Grenoble INP **UGA**
Master CybserSecurity **Year 2024-25**

Advanced Security (L. Mounier)

Written Exam - Thursday January the 23rd, 2025

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Electronic devices are forbidden.

This exam contains two distinct parts: 2 exercises and some questions on a research paper.

Exercise 1: code obfuscation (∼ 4 pts)

We consider the following C code (left) and the two (in)equalities (right):

```
#include <stdio.h>

int main() {
  int i ;
  scanf("%d", &i);

  if (i > 0) {
    printf("positive\n");
  } else {
    printf("negative\n");
  }
}
```

$$\forall x.\ (x^2 + x)\ mod\ 2 \quad = \quad 0 \tag{1}$$

$$\forall x, y.\ (7 \times y^2 - 1) \quad \neq \quad x^2 \tag{2}$$

Q1. Use these two (in-)equalities to add two *opaque predicate* in this code (at the source level) in order to obfuscate – as much as you can – its control-flow graph (CFG) while preserving its nominal behavior. Give the resulting CFG. You can use the `rand()` function to generate a random integer.

Q2. Assume you are a reverse engineer, starting from the binary code and aiming to retrieve the non-obfuscated code, and not aware of (in)equalities (1) and (2). Explain in a few sentences how could you proceed in practice to detect the presence of these opaque predicates?

Q3. Give two examples of other obfuscation techniques you are aware of (or you can think about).

Exercise 2: fault injection (∼ 6 pts)

The following C code receives a sequence of 8 bytes from a network (7 bytes of payload plus one byte for a hash value) and delivers it to some upper layer application. However, the execution plateform on which this code is ran may be targeted by *fault injection attacks*[1]. To (partially) detect such attacks the developer added a lightweight counter-measure by re-computing the hash value and checking its correctness. In case this check fails a `NULL` value is delivered (instead of the corrupted buffer).

[1]the network is supposed to be reliable, the value received at lines 5 and 10 are therefore correct.

The expected security property is therefore that *the value NULL should be returned when the buffer is corrupted*

```
 1 char buf[8] ;
 2
 3 void receiver() {
 4 char x, hash=0 ;
 5 receive(x) ;  // receive a byte from the network
 6 for (int i=0 ; i<7; i++) {
 7     buf[i]=x;
 8     hash=hash+x ;
 9 }
10 receive(buf[7]); // receive the last byte from the network
11 if (buf[7] == hash)  // initial hash value stored in buf[7]
12   deliver(buf) ;
13 else
14   deliver(NULL) ;
15 }
```

Q1. Against which fault model is this code (partially) protected? Give an example of detected (resp. not detected) attack.

Q2. Add some (source level) countermeasures to protect function `receiver` against 1-fault **data mutations** on variables `x` and `i` at lines 6, 7 and 8. With this new protections is it still useful to compute and check the hash?

Q3. Add some (source level) countermeasures to protect function `receiver`, the one proposed in Q2, against 1-fault **test inversions**. Give two distinct examples of successful 2-faults attacks on this new version.

Q4. Give a 2-fault attacks (using data mutation and/or test inversion) leading to a buffer overflow. What could be the consequence of such an attack with respect to the security property?

Questions on a research paper. (∼ 10 pts)

Read the paper given in appendix answering the following questions as long as you read it. When answering these questions you should not copy entire sentences from the paper but rather illustrate your point with examples and comments from your own.

Q1 [section I].

1. Do you know why data execution prevention (DEP) is *rarely deployed on embedded plateforms or micro-controllers*?
2. Briefly summarize the main contribution of the paper, namely:
   - what problem does it address?
   - how this problem is proposed to be solved?

Q2 [section II-B].

1. Explain why ROP, COP and JOP attacks only rely on *code pointer corruption.*
2. According to you knowledge of Rowhammer attacks, tell why it is no so obvious to make this attack successful.

Q3 [section II-C].

1. Is it always the case that the target intra-procedural jumps can be determined at compile-time? (think about the limitations of static disassembly . . .)
2. Why do we need a precise *pairing* of call and return address? Give a small code example illustrating this point.
3. Give a small code example (at the source level) where the address of an indirect jump cannot be resolved at compile time (think about function pointers).

Q4 [section III]. Explain the results given in Table I for these two columns: $W \oplus X$ (aka DEP) and Intel CET.

Q5 [section IV-A].

1. Explain why each block of metadata should be *perfectly aligned in memory* with its corresponding basic block.
2. Why the shadow stack should not be accessible from the main processor?

Q6 [section IV-B].

1. Why metada need to be computed *after* compiler optimization?
2. what is the problem with *branch prediction*

Q7 [section IV-C]. Is it the case that an attack could be detected in the middle of a basic block? Or are they detected only at the end of the block (in state EndBB)?

Q8 [section IV-C, last paragraph]. Why is it important to protect the interupt service routine (ISR)?

