing the "Shadow Stack" of our TV (cf. Fig. 5) is done when the BRAM navigation process points to a call or return instruction. After the FSM state `TE fields`

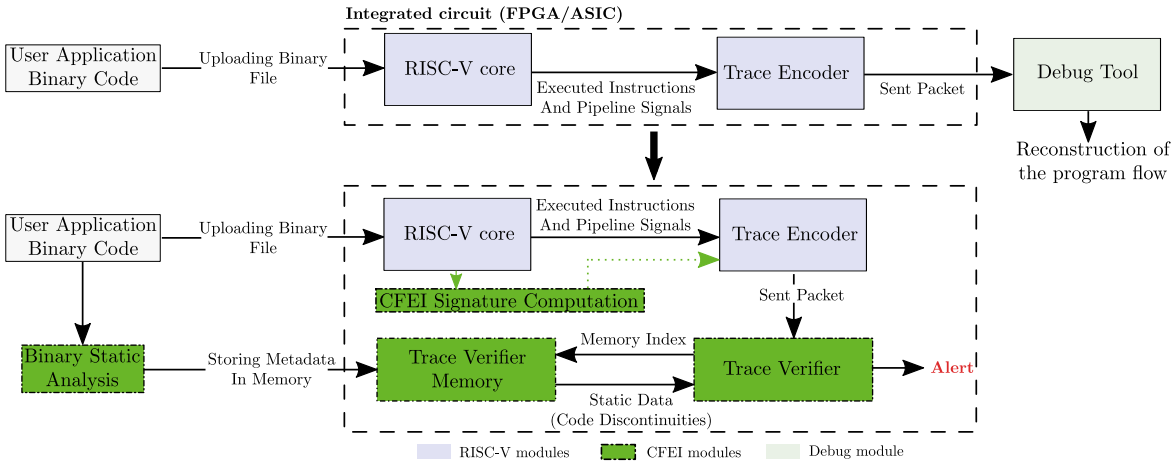

Fig. 4. A schematic of the RISC-V + TE (top) and CIFER solution (bottom).

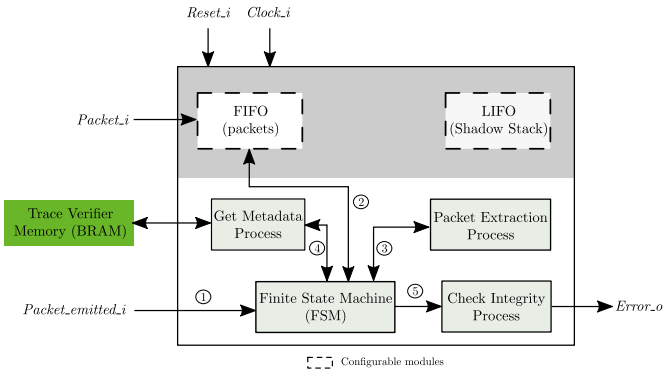

Fig. 5. Architecture of the TV.

`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 III-B). 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. 3. The

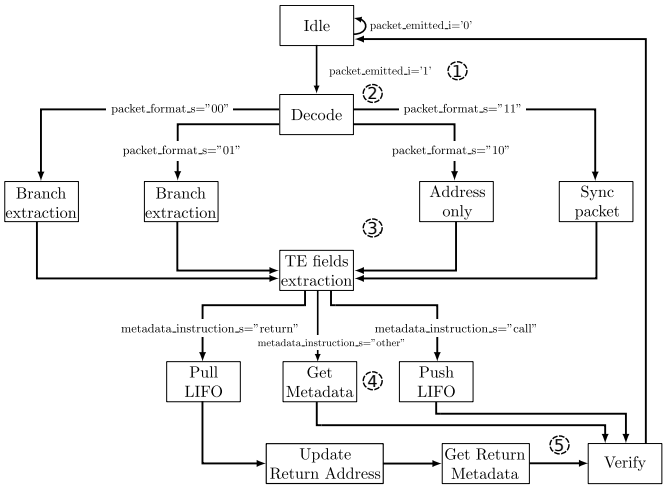

Fig. 6. FSM of the TV.