Q9 [section V-D, last paragraph]. Explain precisely the content of Table II

Q10 [More general questions] Is the proposed solution robust to the following fault models, explaining your answer:

1. data corruption, for instance after a buffer overflow
2. test inversion (due to a physical attack)
3. double fault injection
4. fault injection in the CFI checker (due to a physical attack)

CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity

Jean-Luc Danger [1,2], Adrien Facon [2,4], Sylvain Guilley [2,4], Karine Heydemann [3],
Ulrich Kühne [1], Abdelmalek Si Merabet [1] Michaël Timbert [1,2*],
[1]*Institut Mines-Télécom* Paris, France
{jean-luc.danger, sylvain.guilley, ulrich.kuhne, abdelmalek.si-merabet, michael.timbert}@telecom-paristech.fr
[2]*Secure-IC* Paris, France
{jean-luc.danger, sylvain.guilley, adrien.facon, michael.timbert}@secure-ic.com
[3]*Université Pierre et Marie Curie* Paris, France
karine.heydemann@lip6.fr
[4]*École normale supérieure (ENS), Département d'Informatique, CNRS, PSL University*, Paris, France

***Abstract*—In this paper we present a hardware based solution to verify simultaneously Code and Control-Flow Integrity (CCFI), aiming at protecting microcontrollers against both cyber- and physical attacks. This solution is non-intrusive as it does not require any modification of the CPU core. It relies on two additional hardware blocks external to the CPU: The first one – called CCFI-cache – acts as a dedicated cache for the storage of information to check the code and control-flow integrity, and the second one – CCFI-checker – performs control-flow and code integrity verification. Based on a RISC-V platform implementation, we show that the proposed scheme is able to perform online CCFI validation at the price of a small hardware area overhead and doubling the size of the `.text` section. In most cases, the impact on the run-time performance is on average 32 percent, offering for the first time a generic and practical hardware-enabled cyber-security solution.**

***Index Terms*—Hardware security, cybersecurity, Control Flow Graph, Control-Flow Integrity, Code Integrity, Instruction Hashing, Hardware protection**

I. Introduction

Cyber-attacks are known to be a major threat for all kinds of systems ranging from cloud-servers to embedded devices and industrial control systems. This threat will become even bigger with the proliferation of the *Internet of Things* (IoT). Control-flow hijacking attacks try to take over the target machine with the goal to execute malicious code. A common attack vector is to take advantage of code weaknesses to provoke buffer overflows. This can be used to either directly inject code or – in presence of a protection preventing the processor from executing data segments – to change the return address, which is known as code-reuse attacks [1]. While in principle, code weaknesses enabling buffer overflows can be eliminated by the use of static analysis, memory-safe programming languages or mechanically proved programs, it is very challenging to ensure that no control flow hijacking is possible. Indeed, there is a gap between proven code and code generated by tools, which are themselves seldom proven. Moreover, formal approaches require time and expertise that prevent their use in industry pushed by time-to-market constraints, legacy code reuse, and high competition.

* Michaël Timbert is the corresponding author.

Control-flow integrity (CFI) refers to protections against control-flow hijacking and was introduced in Abadi's seminal paper [2]. The idea is to verify at run-time by a monitor process or by dedicated hardware that the correct control flow is respected. A common specification of the control flow is given by the static *control flow graph* (CFG) of the application, which can be determined at compile-time. We differentiate *intra-procedural* CFI – protecting branches and jumps – from *inter-procedural* CFI, which considers function calls and returns.

In the recent years, many CFI approaches have been proposed. Software-only approaches that offer full CFI protection suffer from a high performance overhead [3]. Some software implementations focus only on specific protections in order to reduce this overhead. Hardware-based solutions range from lightweight solutions – ensuring only some types of control transfer (such as a so-called *shadow stack*) or reducing the amount of reusable code by marking valid call/jump destinations – to solutions covering all control transfers that can be determined statically at compile-time, at link-time, or at load-time of the application [4], [5]. Unfortunately, such approaches either do not cover all control transfers or they require a significant modification of the CPU, which prevents them from being deployed in practice due to either the huge amount of work required for validating a modified processor or the use of off-the-shelf processor cores. This is why we target a *non-intrusive* solution that does not modify the CPU core.

Most CFI approaches assume that the code cannot be modified, due to the presence of widely used *data execution prevention* (DEP) protections. Such a protection is commonly present on high performance processors but rarely deployed on embedded platforms or micro-controllers. Furthermore, different threats may invalidate this assumption: There exist physical attacks able to perform fault injections that result in a modification of the executed code [6], [7]. Since the discovery of the *RowHammer* attack [8], it is known that changes in read-only memory can even be induced by software. Hence, *code integrity* (CI) is also to be targeted in order to protect systems against a large body of attacks that disrupt the execution.

In this paper, we present a hardware-based solution that combines CFI with CI, while being non-intrusive. The *code and control-flow integrity* (CCFI) checks are performed at runtime by a dedicated hardware module outside the processor core. The control flow information – referred to as *metadata* – is stored in a dedicated section in memory and is aligned with the instructions. These metadata are fetched by a cache named CCFI-cache. Whenever a new instruction is requested by the processor, the corresponding metadata is fetched transparently and in parallel, so as not to disrupt or slow down the execution flow. The CCFI-checker verifies the integrity of execution flow changes by checking the effective target addresses. Function calls and returns are protected by an integrated shadow stack. Additionally, we ensure code and metadata integrity by computing a signature based on the executed instructions and metadata fetched by validating it against a precomputed signature contained in the metadata.

The proposed CCFI-cache architecture has been implemented on a RISC-V [9] platform, without modifying the processor core. Our experiments show that the run-time overhead is acceptable for different benchmarks. The price to pay for this very flexible solution is a two-fold increase in instruction memory.

In summary, the contribution of this work is a novel hardware-based CFI scheme that

- is non-intrusive, since the CPU core remains untouched,
- combines intra-procedural and inter-procedural CFI with code integrity,
- has low run-time overhead, and
- only requires very minor code modifications of the application code.

The rest of the paper is organized as follows: In Section II, we give some background on CFI and attacks. Related work and existing CFI solutions are discussed in Section III. In Sections IV and V, we present our solution and its implementation on a RISC-V platform, respectively. We give experimental results in Section VI, before concluding the paper with some remarks on future work in Section VII.

II. Background

In this section, we introduce the basic notions and security guarantees in the context of control-flow integrity.

A. Control Flow Graph

At the level of the machine code, a program is composed of multiple functions which in turn can be decomposed into basic blocks. A *basic block* (BB) is a straight-line sequence of instructions with a unique entry point and a unique exit point, i.e. if the control flow enters a BB, it will execute all of its instructions in sequence until leaving the BB at the exit point. A control flow transfer can only take place at the last instruction. Each function can be represented as a *control-flow graph* (CFG), where each node corresponds to a BB, and edges represent the control transfers between the BBs. A whole program is composed of the CFG of each function linked by edges representing function calls and returns.

B. Control Flow Hijacking

There are multiple ways in which an attacker can take over the control of a machine. In many cases, buffer overflows – due to bad programming – offer an entry point for an attacker. They can be exploited to inject code and/or to compromise return addresses stored on the stack to divert the execution flow.

Executing injected code can be mitigated by DEP which prevents written data to be executed. This protection can be circumvented by *code reuse attacks* that rely on (stubs of) existing functions in libraries, so-called *gadgets*. Known variants of this type of attacks are *return-oriented programming* (ROP), *jump-oriented programming* (JOP) or *call-oriented programming* (COP) [1]. As shown in [10], all such attacks rely on code pointer corruption. In this way, only legitimate code of the application is executed, but the CFG of the program is not respected anymore.

Another threat – invalidating DEP protections – are fault attacks, where memory contents are altered by physical means [6], [7]. Using the *RowHammer* attack [8], a dynamic RAM cell can be changed by rapidly reading neighboring cells before a refresh. Its stealthiness makes this threat extremely dangerous: Even trusted firmware code with a digital signature can be corrupted when residing in RAM. Some examples of attacks enabled by such modifications are Shamir's bug attack [11] (e.g., on RSA) or Sbox tampering attacks [12] (e.g., on AES). Fault attacks are difficult to master, but can be used to change instructions, to manipulate access rights, to skip an instruction, or to directly change the current program counter, in some cases without violating the CFG.

C. Control Flow Integrity

To prevent control flow hijacking and fault attacks, it is necessary to ensure that control transfer instructions execute as expected, i.e. any control transfer originates from an address that corresponds to a control transfer instruction and targets a valid destination address for this specific instruction.

For direct jumps and conditional branches, the valid destinations can be determined at compile-time. Verifying the integrity of these control transfers boils down to checking that for each executed jump or branch, there is a corresponding edge in the function's CFG. We refer to this check as *intra-procedural CFI.*

A different treatment is needed for function calls and returns. Since common functions – such as `printf` – are called from many sites, just checking that the function returns to one of these call sites does not provide a reasonable protection against ROP attacks. Instead, the correct pairing of call and return addresses needs to be ensured. We refer to this as *inter-procedural CFI.*

It should be noted that indirect jumps and calls pose a special problem for CFI as the set of destination addresses can be significant. However, in many cases – such as a `switch` statement which has been compiled to an indirect jump – the set of target addresses is usually small and can often be determined at compile-time. Otherwise, either manual code changes are necessary or these specific instructions must remain unprotected.

While these checks only consider control transfer instructions, it is necessary to ensure that *inside a BB*, all instructions are executed in-order, thereby preventing instruction skips. This verification, which is hard to implement in software, is called *intra-BB CFI*. Finally, *Code integrity* (CI) refers to verifying that all instructions have been executed *unaltered*.

In summary, a combined CFI and CI protection must ensure basic block integrity and verify both intra-procedural and inter-procedural control transfers.

III. Related Work

There exists a large body of research on protections against hardware and software attacks. Due to page limitation, we only present the most closely related work. For an overview on existing techniques, we refer the reader to [10]. Table I summarizes the protection levels of related techniques, which will be briefly discussed in the following.