call instruction `jal ra,318` pushes the index `45` into the LIFO. After the execution of the return instruction at address `0x3cc`, the index is extracted from the LIFO, and then the return address is calculated by adding four to the call address `0x3f4`+4, which is equal to `0x3f8`. In addition, the BRAM index points to the call's index `45` incremented by one referring to the next planned discontinuity at index `46`. At the verification step, the calculated address `0x3f8` is compared to the TE reported address in addition to the signatures. The expected signature for the actual BB, containing the return instruction, is extracted from the previous discontinuity metadata (the branch instruction at address `0x3a4` or the jump instruction at address `0x3b0`). It is equal to `0x523271b45da6` if the branch is taken (`0x6b2dd4598326` if not and the jump instruction is executed). 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`. Our TV works in parallel to the RISC-V core. The six to eight clock cycles required to verify a packet are performed concurrently to the program execution.

V. FIA ON SELECTED BENCHMARKS

This section illustrates a FIA on a VerifyPIN use case and demonstrates how CIFER detects this fault.

A. VerifyPIN

This application aims to authenticate an user by comparing an user PIN to a card PIN code. L. Dureuil et al. [30] demonstrated that the VerifyPin version using hardened booleans and fixed-time loop as countermeasures has vulnerabilities to FIA on the executed instructions. A single fault injection can invert the condition of a sensitive branch instruction. Moreover , Yuce et al. [31] also stated that software countermeasures are not completely secure because their view of the microprocessor is limited to the ISA. FIA at lower level is possible and was experimentally demonstrated by [31] to broke software countermeasures by low-cost, single clock glitch injections. To illustrate one of these faults, a FIA is simulated on the control signals of an identified branch condition as shown in Fig. 7. This fault is identified by [10] on a branch instruction inverting the Arithmetic Logic Unit (ALU) operation test. The targeted branch 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 this instruction will affect "status" to "BOOL_TRUE" regardless of the user PIN. Hence, authentication could be granted with a wrong code PIN. The assembly instruction of this condition is shown as a `bne` instruction in the Fig. 2 at the address `0x3a4`.

```
BOOL byteArrayCompare(UBYTE a1, UBYTE a2, UBYTE size) {
    int i;
    BOOL status = BOOL_FALSE;
    BOOL diff = BOOL_FALSE;
    for(i =0; i < size; i++) {
        if(a1[i] ! = a2[i]){
            diff = BOOL_TRUE;
        }
    }
    if(i ! = size) {
        countermeasure();
    }
    if (diff == BOOL_FALSE) {
        status = BOOL_TRUE;
    } else {
        status = BOOL_FALSE;
    }
    return status;
}
```

Fig. 7. Targeted branch instruction in the VerifyPin code.

TABLE III
COMPARISON OF OUR SOLUTION WITH RELATED WORKS

Solution	No User Code Modification	No Compiler Modification	No Pipeline Modification	No Performance Overhead	Backward Edge Protection	Forward Edge Protection	Code Integrity	Code Execution Integrity	Code Confidentiality
FIXER [12]	✗	✓	✓	✗	✓	✓	✓	✗	✗
NILE [13]	✗	✗	✓	✗	✓	✗	✓	✗	✗
SOFIA [17]	✗	✓	✗	✗	✓	✗	✓	✗	✓
SCFP [18]	✗	✗	✗	✗	✓	✓	✓	✗	✓
CONFIDAENT [19]	✗	✗	✗	✗	✓	✓	✓	✗	✓
CCFI-Cache [22]	✗	✗	✓	✗	✓	(✗)	✓	✗	✗
ATRIUM [23]	✓	✓	✓	✗	✓	✗	✓	✗	✗
SCI-FI [24]	✗	✗	✗	✗	✓	✗	✓	✓	✗
TE-CFI [26]	✓	✓	✓	✓	✓	✗	✗	✗	✗
This Work	✔	✔	✔	✔	✔	✘	✔	✔	✘