A simple and effective protection against code injection is DEP, which is implemented in all modern general purpose CPU architectures. It allows to prevent the execution of memory segments that contain only data (such as the stack), making code injection difficult. It does however not protect against ROP and related attacks nor against hardware attacks.

In [5], de Clercq et al. present SOFIA, an architecture supporting software and control-flow integrity. The architecture has a two stage protection: Firstly, instructions are encrypted with a block cipher in a way that depends on the correct control flow, such that deviating from the CFG results in wrongly decrypted instructions. Secondly, groups of instructions are protected with a *Message Authentication Code* (MAC) to ensure code integrity and confidentiality. The proposed architecture achieves a protection level similar to our technique, while changes in the internal pipeline and the encryption and MAC computation make it both more intrusive and costly.

Intel's *Control-flow Enforcement Technology* (CET) [13] introduces a shadow stack, which stores return-addresses in addition to the normal stack. When a return instruction is encountered, the two addresses are compared and a security exception is raised in case of a mismatch. While the shadow stack is a powerful solution for inter-procedural CFI, it does not provide any other guarantees. The second feature of CET, *Indirect Branch Tracking* provides new instructions to mark valid branch targets, which provides a rudimentary protection against ROP-style attacks. In [15], the authors present HCFI (*Hardware-enforced CFI*), which is a modified SPARC architecture. It combines a shadow stack with a CFI-dedicated extension of the SPARC instruction set. While the solution concentrates only on call/return instructions, it achieves an impressively low run-time overhead of only 1%.

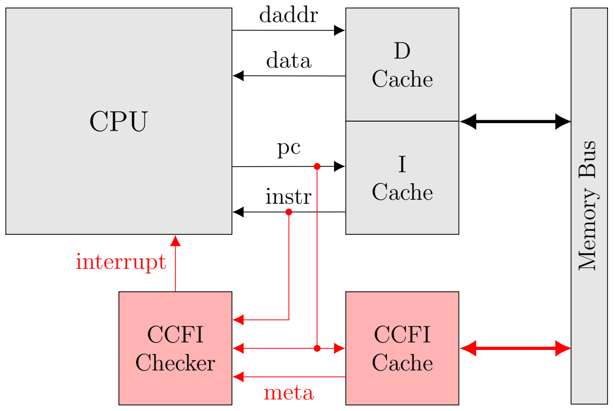


Fig. 1. Overview of the proposed architecture

PICON [3] is a purely software based solution, which is integrated into the LLVM compiler framework. The control flow policy is represented by a push-down automaton, which is then used at runtime by a monitoring process to match the actual execution. PICON provides a robust and portable protection against ROP-style attacks, while it does not protect against hardware attacks and compromised binaries. A similar approach has been presented in [14]. Their solution – called PathArmor – consists in a kernel module that monitors the execution paths of user processes. Its goal is a strong but practical protection of inter-procedural control transfers. By analyzing paths (rather than just single edges) in the CFG, they achieve a context-sensitive CFI without resorting to a shadow stack. These purely software-based solutions can be considered complementary to our approach.

Overall, the originality of our approach is a combined protection against cyber and hardware attacks, while being non-intrusive in contrast to other hardware-based solutions.

IV. Solution

In this section, we present the principles of our proposed solution, before discussing implementation issues in Section V.

A. Architecture Overview

The basic architecture is shown in Figure 1. We consider a simple platform based on a CPU core with separate instruction and data cache, which connect to the memory bus. CFI is ensured by two added hardware modules (shown in red): The *CCFI-cache* fetches the metadata which has been computed at compile-time, containing all control-flow related information. This information is used at runtime by the second module, the *CCFI-checker*. In order to follow and monitor the execution of the CPU, the CCFI-checker is hooked up to the interface signals between the CPU and the instruction cache.

The CCFI-cache has the same characteristics (bit width, size, associativity, replacement policy, . . .) as the instruction cache. For each basic block in the executed program, there is a corresponding block of metadata. Each block of metadata is perfectly aligned in memory to its corresponding BB, with a constant offset. For each access to the instruction cache,

TABLE I
COMPARISON OF SOME OF THE MOST PROMINENT PROTECTIONS

Protection	W⊕X	SOFIA [5]	Intel CET [13]	PICON [3]	HCODE [4]	PathArmor [14]	HCFI [15]	Our solution
a) Inter Procedural	X	✓	✓	✓	X	✓	✓	✓
b) Intra Procedural	X	✓	(✓)	✓	✓	X	X	✓
c) Intra BB	X	✓	X	X	✓	X	X	✓
d) Code Integrity	✓	✓	X	X	✓	X	X	✓
e) Non-intrusive	X	X	X	✓	✓	✓	X	✓

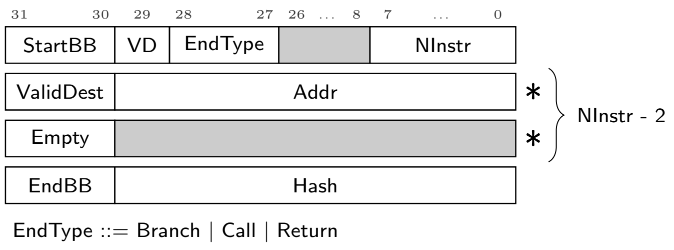


Fig. 2. Metadata format description

a parallel access to the CCFI-cache will be issued. In this way, complex address calculations are avoided. Furthermore, the instruction cache and the CCFI-cache will always be consistent, i.e. either both a BB and its metadata are cached or none of them.