B. FIA scenario

The `bne` instruction once retrieved from the instruction memory is decoded into several signals. A signal `ALU_op` indicates the required operation for the ALU as shown in Fig. 1. This signal is equal to `ALU_NE` where `NE` refers to a Not Equal operation in case of a `bne` instruction. The ALU is asked to compare the non equality of its A and B operands and report the result. For the branch instruction at address `0x3a4` — in charge of comparing the "diff" variable —, 2 non-equal operands (the case of a wrong pin) imply the branch to be taken and, thus, return a "BOOL_FALSE" as a "status" value. The authentication is then not approved. Changing the ALU operator from "ALU_NE" to "ALU_EQ" via a FIA, tests the equality of the operands values. In this case, the branch at address `0x3a4` is not taken and returns a "BOOL_TRUE" value for the status indicating an authentication. Fig. 8 shows the FIA simulation on the `ALU_OP` for the branch condition. A correct execution of the branch instruction at the decode stage `0x00f71863` is reported (cf. Fig. 2). However, due to the FIA, the ALU operator of this instruction is equal to "ALU_EQ". The simulated FIA changed the ALU operation from to "ALU_NE" to "ALU_EQ". This is a complex fault that requires physically 1 bit flip. From a software perspective, the IBEX has executed the `bne` instruction. However, from a hardware perspective the one actually executed is rather a `beq` instruction. An emission of a packet after the execution of (I-1) —the branch—, (I) and (I+1) instructions is send to the TV.

C. FIA detection

The sent packet after the execution of the branch instruction is processed by the TV as shown in our simulation of Fig. 9. The enumerated FSM steps (as described in Fig. 6) lead to the execution integrity verification of the BB containing the branch instruction. As the signature of the faulted execution `0x67c207716200` is different from the signature `0x67C237716200` expected by the TV, CIFER raises an error flag. Therefore, the FIA on control signals is detected. A comparison of our solution with relevant CFI and CFEI state-of-the-art solutions could be found in table III. Compared to the solution [26] which only verifies the program CFI by checking the discontinuity instructions at the core's decode stage, our solution additionally covers the integrity of the executed code by checking signatures of the pipeline control signals. Table I 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 ISA nor the compiler.

VI. SOLUTION METRICS

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.

A. Target Core

CIFER is implemented on an IBEX core [27]. 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. **25** additional were added to the core in order to connect the microarchitecture signals to the MISR module. In total, the core's architecture requires **670** slices.

B. Signals selector and MISR module

Depending on the executed instruction, related control signals are selected to calculate their hash signature. For each instruction, a signature of 47-bit is computed. Note that, the 47-bit signature computation module is designed to compute the signature of an instruction in one cycle. The total slice requirements for these modules is equal to **58** slices. Designed

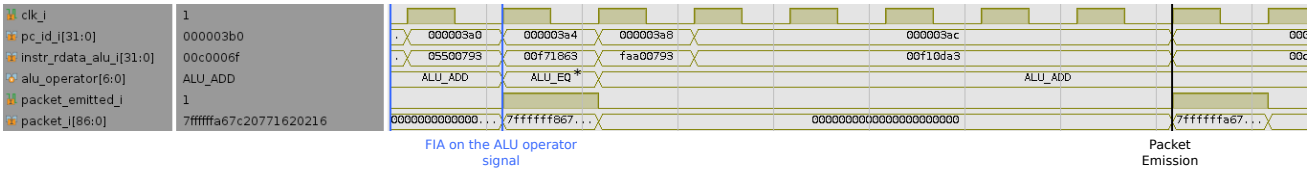

Fig. 8. Simulation of a packet emission due to the execution of the branch instruction.

for testability, a 47-bit MISR module offers a better protection (aka a small aliasing probability [32]) than a 47-bit CRC module or a 47-bit hash function against collisions.