The metadata for each BB contain three crucial elements that serve for the CFI verification:

1) The number of instructions in the current BB
2) The valid destination addresses of the succeeding BBs
3) A hash value of the instructions in the BB and its corresponding metadata

Section IV-B discusses in more detail the format of the metadata.

The actual verification is realized by the CCFI-checker. At the end of each BB, it checks the validity of the target address by comparing it with the precomputed valid addresses contained in the metadata, thereby ensuring intra-procedural CFI. In case of a function call or return, an integrated shadow stack is used to verify inter-procedural CFI. This shadow stack in embedded inside de CCFI-Checker and is not accessible from the main processor. Intra-BB consistency is ensured by a watch-dog counter that controls the number of executed instructions before a control transfer. Finally, code and metadata integrity is ensured by a precomputed signature that is compared to a hash value over the executed instructions computed at run-time. In case of any violation, an interrupt is raised. The details of the CCFI-checker are presented in Section IV-C.

B. Metadata

The format of the metadata is shown in Figure 2. There are four different entry types, which are distinguished by a label contained in the two most significant bits. For each BB, the metadata record has the same format: A StartBB entry, followed by zero or more ValidDest and Empty entries, and ending with a EndBB entry.

The entry StartBB marks the beginning of a BB. The bit VD indicates the presence of one or more valid destination addresses in the record. Note that in some cases destination of indirect branch or jump cannot be computed statically, in this case the VD bit is unset and there will be no verification of the destination address at the end of the BB. The field EndType defines the type of control transfer at the end of the BB: Call and Return indicate a function call and return, respectively. For any other type of control transfer, Branch marks either a BB that will always be succeeded by the next consecutive BB – i.e. there is no branch or jump at the end – or one ending with a direct or indirect jump or branch instruction. Typically, blocks ending with a direct jump or call will have one valid destination and conditional branches two, while indirect control transfers can have an arbitrary number of valid destinations. Finally, there is an 8-bit field NInstr, which gives the total number of instructions in the BB.

The ValidDest entry contains one valid destination address, corresponding to an allowed edge in the CFG. Finally, the end of the BB is marked by an EndBB entry, which additionally contains a hash signature computed over all the instructions of the BB.

All metadata are computed offline at the end of the compilation, after code optimization. For each BB a metadata record of the same size is allocated. Metadata are stored in a custom section of the program file, which has the same size as the `.text` section.

If the number of entries needed for the CFI information is smaller than the number of instructions in the corresponding BB, then the metadata is simply padded with Empty entries before the EndBB entry. The opposite case can occur as well, if the BB is very short or if there are multiple valid address entries. In order to match the BB size with the metadata record, the compiler inserts `nop` (no operation) instructions just before the last instruction of the BB.

Depending on the specific implementation of the CPU, there can be other situations that require adjustment of the binary code. One such case is branch prediction, which potentially leads to a mismatch between the *fetched* instructions and those that are effectively *executed*. Since the CFI-cache only monitors the interface signals of the CPU, it needs to be aware of such features in order to correctly detect the destination of branches. Section V explains how we have resolved this issue for the used RISC-V implementation and gives an example for the correspondence between binary code and metadata. We evaluate the performance penalty of these code adjustments in Section VI.

C. CFI Verification

At loading time, the `.metadata` section is loaded into a reserved memory region, at a constant offset from the `.text` section. This offset allows calculating the metadata address on-the-fly based on the current instruction address.

During the execution of the program, the instruction cache and the CCFI-cache are always kept in a consistent state, which allows the CCFI-checker to follow the execution and verify the control flow on-the-fly using the metadata. The checker processes the metadata in parallel to the execution. For this purpose, the CCFI-checker is composed of the following principal components:

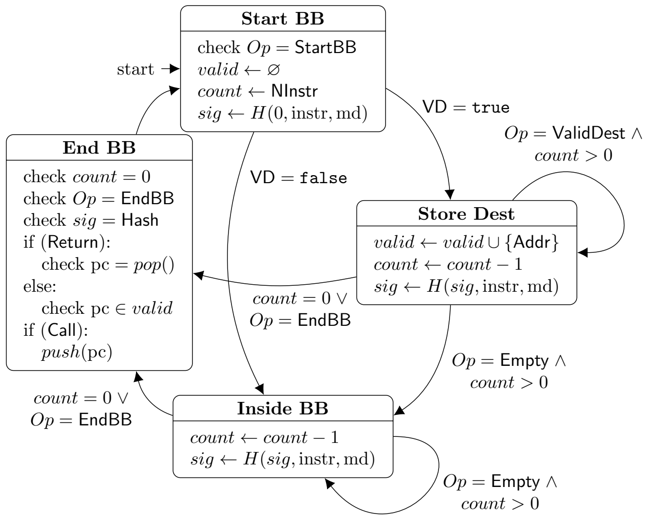
- A set of registers to store the valid destination addresses,
- A shadow stack to store function return addresses,
- An instruction counter, and
- A signature register to compute a hash value of the executed instructions.