C. Trace Encoder

The TE module is extracted from the pulp-platform project [33]. Its implementation needs **239** slices. To verify the CFEI 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. The packet size is increased by 17 bits to be a vector of 87-bit in order to contain the 47-bit signature, the 32-bit destination address, the 2-bit packet format and additional 6-bit depending on its format. These enhancements cost **46** slices while respecting the retro-compatibility of the TE. It can run in a normal [11], CFI or CFEI mode. In total, the TE requires **285** slices.

D. 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).

1) BRAM: CIFER was tested on several benchmarks from Pulpino project [34], Embench-IOT benchmarks [35] 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 **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 16% and 68%. 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 [35] contained the most discontinuity instructions. In total, 1188 lines (discontinuities) were needed for a optimization in O1. An index size of 12 was sufficient to address these lines. With this configuration, the memory implementation requires only 10 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 314 BRAM blocks and **174** 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.

2) TV core: It is composed of the FSM and processes. It requires **168** slices.

3) TV Configurable block: It represents the LIFO (shadow stack) and FIFO for storing the TE packets. 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. 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

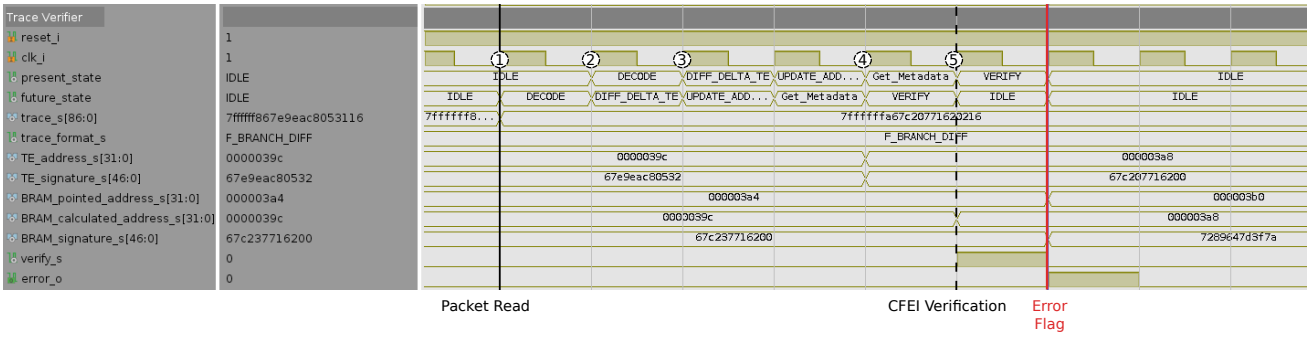

Fig. 9. Simulation of a packet's verification by the TV.

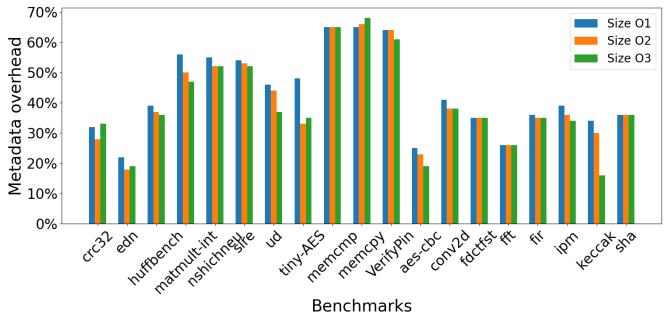

Fig. 10. Metadata overhead

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. Our TV does CFEI verification at runtime after receiving a packet from the TE . It needs 6 to 8 clock cycles to process a packet (cf. Section IV-C). 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.