A simplified view of the control state machine of the checker is shown in Figure 3. The state machine basically follows the structure of the metadata record (cf Figure 2). At the beginning of a BB (state **Start BB** in Figure 3), it sets the instruction counter to the number of instructions in the BB and initializes the signature register. It also checks that the beginning of the BB is correctly labeled with StartBB. The valid destination addresses are collected while traversing the BB (state **Store Dest**) and stored in the internal register bank[1]. If there are Empty entries in the metadata record, the state machine loops in the **Inside BB** state until the end of the BB. During the traversal, the signature register is updated after each instruction. For this purpose, a suitable hash digest function H needs to be chosen.

The actual verification takes place in the **End BB** state. There are two conditions that each triggers the transition into this state: Either an EndBB label is found or the instruction counter reaches zero. This ensures that both too long and too short BBs will be detected immediately. Normally, the end of the BB should coincide with the instruction counter reaching zero, which is verified in the **End BB** state. It is also checked if the hash value extracted from the EndBB entry equals the signature register. Call and Return are verified using the shadow stack. If there have been any valid address entries in the metadata, these are used to verify the effective target address. Note that this applies either for calls or local branches and jumps. The implementation of the internal register bank must ensure that the address comparison can be performed in parallel for all valid entries in one clock cycle, before the state machine continues to process the next BB.

In case any of the checks fails, an interrupt will be triggered, allowing the CPU to react to the attack immediately. Note

[1]Note that the size $k \geq 2$ of this register bank is an implementation parameter that can be chosen freely. Any BB with more than k valid targets can be split recursively until each BB has at most k valid successors.



Start BB
check $Op = \mathsf{StartBB}$
$valid \leftarrow \varnothing$
$count \leftarrow \mathsf{NInstr}$
$sig \leftarrow H(0, \mathrm{instr}, \mathrm{md})$
start
VD = true
$Op = \mathsf{ValidDest} \wedge count > 0$
VD = false
End BB
check $count = 0$
check $Op = \mathsf{EndBB}$
check $sig = \mathsf{Hash}$
if (Return):
check pc $= pop()$
else:
check pc $\in valid$
if (Call):
$push$(pc)
Store Dest
$valid \leftarrow valid \cup \{\mathsf{Addr}\}$
$count \leftarrow count - 1$
$sig \leftarrow H(sig, \mathrm{instr}, \mathrm{md})$
$count = 0 \vee Op = \mathsf{EndBB}$
$Op = \mathsf{Empty} \wedge count > 0$
$count = 0 \vee Op = \mathsf{EndBB}$
Inside BB
$count \leftarrow count - 1$
$sig \leftarrow H(sig, \mathrm{instr}, \mathrm{md})$
$Op = \mathsf{Empty} \wedge count > 0$

Fig. 3. CCFI-checker state machine

that for simplicity reasons, Figure 3 does not show the state transitions in the case of a security violation.

D. Attack Model and Security Guaranties

In this work, we address the protection of embedded platforms without DEP. We consider that the attacker is able to exploit programming bugs which allow buffer overflows. Such attacks can either modify the return address on the stack and/or inject malicious code. Note that this attack model is quite permissive in contrast to the classical CFI setting, which usually considers that code memory is immutable [2].

Additionally, we consider non-destructive physical attacks. This includes random changes in memory by either software-driven attacks (row-hammer) or hardware attacks (such as electromagnetic injection or glitches) leading to instruction skips. Since the successful demonstration of practical attacks such as row-hammer, physical attacks must be considered a realistic scenario, especially in the context of embedded and mobile devices.

Assuming that the main memory contains the code alongside with its correctly generated metadata, the CCFI-checker enables detection of the following attacks:

- Changing a return address on the stack (detected by shadow stack)
- Changing the target of a call, branch or jump outside of the static CFG (detected by destination address verification)
- Returning or jumping into the middle of a BB (detected by StartBB label check)
- Adding instructions at the end of a BB (detected by EndBB label check and instruction counter)
- Turning a branch into a `nop` (detected by signature check)
- Changing the pc to skip an instruction (detected by signature and instruction counter)