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 taking part of CIFER (**46** slices, cf. Section VI-C), the hash

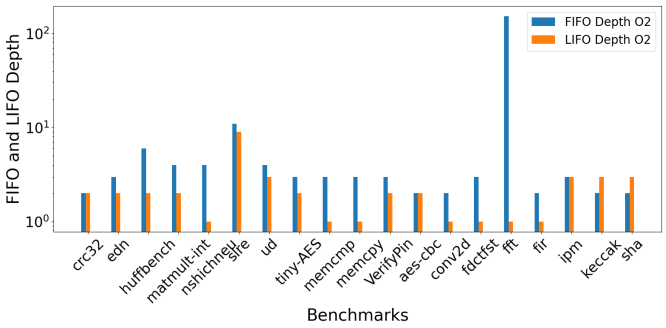

Fig. 11. FIFO and LIFO Depths

module (**58** slices and **25** for its connection) and TV (**357** slices for 16-bit LIFO index) is equal to **486** slices. When working with a 12-bit index, the TV only requires **182** slices. We have simulated the benchmarks used by [26] and a 12-bit index was enough to point all their discontinuity instructions. Therefore, compared to the TE-based CFI solution of [26] which only adds 17% in terms of slices over the IBEX + TE requirements, our solution requires **35.1%** (for an index less than 12-bit) and **55%** (for a 16-bit index). The area overhead is related to store 2*47 extra signature bits for each discontinuity instruction and their verification process. Each benchmark code was loaded into a **256-block** RAM connected to the IBEX core. Our metadata, stored in the TV memory, require **20-block** RAM for a 12-bit index. Therefore, the BRAM metadata overhead is equal to 7.81%.

VII. Discussion

In our work, CIFER has been implemented on an IBEX, a 2-stage pipeline RISC-V core connected to the TE (cf. Fig. 4). The TE receives a 47-bit signature of the instruction control signals to be included in its generated trace. In our implementation, the IBEX 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. Moreover, CIFER could be implemented to other RISC-V cores compatible with the TE. However, the study of the targeted architecture is required. The methodology steps defined in Section III should be followed for other RISC-V cores. This permits choosing the control signals adapted to the architecture and adapting the size of the MISR module and TE trace. The execution integrity of RV32IM instructions has been protected on an IBEX core. A study of the signals of the compressed instructions could also be done for possible integration into CIFER. This is a perspective of our work.

VIII. Conclusion

This paper presents CIFER, a Code Integrity and control Flow verification system for programs executed on a RISC-V core. This solution is based on the RISC-V Trace Encoder, a debug feature that allows to capture the execution path of a program. A signature calculation of microarchitectural control signals is linked to the TE mechanism in order to work in the CFEI mode. We demonstrate how program's instructions execution are protected against FIA. The comparison of a computed signature of microarchitecture control signals, at runtime, with a pre-calculated signature guarantees the execution 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. CIFER's overage area overheads range from 35.1% to 55%. Its implementation does not modify the RISC-V ISA, compiler nor the user code. In our future work, we aim to enhance CIFER to handle interruptions and core exceptions.

REFERENCES

[1] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov. 2012, Conference Name: Proceedings of the IEEE.

[2] P. Kiaei, C.-B. Breunesse, M. Ahmadi, P. Schaumont, and J. v. Woudenberg, "Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, ISSN: 0738-100X, Dec. 2021, pp. 319–324.

[3] M. Abadi, M. Budiu,. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, 4:1–4:40, Nov. 6, 2009. https://doi.org/10.1145/1609956.1609960.

[4] R. D. Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (CFI)," p. 27, 2017.

[5] S. Kumar, D. Moolchandani, and S. R. Sarangi, "Hardware-assisted mechanisms to enforce control flow integrity: A comprehensive survey," *Journal of Systems Architecture*, vol. 130, p. 102 644, Sep. 1, 2022. https : / / www . sciencedirect . com / science / article / pii / S1383762122001643 (visited on 09/15/2022).

[6] S. Tauner and M. Telesklav, "Comparative analysis and enhancement of CFG-based hardware-assisted CFI schemes," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, 58:1–58:25, Sep. 22, 2021. https://doi.org/10.1145/3476989 (visited on 09/15/2022).

[7] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, 2017.

[8] N. Burow, S. A. Carr, J. Nash, *et al.*, "Control-flow integrity: Precision, security, and performance," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, pp. 701–714, Aug. 1, 2017. http://arxiv.org/abs/1602.04056 (visited on 09/28/2022).

[9] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, Jun. 2018. http://arxiv.org/abs/2003.10513 (visited on 09/19/2022).