TABLE II
CODE AND METADATA CORRESPONDENCE

Instruction address	Instruction	Metadata address	Metadata	Metadata description
0x00000A88	lbu a5,0(s1)	0x40000A88	0xA0000004	StartBB | VD=1 | EndType=Branch | NInstr=4
0x00000A8C	nop	0x40000A8C	0x4000029A	ValidDest | Addr=0xA68
0x00000A90	bnez a5,0xA68	0x40000A90	0x400002A6	ValidDest | Addr=0xA98
0x00000A94	nop	0x40000A94	0xFC035B60	EndBB | Hash=0xFC035B60
0x00000A98	lw a5,-68(s0)	0x40000A98	0xA0000004	StartBB | VD=1 | EndType=Branch | NInstr=4
0x00000A9C	addi a5,a5,1	0x40000A9C	0x40000118	ValidDest | Addr=0x460
0x00000AA0	sw a5,-68(s0)	0x40000AA0	0x0	Empty
0x00000AA4	j 0x460	0x40000AA4	0xDAF87E5C	EndBB | Hash=0xDAF87E5C

- Changing any instruction word in memory or up to the CPU interface (detected by signature)
- Deleting or manipulation metadata in any way inconsistent with the code (detected by signature)

One obvious limitation are physical attacks that directly affect the internal state of the CPU, such as the register state or skipping computations within the pipeline. Note that however such attacks will be caught if they directly or indirectly change the instruction address on the cache interface. We also do not consider advanced destructive attacks (such as focused ion beams) that could e.g. cut the interrupt line on the circuit die and thereby practically disable the CCFI-checker.

Furthermore, data only attacks that do not change the static control flow are not detected. An attacker that has full control over the memory could also forge metadata. A typical solution for this problem is to assume that the metadata (or the metadata and the code) reside in a protected read-only memory. Considering our proposed architecture in Figure 1, such a solution can easily be implemented by having a completely separated memory bus for the CCFI-cache, thereby preventing any access to the metadata originating from the CPU.

Finally, special care needs to be taken for the treatment of the interrupt triggered by the CCFI-checker. Since interrupt mechanisms vary greatly on different target platforms, there is no system-independent solution. For instance, if interrupt target vectors are writable from user code, the interrupt service routine (ISR) itself needs to be protected. Any tampering with the ISR would then lead to a re-occurring violation, blocking the system in an infinite loop. In embedded platforms, watchdog counters are typically used to reset the system when it gets caught in a deadlock. Depending on the application, if recovery is considered less important, the interrupt line of the CCFI-checker can also be routed directly to the reset line, preventing any attack path via the ISR.

V. IMPLEMENTATION

To validate the CCFI-cache architecture, we have implemented it on a microcontroller platform based on the PicoRV32 [16], a free implementation of the RISC-V ISA [9]. The CPU core is a 3-stage pipeline processor with a single memory interface for accessing instructions and data. In our platform, the separate instruction and data caches are accessed via an address decoder. The platform uses a crossbar for memory access. To highlight memory contention between instruction cache and CCFI-cache, we have implemented two different memory layouts: 1) Two separate memories for application code (`.text` and `.rodata`) and metadata, 2) a single memory to store both. In all cases, our platform uses one big RAM as runtime memory.

As mentioned in Section IV-B, branch prediction can potentially pose a problem for the CCFI-checker, since the address seen on the memory interface (i.e. the program counter) may not coincide with the effective branch target address. The PicoRV32 does not feature branch prediction, but branches are decided late in the pipeline, such that the instruction following a conditional branch is always fetched. We resolve this specific case during the compilation phase by always inserting a `nop` instruction after a conditional branch, thereby deferring the actual control transfer by one instruction. We claim that the proposed architecture is suitable for more complex prediction schemes, for instance using a checker that mimics the prediction logic. A detailed discussion of the required modifications is beyond the scope of this paper.

Table II shows an example with two BBs and the corresponding metadata. The upper BB ends with a conditional branch and there are two valid destinations stored in the metadata record. In the code, there are two additional `nop`s, which are needed to match the size of the metadata, and – for the second one – to resolve the prefetch of the conditional branch.

VI. PERFORMANCE

We have tested the solution on a Digilent Nexys4 DDR board with an Artix 7 FPGA [17]. The resource usage is summarized in Table III. As can be seen, the hardware overhead is small, it is in the order of 10% in terms of LUTs and FFs [2].

Figure 4 left shows the impact of the inserted `nop` instructions on the code size. Over the different benchmarks (bubble sort, Dhrystone, and an AES encryption) and optimization levels (`-O1`, `-O2`, `-O3`, and `-Os`), the overhead ranges from 9% to 30%.

Figure 4 right shows the impact of our solution on runtime performance. We compare 1) the original program with 2) the modified program, but without protection, 3) CCFI enabled

[2]The difference between total CCFI and the sum of CCFI-cache and checker is due to glue logic and the more complex memory and bus infrastructure.