[10] 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*, vol. 71, p. 102 862, Nov. 1, 2019. https : // www . sciencedirect . com / science / article / pii / S0141933118304745.

[11] RISC-V, *Working draft of the RISC-v processor trace specification*. https://github.com/riscv/riscv-trace-spec (visited on 11/29/2020).

[12] A. De, A. Basu, S. Ghosh, and T. Jaeger, "Hardware assisted buffer protection mechanisms for embedded RISC-v," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4453–4465, Dec. 2020, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[13] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, "Nile: A programmable monitoring coprocessor," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 92–95, Jan. 1, 2018. http : //ieeexplore.ieee.org/document/8219379/ (visited on 10/26/2022).

[14] K. Asanovic, R. Avizienis, J. Bachrach, *et al.*, "The rocket chip generator," p. 11,

[15] C. H. Kim and J.-J. Quisquater, "Faults, injection methods, and fault attacks," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 544–545, Nov. 2007, Conference Name: IEEE Design & Test of Computers.

[16] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," p. 29,

[17] R. de Clercq, J. Gtzfried, D. bler, P. Maene, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," *Computers & Security*, vol. 68, pp. 16–35, Jul. 1, 2017. https://www.sciencedirect.com/science/article/pii/S0167404817300664 (visited on 09/27/2022).

[18] 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)*, Apr. 2018, pp. 214–226.

[19] O. Savry, M. El-Majihi, and T. Hiscock, "Confidaent: Control FLow protection with instruction and data authenticated encryption," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Aug. 2020, pp. 246–253.

[20] I. T. L. Computer Security Division. (Jan. 3, 2017). Round 2 - lightweight cryptography | CSRC | CSRC, CSRC | NIST, https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates (visited on 10/19/2022).

[21] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[22] J.-L. Danger, A. Facon, S. Guilley, *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)*, Aug. 2018, pp. 529–536.

[23] S. Zeitouni, G. Dessouky, O. Arias, *et al.*, "ATRIUM: Runtime attestation resilient under memory attacks," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, ISSN: 1558-2434, Nov. 2017, pp. 384–391.

[24] T. Chamelot, D. Courousse, 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)*, Antwerp, Belgium: IEEE, Mar. 14, 2022, pp. 556–559. https://ieeexplore.ieee.org/document/9774685/.

[25] M. Gautschi, P. D. Schiavone, A. Traber, *et al.*, *Near-threshold RISC-v core with DSP extensions for scalable IoT endpoint devices*, Feb. 2017. https : / / ieeexplore . ieee . org / document / 7864441 (visited on 09/28/2022).

[26] A. Zgheib, O. Potin, J.-B. Rigaud, and J.-M. Dutertre, "A cfi verification system based on the risc-v instruction trace encoder," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 456–463.

[27] *Ibex RISC-v core*, original-date: 2017-08-08T12:16:36Z, Sep. 27, 2022. https://github.com/lowRISC/ibex (visited on 12/09/2019).

[28] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[29] F. Elguibaly and M. El-Kharashi, "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, Aug. 1997, 519–522 vol.2.

[30] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. d. Choudens, "Fissc: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*, Springer, 2016, pp. 3–11.

[31] B. Yuce, N. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," presented at the Proceedings - 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, 2016, pp. 47–58.

[32] D. Pradhan, S. Gupta, and M. Karpovsky, "Aliasing probability for multiple input signature analyzer," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 586–591, Apr. 1990, Conference Name: IEEE Transactions on Computers.

[33] Pulp-platform, *Trace debugger for risc-v core*. https://github.com/pulp-platform/trace_debugger (visited on 11/01/2020).

[34] A. Traber, F. Zaruba, S. Stucki, *et al.*, "Pulpino: A small single-core risc-v soc," in *3rd RISCV Workshop*, 2016.

[35] D. Patterson, J. Bennett, P Dabbelt, C Garlati, G. Madhusudan, and T Mudge, *Embench: A modern embedded benchmark suite*, 2021.