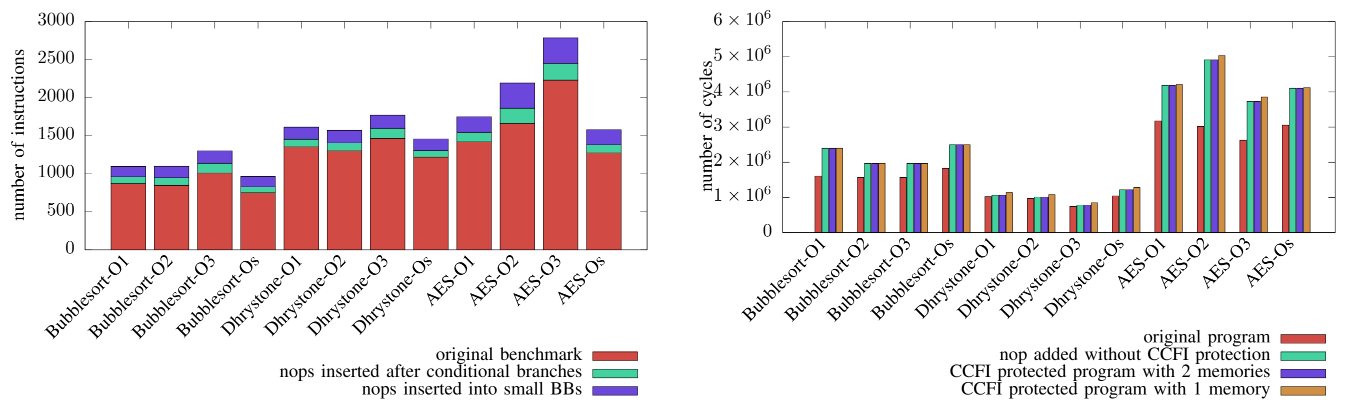


Fig. 4. Overhead on code size and execution time

TABLE III
HARDWARE COST

Component	LUTs	FFs	RAMB18
Original	11094	8939	8
CCFI-cache	531 (+4.8%)	139 (+1.6%)	8 (+100.0%)
CCFI-checker	294 (+2.6%)	443 (+4.9%)	0 (+0%)
Total CCFI[2]	1250 (+11.3%)	777 (+8.7%)	8 (+100.0%)

with parallel memory access, and 4) CCFI enabled with sequential memory access. We can see that most of the performance penalty is due to the inserted `nop` instructions leading to a runtime overhead between 2% and 63%. Beyond this overhead, the CCFI protection using parallel memory access does not further impact the performance. The implementation using sequential memory access incurs a small additional cost in the order of 1% (up to 8% in the worst case), which is directly related to the instruction miss rate of the benchmark.

a) Fault attacks.: We have simulated fault attacks by modifying an instruction code in the main memory and in the cache. All performed attacks have been detected by the CCFI-checker, through the signature check. Any fault injection inside the processor which directly manipulates the program counter is detected as well.

b) Software attacks.: We have also tried several software attacks on the protected platform. Any buffer overflow manipulating the stack is detected thanks to the shadow stack. Even changing *unprotected* indirect jumps is detected if the destination address is not the beginning of a BB (labeled by a StartBB entry). Thus, ROP or JOP attacks are made very difficult, since the number of useful gadgets is significantly reduced.

VII. Conclusion

We have presented a non-intrusive hardware-based protection able to effectively mitigate cyber- and physical attacks. Our solution uses precomputed control flow information which are verified at runtime. Only requiring a double code memory size, our solution is very competitive regarding the hardware overhead and the performance penalty, which are minimal and affordable in most cases which make our technology practical and deployable.

Up to now, the metadata are computed as a post-processing step using the binary code. The integration of this step into the compiler flow is left to future work, to have better control over well sized BBs and the resolution of indirect jump destinations.

References

[1] N. Carlini and D. A. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, 2014.*, 2014, pp. 385–399.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, 2009.

[3] T. Coudray, A. Fontaine, and P. Chifflier, "PICON: control flow integrity on LLVM IR," in *Symposium sur la sécurité des technologies de l'information et des communications, Rennes, France, June 3-5, 2015*, 2015.

[4] "hidden for blind review."

[5] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," in *Design, Automation & Test in Europe (DATE), Dresden*, 2016, pp. 1172–1177.

[6] D. Karaklajić, J. M. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2295–2306, Dec. 2013.

[7] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *Smart Card Research and Advanced Applications (CARDIS), Bochum. Revised Selected Papers*, 2015, pp. 161–176.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, 2014.

[9] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[10] L. Szekeres, M. Payer, T. Wei, and R. Sekar, "Eternal war in memory," *IEEE Security & Privacy*, vol. 12, no. 3, pp. 45–53, 2014.

[11] E. Biham, Y. Carmeli, and A. Shamir, "Bug attacks," in *CRYPTO*, ser. LNCS, vol. 5157. Springer, 2008, pp. 221–240, Santa Barbara, CA, USA.

[12] A. C. Aldaya, A. C. Sarmiento, and S. Sánchez-Solano, "AES t-box tampering attack," *J. Cryptographic Engineering*, vol. 6, no. 1, pp. 31–48, 2016.

[13] Intel, "Control-flow enforcement technology preview, revision 2.0," June 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf