POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Control-Flow Integrity for Embedded Systems: Study Case of an FPGA-Based Solution

Author: Antonio Ettore Epifani

Supervisor: Paolo Ernesto Prinetto

April, 2021

Abstract

Nowadays, embedded devices are taking on a significant degree of pervasiveness in many sectors of our daily life, from industry to home automation, from healtcare to the urban dimension. These devices are responsible for creating and managing an extraordinary amount of data concerning our lives, which poses equally extraordinary challenges: on the one hand, it becomes essential to secure the entire data traffic, and on the other, equally important, these systems need to be reliable and not easily manipulated, according to the *resilience-by-design* paradigm.

Physical security is fundamental but insufficient, since many breaches can be opened by the code that such systems run. A relevant amount of weaknesses comes from the diffused employ of typical embedded system languages, such as C and C++. These languages offer a high level of hardware control and optimization, but at the same time, they are *memory-unsafe*, i.e., the protection of the memory from possible corruptions is completely left to the programmer.

Memory vulnerabilities open the door to many typologies of attacks, massively reported and studied by security researchers, such as *Code-Reuse Attacks (CRA)*, in which the flow of the program is hijacked to sections of code already present in memory but not intended to be executed in that order. One of the most famous offensive techniques belonging to this category is called *Return-Oriented Programming (ROP)*, along with some of its variants, *Jump-Oriented Programming (JOP)* and *Call-Oriented Programming (COP)*. This technique exploits a memory vulnerability inside a program in order to collect a malicious sequence of bytes, said *gadgets*. These gadgets are used as a chain of little actions to form a greater malware, able to execute arbitrary code in the context of the program.

The purpose of this thesis work is to present a practical use case of a Control Flow Integrity solution based on a Control Flow Monitor synthesized inside a reconfigurable hardware module. In this thesis, an ad-hoc firmware is uploaded onto a development board, the SE*cube*™, which integrates a microcontroller, an FPGA and a Smart Card Reader. In this scenario, the device communicates with a smartphone application via a BLE peripheral in order to perform its intended tasks. The application, though, can deliver a malicious payload which will hijack the program counter of the firmware. The final aim of this work is to show the different behaviour of the embedded system in presence or not of the Control Flow Integrity solution previously desribed.

Contents

Chapter 1

Introduction

1.1 The Importance of Embedded Systems Security

In the last decade, the Digital Revolution has brought extraordinary changes in the world. We are witnessing a massive increase in the amount of data exchanged over the Internet and there is more and more people connected to the Internet. According to [22], in 2017 the estimated number of connected IoT devices was around *27 billions*, and this number is forseen to increase by a 12% annualy, *reaching 125 billions in 2030*. IoT devices are getting more and more diffused in every field, from business to private life. Not only they produce and exchange massive amounts of data, but with digital progress, they are in charge of managing many procedures and performing tasks that were normally conducted by humans. We may find eloquent examples in domotics, modern medicine and biotechnology, autonomous driving and smart production systems, just to cite a few. The motivation of this transition is straight forward: moving to digitalized and connected environment improves and efficiency and performance dramatically.

Being the benefits so obvious, we cannot ignore the risks. *The Malfunctioning of these systems would lead to potentially enormous losses and catastrophic safety or privacy related issues.* Just to mention the impact of a security hole in one of such systems, in [34], the author describes serious flaw in the information exchanging protocol of a commercial Smart Lock device. According to its write-up, in his research, the author found out that the bug would allow major sensitive data disclosure about the owner of the device and unauthenticated access to any malicious intruder who is capable of accessing the server to which the device connects.

Another interesting example of security issue related to the IoT world is the *Mirai Botnet*, which was discovered in 2016. The malware, developed by Mirai, had first to infect a vulnerable device and would reproduce from that device by looking for other vulnerable devices, trying to login inside those devices via telnet using some common credentials. The Mirai Botnet was responsible for many Distributed Denial of Service attacks. Some of them reached bandwidths like 1 Tbit/s.

In the past, common vulnerabilities in Operating Systems running on consumer-tier laptops and PCs, would have rarely caused more than some data or financial losses. In the era of Internet of Things, being all these systems around us, and being them capable of transmitting and receiving data over the air poses two major problems, as evinced in the

example above:

1. The impact of a vulnerability inside a device can now be worringly high
2. The attack surface available to a malicious attacker is drastically expanded, to the level that it is possible to compromise one of these devices from another part of the globe.

The root of these two problems can be spotted in the fact that IoT devices are constantly connected to the Web, since they have to communicate and monitor remotely different parameters and collect important amounts of data. IoT devices belong to the broadest category of *Embedded Devices*. It is possible to insert in this category all those devices which are built for special purpose tasks and typically are composed of a system of small dimensions. These two features come from the strict requirements of the contexts in which these products are adopted. They need to

- be power-efficient
- fit in small slots/places
- perform (well) a single task
- be subject to stringent real-time constraints

Due to these necesseties, embedded systems often run small stack of software on their hardware; few are the cases in which they are provided with a full featured Operating System. Modern development stacks instead, nowadays, engulf different libraries and middlewares, which are responsible of guaranteeing the most common security features, such as sandboxing and memory protection. Embedded Devices, on the other hand, run so called *bare-metal* applications, which usually have small abstraction from the hardware, in order to be fast and small. For this reason, they are usually developed in low abstraction languages, such as C and C++. The adoption of these programming languages enables the developer to have fine control and total responsibility over memory management. Without a Security-by-Design approach, the application will likely have different vulnerabilities which can be easily exploited.

1.2 Memory Corruption and Buffer Overflow

Memory vulnerabilities are the main means of exploitation in embedded applications cyber attacks. The principle behind them is that a malicious interactor with the application is capable of *writing into a memory location where she is not supposed to have access to*. Being able to write inside a program memory can have several consequences, but the most of the time it results in a complete takeover, on behalf of the attacker, over the program execution. Typically, an attacker aims at exploiting a memory corruption vulnerability in order to arbitrarily write or read memory portions of the process/program.

One of the main methods of corrupting a program's memory is known as *Buffer Overflow*. Buffers are regions of memory which are accessed in order to read or store data inside. One of the most crucial challenges a programmer has to face when developing her code is

to well design these buffers and allocate for them the correct amount of space. This is not trivial, as the programmer has to either make assumptions about the length of the data that the buffer has to store or check at run-time the length of the data before passing it inside the buffer, which can result inefficient.

If these checks can be somehow deceived or the assumptions can be broken, a buffer overflow vulnerability is present. *The attacker writes more data into the buffer than what the buffer can hold and the excess data will overwrite a memory region which does not belong to the buffer.* This is also possible and easy for an intruder because of the simpliity of C in terms of memory management. Being a relatively low-level language, C does not provide by default any checks on the boundaries of arrays and buffers.

Let us consider the following snippet of C code:

```
int main(){
        char buffer[32];
        scanf("%s", buffer);
        ...
        return 0;
}
```

is vulnerable to memory corruption by buffer overflow because of the function `scanf`, which allows input incoming from the user without any constraint on the length. Being the buffer allocated for 32 bytes of spaces, if the user inputs a string longer than 32 characters, it will overflow the buffer and corrupt some other memory regions. Typically this, would result in a segmentation fault which would make the program crash. If the excess bytes, though, are suitably crafted, the user could overwrite the return address of the function on the stack and redirect the program counter to a location of her choice.

The one listed above is the most simple form of buffer overflow which is also called *Stack Overflow*, because it overflows a buffer present in the stack, possibly overwriting past the current frame.

A buffer overflow, though, may be sufficient on its own to cause a denial of service in the program, but not to escalate to any profitable state for the attacker, like *Information Leakage* or *Arbitrary Code Execution*. What is usually done by the attacker is to inject in memory a *Shellcode*, which is a sequence of bytes which can be interpreted as machine instructions. If the attacker manages to inject this snippet and to redirect the execution of the program to that snippet, she can perform any operation written inside the Shellcode. The name "*shellcode*" derives, indeed, from the fact that this snippet is often used to spawn a command line shell in the context of the vulnerable program and gain flexible control over the machine.

It is necessary, therefore, to carefully analyze the program and craft a proper payload to reach any valuable state. Before concluding this introduction, it is important to remark that buffer overflows are just the first stage of a memory vulnerability exploitation.

What we will focus on in this work is the second stage, which consists in crafting a payload that manages to takeover the normal control-flow of the program. Thus, any mitigation to stack overflows and similar overflowing techniques, is not going to be discussed in the following pages. The objective of this work is to remark the importance of providing security solutions which respect the constrains of the embedded world. We do this by describing a solution developed and presented at our department at Politecnico di Torino.

We do also desire to show the benefits of such a solution by providing a real world example. We created an environment that reproduces an IoT Smart Access Monitoring system. We then attack the system by individuating a vulnerability in the firmware running on the microcontroller. Finally, we apply our security mechanism by showing how it is implemented and uploaded onto the board of choice.

The following of this thesis articulates on three chapters. In the next chapter we are going to describe the issue of Code Reuse Attacks, introducing Return-Oriented Programming and its variants. Next we list and describe some of the most popular solutions belonging to the category of Control-Flow Integrity. We make a difference between Software and Hardware Mitigations.

In the third chapter we describe the solution that has been presented at PoliTo, by listing its goals, and discussing about its architectures, referring to the main article on which this work is based **??**

The last chapter describes the procedure that has been executed in order to exploit a memory vulnerability on the board. We first introduce the SE*cube*™ device and motivate its choice. We then explain how we reached code-redirection with a ROP gadget chain and demonstrate the effects of this attack. Finally we describe how to synthesize and upload the design of the CFI monitor presented in **??**, and how the adoption of this solution of our design manages to block the attack we previously described.

Chapter 2

Background: Code-Reuse Attacks (CRA)

2.1 The principle

As already mentioned in 1, memory corruption is the first step of an adversary to hijack a program, and if the aim of the attacker is to redirect the control-flow of the program, she has two options of attacks:

- Code-Injection
- Code-Reuse attacks

The former is a simple strategy that exploits the memory vulnerability to inject a payload composed of a series of byte representing the machine instruction the attacker is willing to execute. Typically, in calling conventions, the return address is written onto the stack, and if the attacker manages to overwrite that address, she can potentially reach any points in the code.

In a common Stack Overflow with code-injection attack, the payload that the attacker would use to perform a simple code-injection resembles the following structure:

1. Optional padding to fill the buffer and smash the stack frame
2. Sequence of bytes which represented the opcode of the instructions that the attacker intends to execute
3. Address which would overwrite the return address of the vulnerable function and would point to the beginning of the injected opcodes.

Note that the padding at the beginning has a double use in this mechanism, since it not only allows to fill the buffer and align the instructions and the address properly inside memory, but it can serve as a *NOP sled*; the `NOP` instruction has the only purpose of increasing by one the program counter and doing nothing more. If more `NOP` instructions are put one after the other, their only effect is to reach as soon as possible the first instruction which is not a `NOP` instruction, acting as a sled.

After the payload has been sent to the victim, the aftermath on the stack memory will be the one represented in Fig. 2.1 at the right side

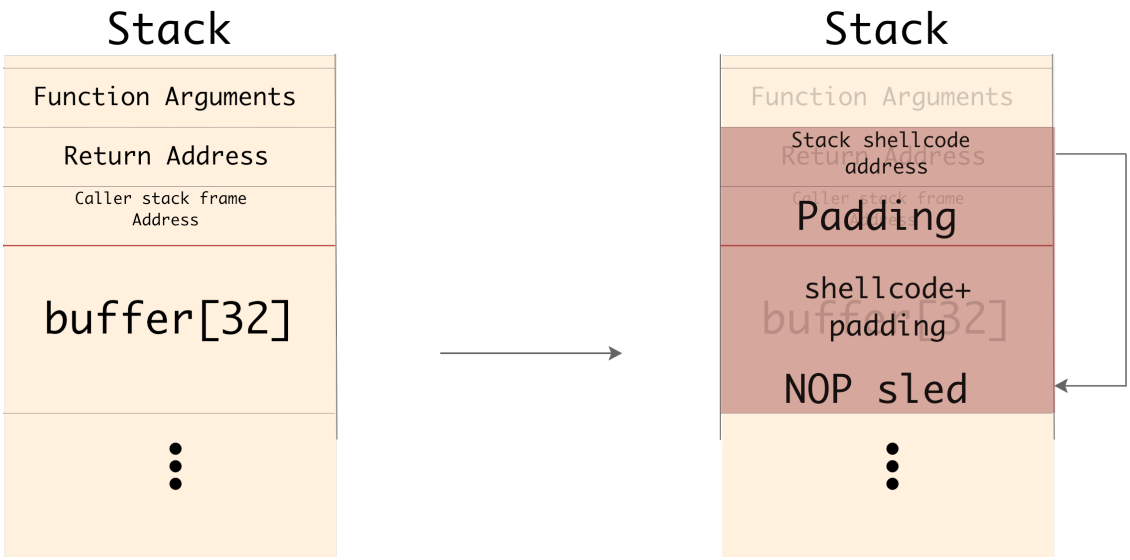

Figure 2.1. Stack before and after code injection. The return address is overwritten with the address of the injected code

Code-Injection is simple and allows the attacker to exploit the vulnerability without requiring any additional information about the program except for the stack addresses at which her payload will be loaded. Nevertheless, a simple countermeasure can be adopted to prevent an attacker from injecting malicious code onto the stack: *NXE - Non eXEcutable stack*. This mitigation simply makes stack non executable, thus forbidding any payload inserted in stack-dwelling buffers to hold shellcodes or other instructions opcodes. If the attacker manages to overwrite the return address of the unsafe function, the program will raise an exception and stop. A parallel approach is adopted for the heap section in the process.

These kinds of mitigation almost annihilate any effort to inject arbitrary instructions inside the program. This is the main drawback of code-injection, and the reason why nowadays, hackers prefer to adopt a different approach, when trying to redirect the program counter.

Code-Reuse Attacks supply to the lacks of code-injection. The principle is to reuse *already existing code inside the program*. An intruder would proceed as follows to devise such kind of attacks:

1. find interesting pieces of code inside the program that perform the wanted computations

2. retrieve the location of the code snippet and write it to an address which is going to be put into the `PC` register

The main challenge for an attacker is to find useful snippets of reusable code inside the program. A common practice for an attacker is to choose functions from the linked library `libc`, which is almost certainly present in any program that includes standard libraries. This technique is generally known as *return-to-libc*. A program abuser can reach maximum impact with this technique with an extremely concise payload, by returning to the *system()* function, for example. By calling *system()*, the attacker can execute any other program on the system, provided that she manages to pass the argument of the executable. A simple solution could be to pass to an input of the program the name of the desired executable as a string, or better, as an environment variable. `/bin/sh` is a good choice, being it short and providing full access to the system. A graphical example of the situation before and after exploitation for a return-to-libc attack is shown in Fig. 2.2

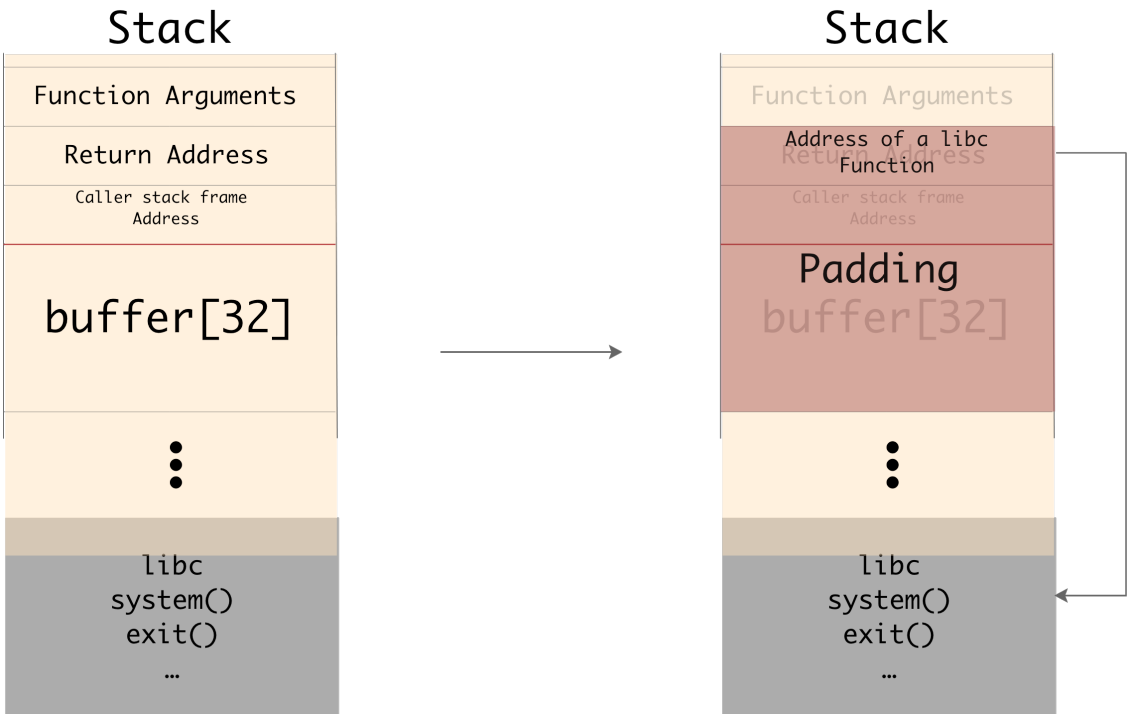

Figure 2.2. Exploitation of a buffer overflow for a return-to-libc attack

Defending against return-to-libc is not as trivial as simple code-injection. The attacker does not execute code on the stack, hence, the simple Data Execution Prevention mechanisms adopted to counteract normal code-injection will not work in this case. Furthermore, reusing code allows for more complexity, which is limited in normal code-injection by the fact that the desired instructions must be passed as opcodes.

2.2 Return-Oriented Programming (ROP)

One of main drawbacks of *return-to-libc* attacks is that the intruder has no chance of using more than one function per time, and the degree of control is limited to the control over a shared library function. Because of this, generally *return-to-libc* is not considered Turing Complete, except for some complex variants of the technique, as described in [32]. In order to mitigate this issue, in [28], Shacam explains a new code-reuse technique, which confers much more control to the attacker than normale return-to-libc. *ROP - Return-Oriented Programming* is a Turing-Complete methodology of exploiting preexistent code in program memory. Shacam conjectures that in a sufficiently large code-base, the number of available gadgets will be sufficient to build a chain for any computation the attacker may require.

In order to concatenate these gadgets and create the ROP chain, an attacker has to dispose the location of the beginning of each gadget one after the other, in order of execution. After the instructions composing the gadgets have been executed, the `ret` instruction at the end of each gadget will pop from the stack an address to be put in the Program Counter. In a normal function call, the `ret` instruction would have popped the return address of the function caller, put there by the `call` instruction. In the end, the ROP chain can be seen as a portion of code

Basically, the principle of ROP is to create a chain of small pieces of code called *gadgets*. These gadgets are typically ending lines of a basic block, which is a set of instructions comprehended between two branch instructions. It requires, then, for a gadget to be such, to end with a branch instruction, more specifically, `ret` instructions in x86 ISA. In contrast to return-to-libc, this allows to reduce the number of lines an attacker has to look for in order to perform a precise task, thus increasing the number of possible operations permitted. It is demonstrated that with a sufficiently large code-base, it is possible to perform any computation, and with different flavours.

Hereafter, we will consider an example of a Return-Oriented Programming payload and its effects on the program memory and control-flow. Let us first observe the same vulnerable piece of code:

```
void func(){
        char buffer[32];
        scanf("%s", buffer);
        ...
}
```

This snippet carelessly takes an input from the user and passes it to the buffer without any boundaries check. The stack layout of such code is shown in Fig. 2.3

The malicious user simply has to fill the first 32 bytes arbitrarily plus 4 additional bytes to overflow also the frame pointer. She then closes the payload with two addresses which will be the gadget addresses. Fig. 2.4 describes the structure of this simple payload

We assume that the array `buffer` is the only variable allocated in the current stack frame. After the payload has been moved inside the buffer by `scanf`, the first four bytes after the sequence of 32 'A's will overwrite the frame buffer (we are still considering an x86 architecture here, but this scenario is the same on other platforms too, like ARM). The other 8 bytes represent the address at which the gadgets are located. When the `ret` instruction will pop the first address, the processor will fetch the instruction at that address

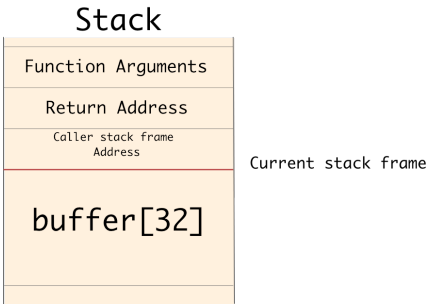

Figure 2.3. Stack before payload injection

Figure 2.4. Payload Example of a ROP attack

and then it will return as well, popping the next address. Fig. 2.5 clearly illustrates the order of execution of the program after exploitation.

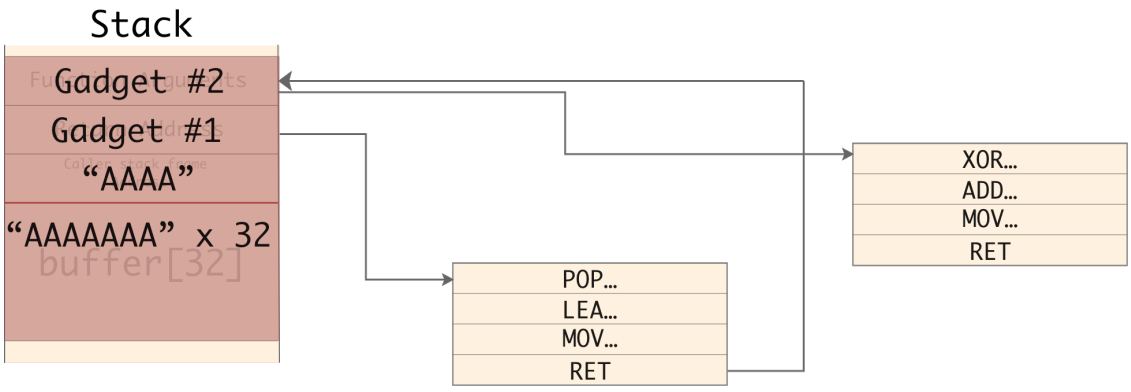

Figure 2.5. Stack's situation after ROP

We have discussed up to here Return-Oriented Programming in x86 machines, which provide the `ret` instruction to recover the return address from the stack and jump to the instruction after the procedure call. Not all architectures have an explicit return instruction. It is the case of ARM processors. ARM ISA does not feature any `ret` instruction. Instead, the calling convention for ARM demands that the return address is stored in the `ldr` register before branching to the procedure. The `ldr` register is then pushed onto the

stack along with other registers. Before returning, the corresponding `pop` will place the content of the `ldr` register inside `pc`, thus imitating the behaviour of a x86 `ret`.

```
        MOV ldr, [pc, 8]
        BX r0 #r0 contains procedure address
        ...
#       procedure preamble
        POP {r0-r4, r7, ldr} #save ldr
        ...
#       procedure ending
        PUSH {r0-r4, r7, pc} # take value previously in ldr into pc
```

The working principle of ROP is the same, the difference is that the algorithm of finding gadgets has to take in consideration different structures for returning instructions.

Finally, it is worth mentioning that there exist variants of Return-Oriented Programming, that are based on the same concept of gadgets, but they exploit simple branch instructions like `jmp`, `bx`, `call`, and others. Researchers have developed new techniques like *Jump-Oriented Programming - JOP* [4] and *Call-Oriented Programming - COP* [26]. The former uses `jmp` instructions and chains the gadgets with a *dispatcher gadget*, which is a point to which a jumps at the completion of its execution and from which the next gadget will be reached. The second uses gadgets that end with a `call` instruction. These techniques allow to overcome some defense mechanisms developed to detect ROP, like DROP [9] and DynIMA [15], which alert the system in case too many `ret` instructions are executed one after the other.

Beside specific mitigations for each variant, in literature there exist plenty of disserations and methodologies to implement *Control-Flow Integrity*. These techniques aim at guaranteeing protection against code-reuse and code-injection attack. They achieve this by building a model of the Control-Flow of the program and assuring that it is respected during execution. We shall look now at the two broad categories of these techniques: *Software-Implementation* and *Hardware-Implementation* of Control-Flow Integrity.

2.3 Code-Reuse Attacks Mitigations: CFI

We have analysed so far different possibilities of Code-Reuse attacks and some possible defenses. The truth is that no mitigation mentioned so far is capable to stop completely Code-Reuse Attacks. In the following lines we are going to analyze a vast category of CRA defense mechanisms: *Control-Flow Integrity - CFI*. CFI tackles the principle of code redirection by building a model of the Control Flow of the program: the *Control-Flow Graph - CFG*. Before we enter into the details of the concept of CFG, we shall better explain *basic-blocks*. Basic blocks are the snippets of instructions that are comprehended between two branch instructions. A CFG is *a directed graph of which nodes are basic blocks of the program and edges are the different branches operation*. Hence, an edge connects two basic blocks if there is a branch instruction from one to the other. Fig. 2.6 shows an example program and its corresponding CFG.

The objective of a Code-Reuse Attack is essentialy to violate this graph, by reaching nodes from others with paths that are not present in the original CFG. Stated this way, the

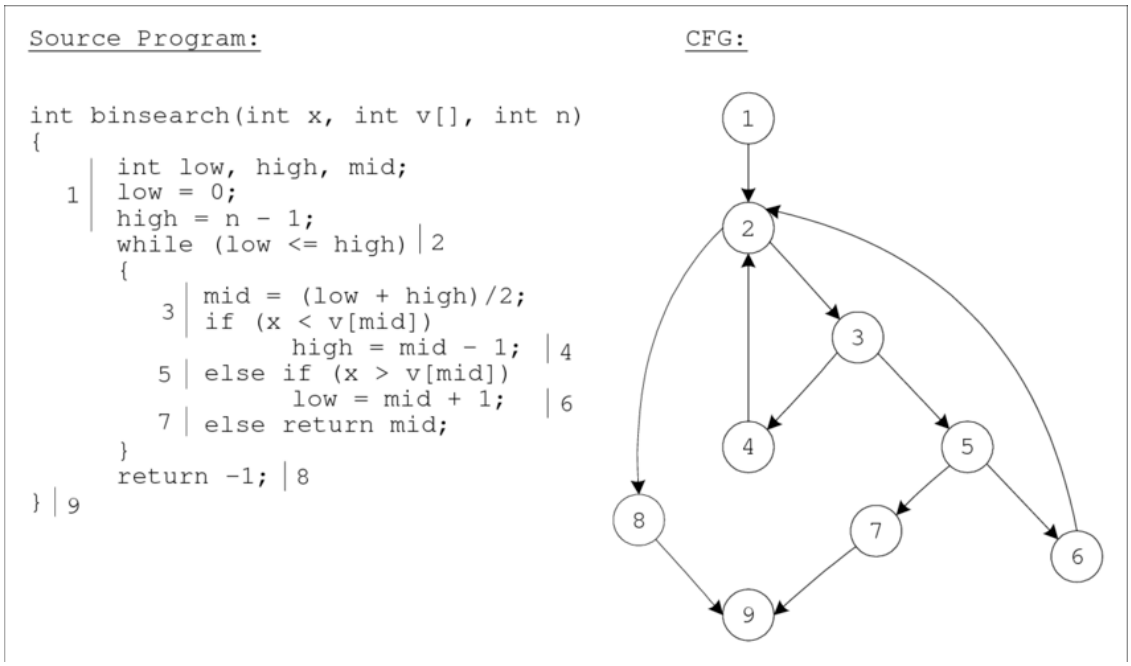

Figure 2.6. Example of Control Flow Graph

scope of Control Flow Integrity is to assure that the CFG is not violated with unforeseen branches and in case halt the system or reestablish the normal flow of execution. Therefore, a CFI solution has to:

1. Build ahead of execution a CFG of the program
2. For each branching operation, check if it is whitelisted in the edge table of the graph
3. Perform action in case it is not allowed.

2.3.1 Software-based Mitigations

Since CFI enforcement has been first implemented in software, we analyse some software solutions in this section. The first proposed implementation is the one from [1], where Abadi *et al.* describe a solution based on simple *Code Instrumentation*. In their paper, they show two alternative methods to wrap branch operations with some machine code that checks whether the destination of the jump is valid.

Note that it is not necessary to instrument any single branch instruction, since those instructions that are eligible victims of CRA are those that take a computed argument that can be manipulated by an attacker, like `jmp eax`. Thus, the instrumentation has to be performed only on indirect jumps.

Still, the Control Flow Graph pruned of direct jumps can be still very complex for certain programs, and this produces an invalidating overhead on the execution. Therefore,

Source				Destination		
Opcode bytes	Instructions			Opcode bytes	Instructions	
FF E1	jmp	ecx	; computed jump	8B 44 24 04 ...	mov eax, [esp+4]	; dst
can be instrumented as (a):						
81 39 78 56 34 12 75 13 8D 49 04 FF E1	cmp jne lea jmp	[ecx], 12345678h error_label ecx, [ecx+4] ecx	; comp ID & dst ; if != fail ; skip ID at dst ; jump to dst	78 56 34 12 8B 44 24 04 ...	; data 12345678h mov eax, [esp+4]	; ID ; dst
or, alternatively, instrumented as (b):						
B8 77 56 34 12 40 39 41 04 75 13 FF E1	mov inc cmp jne jmp	eax, 12345677h eax [ecx+4], eax error_label ecx	; load ID-1 ; add 1 for ID ; compare w/dst ; if != fail ; jump to label	3E 0F 18 05 78 56 34 12 8B 44 24 04 ...	prefetchnta [12345678h] mov eax, [esp+4]	; label ; ID ; dst

Figure 2.7. Branch Instrumentation proposed by Abadi *et al.* in [1]

once the principles of CFI were established with Abadi's work, the objective of researchers has been from that point on to improve implementation efficiency.

Two other solutions, notorious in literature, are *Control-Flow Locking* by [3] and *Control-Flow Lazy Check* by [8].

In Control-Flow Locking, the principle is to protect branching sites with *locking* and *unlocking* operations. This is similar to mutexes, though, in [3], the authors suggest that atomicity and waiting are not two necessary conditions. The first and simplistic approach they show is to use as locking variable `k` a *key* which can be 0 or 1. The locking operation (`k = 1`) will be performed before each *indirect* branch, while the unlocking operation (`k = 0`) is performed after each valid indirect transfer target. Of course, before acquiring the lock, it is necessary to check the the lock is available (`if(k != 0) abort();`). This means that the attacker's freedom of redirecting code is greatly limited to valid control-flow transfer sites.

According to the authors, it is possible to reach full CFI enforcement with a key with multiple bits. The locking and unlocking operation, in this way, can make distinction between

- directly-callable functions return instruction
- indirectly-callable functions return instruction
- indirect function calls or jumps

The actual implementation of the different typologies is show in Fig. 2.8, which is taken from [3].

In SPEC CPU2000, the second methodology proves to give less overhead in general with respect to the first solution proposed by Abadi *et al.*. In terms of security, in [8], the authors claim, though, that in Control-Flow Locking, an attacker can *"redirect control flow to the assignment instruction in the lock code, so the multi-bit control flow locking is turned to be the single-bit one"* [8]. Chen *et al.* have proposed, in the same work, a different approach, called *Control-Flow Lazy Check*. This solution, as the name suggests,

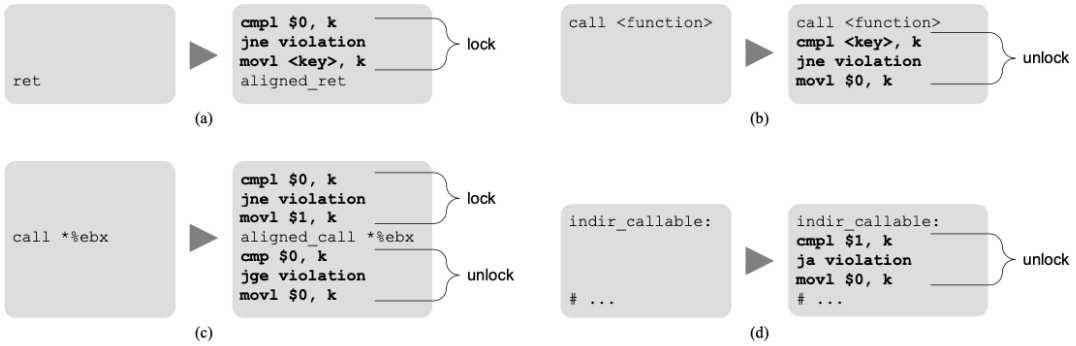

Figure 2.8. Code Instrumentation proposed in [3] in the different typologies of control transfer sites.

aims at detecting Control-Flow violations after the control transfer has occured and before the the code can branch again from that site. Control-Flow Graph is built with an array of pairs of addresses. The first address is the address of the *current indirect branch*, while the second is the address of the *next direct or indirect branch*. This pair is inserted in the array for each two consequent branches (of which the first has to rigorously be indirect).

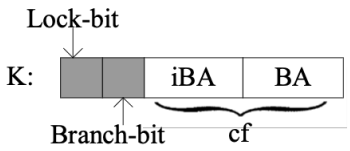

Figure 2.9. Layout of K

During execution, the code is instrumented with various checks around each branch instruction. A small container, named K in the paper, represented in 2.9, stores information of current and previous branch during runtime. At each branch, it is checked on K whether the previous branch was a direct or indirect branch. In case it was not, the current branch address is stored next to the previous branch address. The two addresses are compared to the addresses of the corresponding entry of the CFG, and if they do not match, then code redirection is detected. In order not to allow the attacker to tamper with the checking code, a lock-unlock mechanism wraps everything.

The authors of the paper have called this method *fine-grained*, since these checks are present at every branch site. They noticed that this solution brought also unacceptable overhead to the execution of programs. For this reason, they came up with the idea of *coarse-grained Control-Flow Lazily Check*. In this case, the main checking operation is performed *every two direct branches*. In [8], the authors claim that for an attacker who wants

to perform a complete intrusion, she has to reach a system call. This assumption makes possible the relaxation of checking the code between two direct branches. As depicted in **??**, K now stores the sequence of indirect branches between two direct branches. Whene the sequence concludes (a direct branch is reached), the indirect branches sequence up to that point is checked against the one present in the CFG, at the corresponding entry

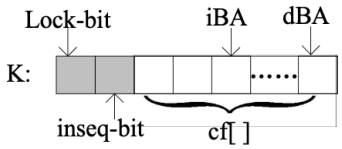

Figure 2.10. Layout of K in coarse graind CFLC

To conclude the discussion about Software mitigations, [8] presents a table describing the response to different attack techniques of the various methodologies explained until now. It is possible to notice how this last two approaches grant protection also against shellcode injection. The table is here reported in Fig. 2.11

Attack techniques		**Defenses (enforcement of control flow integrity)**			
		CFI	***CFL***	***CFLC(fine-grained)***	***CFLC(coarse-grained)***
code reuse attacks	ROP JOP	detected before current branch	running current branch instruction, detected before running targets or next branch	running current branch instruction, detected before next branch	running current branch instruction, detected before next direct branch or system call instruction
	control flow transfer directly into system call	detected before current branch	detected before running system call instruction	detected before running system call instruction	detected before running system call instruction
	RILC	detected before current branch	running current branch instruction, detected before running targets or next branch	running current branch instruction, detected before next branch	running current branch instruction, detected before next direct branch or system call instruction
code injection attacks	shellcode contained with direct branch	none	none	running current branch instruction, detected before next branch	detected before current branch
	shellcode contained with indirect branch	none	none	detected before current branch	running current branch instruction, detected before next direct branch or system call instruction

Figure 2.11. CFI Software Implementation comparison

The main drawback of software approaches is still unsolved, because each of them, in relation to the structure of the program they instrument, gives an overhead which in some cases may be unacceptable. These are not the only solutions based on software, of course. There exist some other techniques that are more platform or OS dependant. We are not going to discuss about them, but just to cite a few, [14], [19], [18], [33] are implementations of CFI methods which strongly depend on the Operating System and on the architecture ([14] and [18] are specific to smartphones).

In the next section, we are going to explore some hardware solutions and compare the performance impact with respect to the software counterparts.

2.3.2 Hardware-based Mitigations

In literature, many hardware based solutions have been proposed. In contrast to software solutions, modifications or extensions of the existing architectures are necessary. This makes hardware implementations hard to develop, test and deploy. We are going to discuss some of them in a wider perspective, dividing them by categories.

The first kind of hardware implementations is based on the encryption of the return address or the target instructions of an indirect jump. In this scenario, we find works like [21] and [25]. The two papers describe similar approaches, but with slightly different circuitry implementations. In Fig. 2.12, a general schema of the protection mechanism is showed.

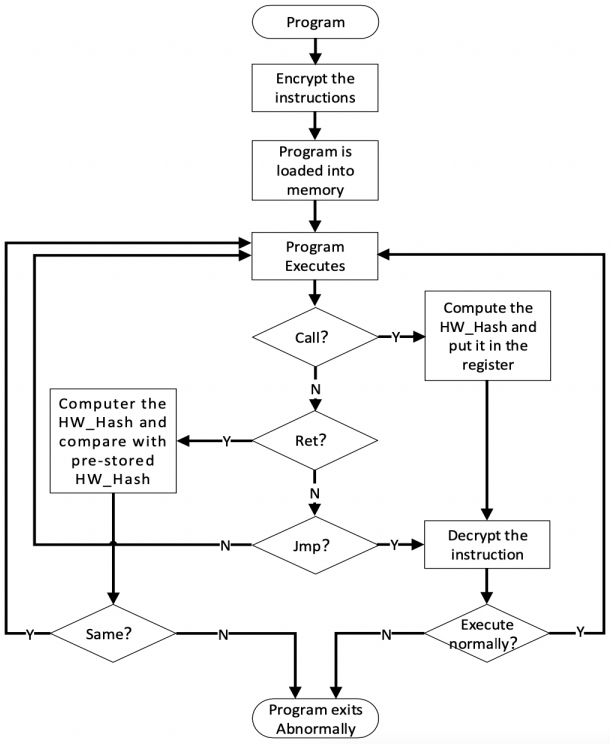

Figure 2.12. General schema of Branch protection through encryption

It is necessary to make a distinction between function calls and `jmp` instructions while considering this approach. When a `call` instruction is executed, along with the normal effects on the stack and the program counter, the additional modules described in those works encrypt the return address and push it onto the stack. In [21], there is an additional stack to which these encrypted addresses are pushed, while [25] simply pushes them onto the standard program stack. When `ret` instruction is executed, the *encrypted return address* is popped, decrypted and stored in the program counter. In case of a ROP attack, if the attacker has tampered with the address onto the stack, it will result in an invalid address

when the decryption module will decrypt it and the program will fail. In [21], the address is popped from the second stack, decrypted and then compared with the address present on the normal stack. If there is a mismatch, code-redirection is detected.

In [10], a different approach is adopted. The authors theorize a modification to CPUs by paring each byte with a *taintedness bit*. These bits are used to instrument CPU registers and memory. Whenever the bytes come from external inputs, like keyboard or network, their relative taintedness bits are marked. The general principle that the authors suggest is that *no tainted byte should end into the program counter*. This method requires not only a radical change of internals of processors and memories, but also operating systems' APIs shall provide support for such functionality and mark as tainted bytes that come from certain sources.

Another family of solutions make use of the open source architectures OpenRisc and RISC-V, which provide processors' architecture implementations under open source license. This makes theorization of hardware solutions simpler for researchers. In this environment, we are going to discuss about three solutions which present security mechanisms against ROP and other code-reuse attacks. The above mentioned articles are [16], [5], [2]. They all have in common an approach to defend against ROP attacks, which is a *Shadow Stack*. This stack is only used for function calls and preserve the integrity of the return address. For example, [2] implements the secondary stack with an additional module inside the cpu. The module provides custom instructions which allow to store and retrieve the return address and syncrhonize with the system stack. When the return address is pushed onto the normal stack, this is also pushed onto the secondary stack. When the function returns, the address is now popped directly from the secondary stack. In [5], the process is similar, though, when returning from the procedure, the address stored in the secondary stack is XORed with the original stack address. This allows to detect an intrusion, and act accordingly. The authors in [5] also describe a Finite State Machine which synchronizes the circuitry of the second stack with the execution of the program. In [16], FIXER also pushes the return address on the stack and compares that with the one stored on the original stack. The paper also mentions how the code shall be instrumented during compilation phase to allow the configurable module to interact with the code. Instrumentation code is put at function calls and return instructions and it consists in tags that are expanded preliminarly to compilation. Tags expansion parses the assembly code and inserts instructions specific to the configurable module.

In [16], *forward-edge protection* is also implemented via a *"policy matrix"*, which contains, for each function in the program, entries corresponding to the functions that it can call *via function pointers*.

Along with solutions that make use of a secondary stack to protect return addresses and function calls, there is in literature another model of mitigations, which aims at protecting *the entire basic blocks inside a program*. In order to do so, it is necessary to collect information about those basic blocks premiliminarly to code execution. The following solutions all present a method to collect information and metadata about the program execution. For the sake of the discussion, we are going to focus on the impact that the solutions have on the existing hardware designs and on the implementation of the basic block protection.

The authors in [7] present a solution implementing a module external to CPU. The interfaces with the main processor send to the module program counter updates and instructions, while the module responds back with possible Control-Flow exception. In Fig.

2.13 it is possible to observe the general schema of the internals of the module.

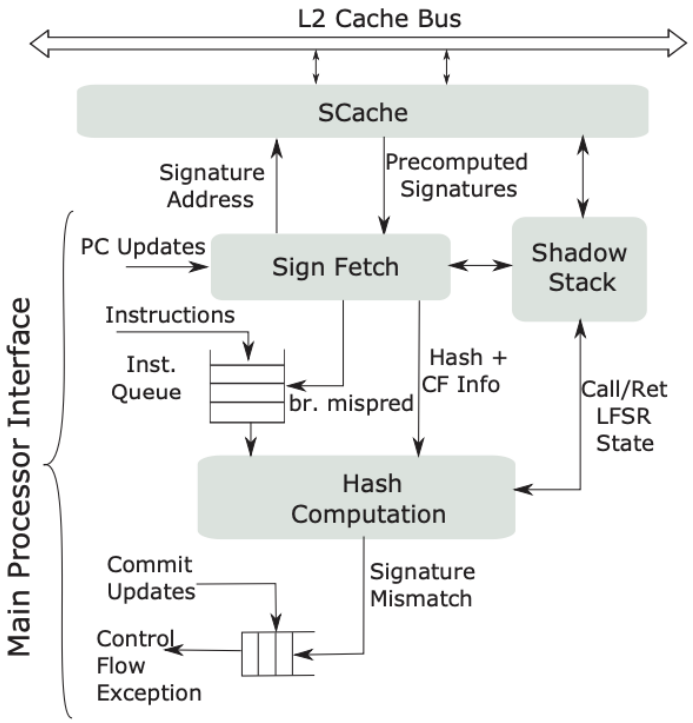

Figure 2.13. Implementation of external module presented in [7]

The precomputed hash is taken from the SCache and then it is compared with the signature computed at the end of a basic block. As soon as instructions are decoded, they are fetched by the CFI co-processor and the cumulative signature is computed. It is important to notice that this implies computation of signature with instruction prefetched for branch prediction. It is then important to evaluate the result of the hash comparison on the basis of a misprediction of the branch. If the branch is taken, instead, a mismatch will be signalled to the CPU as a Control-Flow Exception.

In [12], a similar approach is taken. The run-time hash is alwasy computed at the end of the basic block. What changes here is that along with the basic block instructions, there is also some metadata inside the hash computation, which contains information about branches and calls.

Another class of solutions proposes changes in the *branch-prediction mechanisms* inside scalar and super-scalar processors. In [20] and [29] this direction is followed as well. Branch-prediction is what allows modern CPUs to fetch instructions located at the target of a conditional branch and start processing those instructions, even if the branch will not be taken; in such a case, the CPU will simply revert its status at the point before the jump, and will now fetch the correct branch of instructions. The number of cycles wasted in the

pipeline is the same as if the processor had stalled until the branch was evaluated. In the case, though, the prediction was correct, no cycles in the pipeline have been wasted, and the result of the speculated instructions can be commited.

The author of [20] establishes its work in this spectrum. By exploiting the already existing branch prediction mechanism inside processors, the authors propose a module parallel to the Branch Prediction Unit - BPU. The link between CRAs and Branch Mis-predictions in instructions execution lies in the fact that *the former indistinguishable from the latter, from the BPU perspective*. If considering normal execution, branch mis-predictions are accounted as standard. When considering, though, unintended program behaviours, any branch mis-prediction can be considered a potential intrusion. In [20], a *Mis-prediction Validation Unit* is put beside the BPU. When a mis-prediction is discovered, the MVU checks if it was due to an invalid code redirection. It is of course necessary to provide the module with a list of valid branches, in order to validate the mis-prediction. The authors, though, do not provide any further detail about the Control Flow Graph generation.

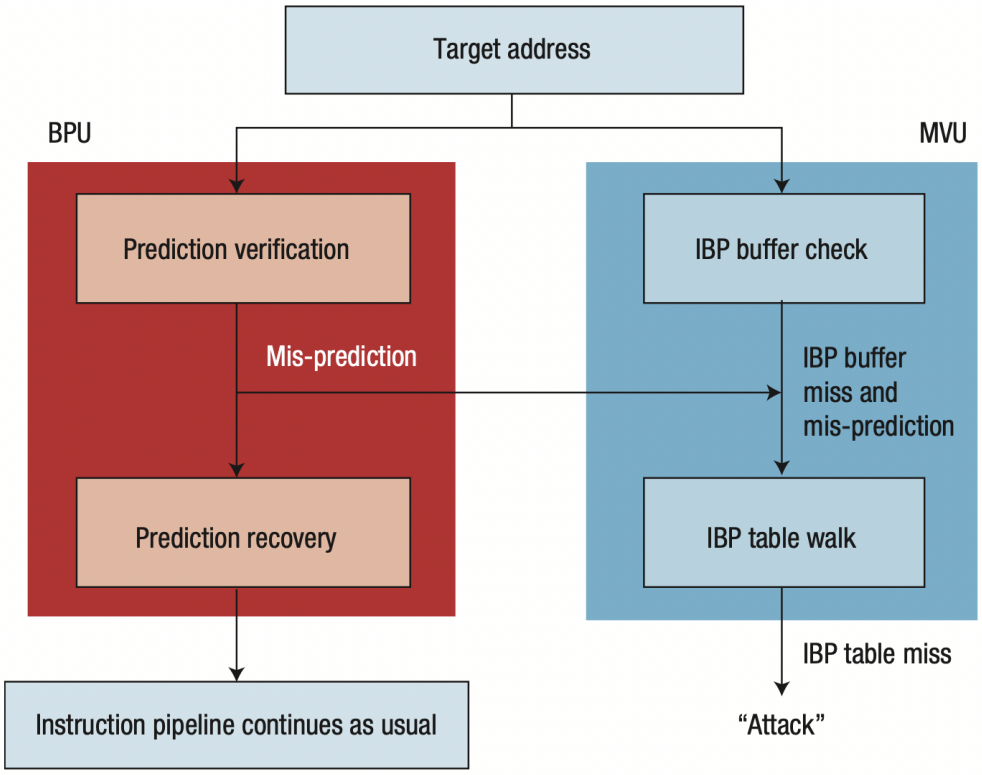

Figure 2.14. Mis-prediction Validation Unit. Image taken from [20]

In order to conclude this disseration about techniques to protect from Code-Reuse attacks, we are going to include in the list of hardware solutions the works presented in [11], [13], [30], which all have in common the extension of the Instruction Sets of the SPARC

processors to implement CFI solutions directly in hardware. The additional instructions provide protection mechanisms similar to the ones discussed in all other solutions; the point here is that code instrumentation is directly interpretable from the processor as machine code, that will act directly on the hardware modules used to offer the protection mechanisms (like shadow-stacks).

After having acquired a wider panorama of the current state of the art, it may be easier now to understand where the solution that we are going to explain in the followings of this thesis work can be positioned. As better explained in section 3.6, the solution presented at our department tries to embody some of the advantages both from the software and the hardware typologies.

It should be clear though at this point, that there exists no solution which can guarantee CFI at low cost. The trade is basically between performance and hardware complexity. It is also important to keep in mind that any solutions may provide false positives, thus detecting CFI violations where they actually don't exist. Finally, nearly all solutions require code instrumentation before letting the code run under the protected environment. The extrapolation of a CFG is fundamental in order to guarantee a correct detection of intrusions. It is indeed an active area of research, the creation and implementation of the CFG. In the area of embedded systems, where memory availability requirements may be quite stringent, it is crucial to devise a CFG which does not occupy considerable space.

Chapter 3

FPGA-based Control-Flow Integrity (CFI) for Microcontrollers

3.1 General Features

In the wide spectrum of the solutions we have presented so far, the CFI technique that this work aims at describing is to be considered as a hardware solution, while of course, some software integration is required. This solution is conceived to be implemented mainly on embedded systems. The target for which it was developed is the ARM family of microcontrollers and it assumes that the firmware to be protected runs on bare-metal, without an Operating System.

The majority of the information that is going to be reported in the following pages comes from the work of Prinetto, Maunero and Roascio in [24].

As the title of their work suggests, the main requirement for this solution to be implemented is a reconfigurable hardware module, like an *FPGA*, which the microcontroller can communicate with. The advantages of having a module which does not feature an immutable design are obvious, while what is most valuable from hardware solutions, *i.e.* performance, must be kept. According to [31], it is expected to see FPGA spread and become increasingly more popular. FPGAs allow to deliver more computational power to devices. The ease of developing dedicated hardware with lines of code is becoming more and more valuable, in a world which sees IoT and 5G as leading technologies at the horizon.

The most used communication scheme between microcontroller and FPGA sees the FPGA memory mapped, and the CPU can access it via writing or reading to an address.

Given this trend, we claim that it is possible to implement a security solution to guarantee control-flow integrity which features several advantages. First, we want to protect embedded systems with microcontrollers, without the support of any OS-provided security feature like ASLR, privileged execution levels and such. In the embedded world, as already mentioned in chapter 1, space and computing resources are rather limited, so the first characteristic a security solution should have is to be able to be run on *bare-metal*. Often, the strict timing requirements, like in *real-time systems* impose the use of software

which does not make use of abstraction. It is, then, important to provide a solution which does not heavily impact performance, thus it is the least invasive possible.

Though nowadays, open hardware designs are available, and any company with good profits can develop and create its own custom microcontroller, the expenses of engineering and production can still be very hard to front. This is why an FPGA-based solution which is architecture, board and platform independant is optimal.

Many solutions presented in **??** relied on creating and storing secrets. It is however possible, with sufficent means and costs, to overcome the memory and IC protection defenses. Our solution does not use any kind of key or challenge and overcomes this threat.

Finally, as described in [23], it is important to take in consideration interrupts, which, being asynchronous, can happen at any point during program execution, compelling control-flow integrity solutions to reconsider many assumptions about branch targets.

3.2 CFG Edges Identification and Categorization

The solution we are presenting here includes specification on how to instrument the code to be protected. Code Instrumentation, as already seen in chapter 2, in Control-Flow Integrity, means to wrap crucial points with some labels/instructions that support or give information to the main protecting algorithm. Here it is not different. First, though, we have to carefully understand what are the points that require code instrumentation.

The Control-Flow Graph is made of nodes and directed edges. Being the nodes the basic blocks of the program, the edges represent the control-flow transfer among these nodes. A first classification of edges is:

- forward edges
- backward edges

The former are representing jumps and function calls, while the second are function returns. In terms of CPU operations, in forward edges, the jump/call instructions simply take an argument which is the address or the relative offset of the location at which the edge points. This location is called the *target* of the edge. This practically results only in the modification of the program counter. Backward edges instead, having to return to a location which was previously computed/store, take their target from data memory (typically from stack).

It is then necessary to differentiate *direct edges* from *undirect edges*. Direct edges are edges representing jump/call instructions which have as argument hardcoded labels or addresses. Indirect edges are the edges where the target has to be computed and passed to the jump or call instructions by register or memory.

Direct edges do not require any kind of protection. They are secure by default, as long as the code section is not writable. Indirect edges instead require further discussion. In [24], the authors, after a series of assumptions in line with the characteristics of embedded devices, arrive at the conclusion that, quoting, *"it is always possible to list all the destinations of all the edges of a CFG"*. Hence also indirect edges can be classified as secure, in specific conditions. In order to arrive at this assertion, let us introduce the *origin tree*. The origin tree is a representation of data dependancy of the content of a register or memory address. In Fig. 3.1, an example is reported.

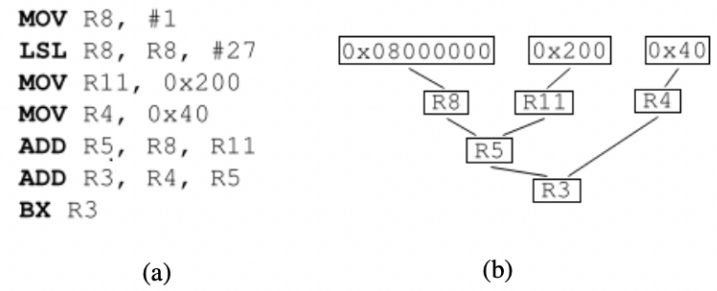

Figure 3.1. Code Snippet and origin tree of the R3 register

At the root of the origin tree there is the register of interest of which we want to trace the origins back. The nodes are the memory locations or register that have contributed in the current content of the register. This representation, thus, allows us to spot potential vulnerabilities in the construction of a value of a register. The leaf of the tree are locations where it is not possible to trace back anymore because either are constant values or are inputs taken from the outside of the program. Here the authors of [24] assume that

- The entire program binary is stored in memory, statically linked.
- Code is immutable
- External inputs are never used directly as code pointers

Under these assumptions, it is possible to derive some conclusions. Being the program statically linked and present in memory in its wholeness, *"the construction of the origin tree is always possible, no matter the complexity in constructing it"* [24]. We can also deduce that, if code is not writable and no function pointer can be arbitrarily crafted directly from the user input, *"the set of targets of an edge is always finite and enumerable"* [24]. With these postulates, we have now the means to identify what are *secure and insecure edges*. It is straightforward, as already seen in the previous chapter, that direct edges, which take a constant value as target location, are secure by nature. If code memory cannot be modified, that instruction will always jump to that target, thus its origin tree will be composed by a single node. In the case of indirect edges, we can make a further distinction, in contrast with the other works we have presented so far. *Not all indirect edges have to be considered vulnerable*. Indeed, with the help of the origin tree, we can trace back the origin of a value and look inside the tree if that value was tainted at some point. This is similar to the approach taken by [10], but we are not considering any fundamental hardware modification here.

Now the challenge is to understand how can we identify the nodes where the value of the root can be tainted. A conservative approach is to look for nodes which are present in the data sections of the binary, being them the only areas where the code can actually be modified by the user. This would still overshoot the problem, but up to now, there is

no other solution to be more precise and identify vulnerable points. It would necessitate to have a pattern recognition mechanism, backed by a database of vulnerabilities, that is able to recognize portions of code that may actually be considered dangerous. As there is no such kind of techniques that is consistent and reliable enough, the authors of [24] have decided to adopt a conservative approach in the implementation of their solution, which anyway is much less conservative than the majority of the solutions taken in consideration up to now. Note, for example, that a backward edge, representing a return operation, is always an indirect branch, but it is not necessarily insecure. In ARM instructions, if the return address is stored in `lr` at the beginning of the function, and that register is never pushed onto the stack, that will still be a secure edge, according to the origin tree.

3.3 The Problem with Asynchronous Calls: ISRs

Up to this point in the discussion, we have considered CFI as a property to be guaranteed in the context of applications executing in a relatively predictable and determined state. The flow of execution, as intended by the programmer, if well coded, has a finite number of branches with a finite number of target each. In such an environment, this is assumptions are enough to theoretically protect Control-Flow. In practice, there are many contexts, where *the flow of execution is not predictable by design*. It is the case of *asynchronous interrupts*. Interrupts are now a wide-spread feature also for embedded devices, to be considered essential in the case of real-time systems.

The main challenges to be front when dealing with Interrupt Service Routines are the following:

- Asynchronous function call
- Context Switch

The first challenge is the one that causes major problems in our scenario. The solution in [24], as many others, relies on code instrumentation in critical points of the code where a branch instruction may be manipulated from an attacker. Withouth any unpredictable behaviour, it is already challenging to identify these points. Now that we are introducing an undeterministic factor, the asynchronous interruption of code-execution completely invalidates all the static analysis assumptions taken on the code in the offline-phase.

The second challenge is that, when serving an interrupt request, the currently running execution context has to be preserved, since likely the current procedure or task will not have reached a final state. This is why registers that are fundamental to the current context have to be preserved. When an interrupt is raised, modern processors and microprocessors, (*e.g.* the ARM Cortex-M series) automatically perform a *Context Switch*. This operation saves onto the stack the main registers (in case of ARM processors, `r0`, `r1`, `r2`, `r3`, `r12`, `lr`, `pc` and `xpsr` are pushed), and then load the address of the interrupt service routine inside the program counter.

This means that a memory vulnerability inside the interrupt service routine can be exploited by an intruder, which can now manipulate not only the return address, but also other registers like the status registers or registers containing function arguments.

If not by disabling interrupt, or by using polling methods to verify peripheral states, a technique to preserve context is necessary.

3.4 CFI Monitor Architecture

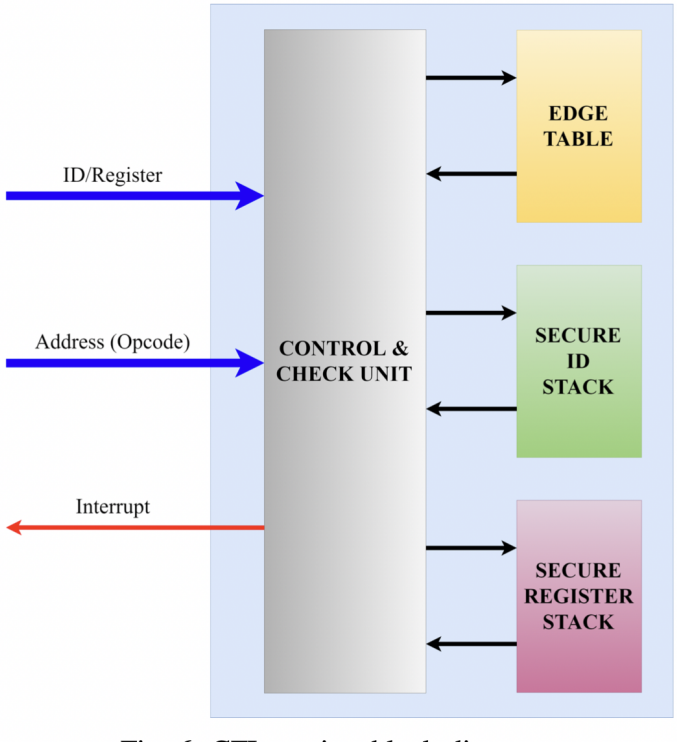

Figure 3.2. CFI monitor Architecture. Image taken from [24]

The monitor architecture, depicted in Fig. 3.2, is based on a High Level State Machine synthesized onto an FPGA. The interface of the monitor with the CPU consists of:

- A bus from the CPU transmitting data
- A bus from the CPU transmitting addresses
- A single-bit line which can raise an interrupt in the CPU

This simple interface is what makes this solution simple to implement in existing architectures. In order for it to work, a simple write instruction (*e.g.* a `store`) has to be issued from the CPU. This instruction sends on the bus an *opcode* which will be interpreted from the monitor in order to execute the proper operation.

Internally, the FPGA is composed of:

1. CFG edge table
2. label ID stack

3. register stack

The first table is prefilled in the offline phase with the valid edges that constitute the CFG. The second is a stack where the different call location IDs are pushed. This stack is similar to the one described in other solutions in section **??**, and it serves similar purposes. When a function is called, but it may be called in different code locations, this stack allows to correctly identify the caller ID, in addition to verifying that the return address belongs to a valid edge address pair. It is also useful when different functions are called one from the other and stack frames start to increase.

The last stack is the register stack, which is used to save the execution context before a context switch. The registers that are automatically saved onto the stack when an interrupt request arrives to the CPU are automatically saved onto this stack as well. When returning from an interrupt service routine, these registers will be restored from this stack.

All these components are managed and orchestrated by the *Central Control Unit*, which receives signals from the CPU and is in charge of sending interrupts to the CPU in case the HLSM reveals an invalid code-redirection.

Since the checks performed by the CFI monitor happen in parallel with respect to the normal execution of the CPU, if strict timing constraints are not set, an attacker could redirect code where instrumentation is not present and still be able to execute some instructions. For this reason, the CFI monitor, triggers a stringent timer that takes into account the execution of branch instructions and instrumentation code. When the timer expires, the CFI monitor expects to receive data regarding the target of the branch instruction. If no data is received during this interval, the CFI monitor will halt the system, alerting that an invalid code-redirection has happened. CPU and FPGA share the same clock source, so the monitor simply has to consider how many clock cycles each instruction of the code instrumentation adn branch instruction take. While executing instrumented code, interrupts have to be disabled, since this would certainly make the timer expire without any target code instrumentation being sent.

Finally, after the offline phase, where the FPGA monitor is set and code instrumented, any read or write operations to the FPGA are disabled, so that it is not possible in any way to modify or tamper the structure of the CFI monitor.

3.5 Phases of the Protection Mechanism

The protection mechanism of our solution is divided in two phases, as described in [24]. The first phase is an *offline phase*, which is executed before that the code is run on the system. This is a crucial phase, since the structure and *critical points* of the binary that will be executed are analyzed. When the code is run, the *online phase* starts. Here is where all the preparation of the offline phase will be exploited to guarantee protection against code-reuse attacks. The code instrumentation previously laid on the binary is now executed at every critical point and it will put the CPU in communication with the FPGA, by sending addresses and data.

Let us first see what code instrumentation generally does. After the program is compiled to assembly code, a static analysis tool will be used to spot *critical points*, which are the points where edges that need protection in the code lie. After these have been identified, the corresponding source and target basic blocks are given a unique *label ID*, which may

be the hash of the basic block position in the code. These labels serve to identify edges by basic blocks pairs, which will be stored inside the *Secure Edge Table*, described in section 3.4. After labels have been created, the code is instrumented on the base of the typology of critical point. There are 9 possible critical points types, each of which may need different instrumentation. They are listed in [24]

1. *Forward edge with single target*: before the control-flow transfer happens, the ID of the source basic block is sent to the monitor, while the ID of the target basic block is sent after the transfer. These two IDs will be used to generate a pair representing an edge which will be looked up in the Secure Edge Table. If the search misses, the transfer is invalid and CPU will be halted due to code-reuse intrusion.

2. *Backward edge with single target*: this case is not much different from the above one, since it is a simple branch which can end only in one point in the code

3. *Forward edge with multiple targets*: when a basic block can branch to different targets, all of the are instrumented, and in Secure Edge Table, there will be many entries with the same source BB and different target BBs. In this case, it is not possible though to exactly predict where the branch should end, as this requires complex dynamic analysis. Still it reduces strongly the availability of gadgets in the hands of the attacker.

4. *Forware secure edge to a function with backward edges with multiple targets*: This Critical point does not need protection, because the branch cannot be manipulated, but since the function been called is capable of returning to multiple locations, it is necessary to save the caller ID. This will be pushed onto the Secure Label Stack, in order for it to be retrieved later when the function returns.

5. *Backward edges with multiple targets*: This is the follow up of the previous situation, in which a function with multiple potential callers has been called. In order to recognize who called it, the ID of the caller is popped from the Secure Label Stack. If the caller ID does not correspond to the target ID, an exception is raised and execution is halted.

6. *Forward edges to a function with backward edges with single target*: Since the source BB here has to be protected, the BB ID of the caller is sent to the monitor and will be checked if the target is a valid target. The same operation will be performed when returning from the function

7. *Forward edges to a function with backward edges with multiple targets*: Same as above, and the function ID label will be pushed onto the Secure Label Stack.

8. *Entry point of an Interrupt Service Routine*: All interrupt service routines are instrumented at their entry points. The instrumentation performs a series of `store` operations that save all the registers that are automatically pushed onto the stack when a context-switch happens. These registers are saved onto the Secure Register Stack.

9. *Exit Point of an Interrupt Service Routine*: When returning from an ISR, a number of `load` operations equal to the number of `store` operations performed at the entry of the ISR is executed, restoring the content of the registers as it was before the context-switch. These values popped from the Secure Register Stack is compared with the content of the register present on the original stack. If they are different, an intrusion is detected.

When the analysis phase is completed, an instrumented binary and a file containing the edge table will be generated. This file has a `.mif` extension and will be given to the synthesizer in order to generate the ROM that will constitute the Secure Edge Table. The Synthesizer creates a *bitstream* which is passed to a *secure boot loader*, which sets up the connection between the CPU and the FPGA and sends the bitstream (contained inside an array) to the FPGA, in order to program the monitor.

In Fig. 3.3 a summary of the phases described until now is present.

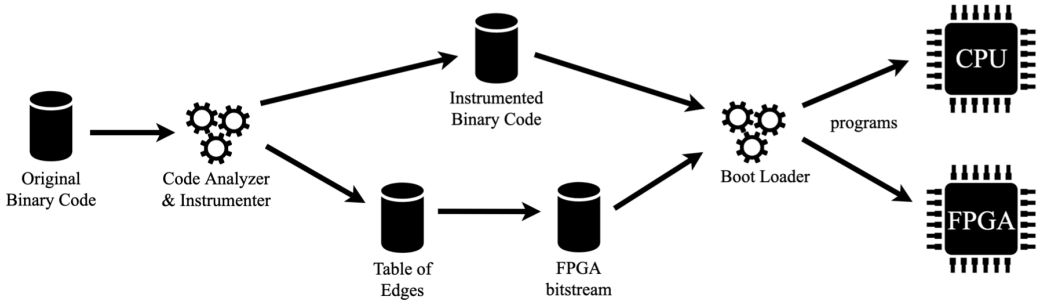

Figure 3.3. Offline steps, image taken from [24]

3.6 Advantages of the Hybrid Approach

When discussing about the category of mitigation this solution belongs to, we may say that it is both a hardware and a software solution. It possess the most important features of both the categories, and tries to reject their disadvantages. It is based on code instrumentation and on an additional hardware module. Software solution applications may result in excessive overhead for certain types of systems, because in order to obtain the desired robustness and protection, many instructions per branch should be added. Depending on the nature of the binary instrumented, the overhead will scale proportionally with respect to the number of branches present in the code. Even if the assumption that some branches do not require protection is taken, this may still be not sufficient to have a measurable degree of control over the overhead. This is the fundamental flaw of software solutions. In solution presented in [24], code instrumentation is present, and so is the overhead, but it is extremely reduced to the necessary instructions that are required to perform a `store` operation at a certain address.

In hardware solutions, there is no such overhead, due to the fact that dedicated hardware takes care of the meaningful functions. Nevertheless, hardware is fixed and immutable. Implementing a solution in hardware fits only one platform, requiring to implement a different design for different architectures. In solutions described in **??**, some of them redesigned various internal stages of modern CPUs. This extensions/modifications of already proven designs is expensive and requires producer to add support for such functionalities.

Our solution avoids these problems by mixing the two approaches. Having a reconfigurable hardware module makes it possible to synthesize a design and change it whenever an update is needed, all without degrading performance, which is hardware mitigations' apanage. With reconfigurable hardware modules it is always possible to implement new opcodes, new data structures and correct errors. Another advantage of this solution having two separate components (*i.e.* code instrumentation and on-FPGA protection mechanism), allows to deploy the same FPGA design on different platforms, by just adapting code instrumentation to the current architecture.

The other advantage of inheriting software mitigations peculiarities lies in the fact that it is possible to *trade between performance and security*. It is indeed only necessary to instrument branches that are critical from the security point of view while leaving unprotected those that are critical performance-wise. According to [24], this choice could be justified by the fact that those branches that require not to be overloaded with protection instructions can be effectively engineered differently, in the attempt to fundamentally patch vulnerabilities.

Our solution, in the end, does not provide the highest performance or greatest adaptability and scalability, but aims at being the most practical and efficient way to properly deploy a CFI solution which does meet performance constraint of all possible systems typologies *and* is flexible enough to be widely adopted by manufacturers without grand efforts.

Chapter 4

A Real-World example: Attack and Protection of the SE*cube*™ Chip

4.1 Device Architecture and Features

After a theoretical discussion of the inner workings of the CFI monitor, the goal of this thesis work is to demonstrate practically the implementation of such method onto a real embedded system. As already mentioned, in order for this solution to work, a particular hardware configuration is required: a microcontroller which is capable to communicate with a reconfigurable hardware module to exchange sensitive data with, as briefly depicted in Fig. 4.1.

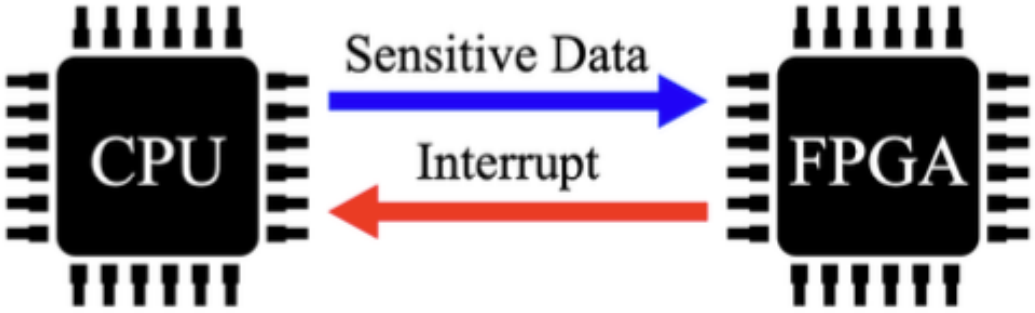

Figure 4.1. Simple communication schema between CPU and FPGA, taken from [24]

The solution we resorted to was the SE*cube*™ chip, provided by Blu5 Group®. It is a 3D System-in-Package, providing the following features:

- A STM32F4 microcontroller with an ARM Cortex-M4 core, 2MB of Flash and 256 KB of SRAM
- A MachX02 FPGA by Lattice Semiconductor™ with 240 KB of SRAM and 7000 4-bit LUTs
- An EAL5+ Certified Smart Card

The STM32 family of microcontrollers is wide-spread in the field of embedded systems, so this architecture was chosen also because it reflects a real-world scenario. The SE*cube*™ comes with an Open SDK, available at [27]. The SDK provides core functionalities for the main purposes of the SE*cube*™, which is intended to be use as a slave in conjunction with a host to which it provides security features like key-management and encryption.

As we will see in the following sections, we have developed on top of this SDK functionalities that mimic those that may be required in a real-world example, in the field of IoT.

4.2 Study Case: A Smart Access Monitor

In order to demonstrate the impact of an attack to such kind of systems and the value of a simple and efficient defense, like the one we presented aims to be, a practical example of how the SE*cube*™ can be deployed, attacked and then protected has been prepared. As a disclaimer, it is important to note, that the work done in preparation of this example has been developed from ground-up, for what it concerns functionalities which were not implemented in the Open Source SDK bundled with the SE*cube*™. No pre-existant code was attacked and the vulnerabilities which were exploited were implemented on purpose and tailored to better represent the risk of a breach inside such systems.

The study case that we have presented is the example of a *Smart Access Monitor*, which may be present inside smart buildings. Such a system should be designed in order to provide different security features, depending on what it is put at the edges of. It may be used to protect the access to an area which is confidential, and as such, requires a limited number of people, with certain privileges, to have the authorization to enter. This is the case in almost every building that hosts offices, hospitals, military and industrial facilities and many more.

In the context in which this thesis work is being written, where COVID-19 has spawned the necessity to limit the number of individuals lying in a certain space, this monitor could be in charge of controlling the access to an area, counting the number of people that have entered it and denying access to any excess person.

In these use-cases, corruption of the system would not only mean the dismantling of the security features that it intends to guarantee, but also the arise of new threats, like data breaches including sensitive information about actors and denial of service. For example, in the case of a company office, a successful attack on a smart access monitor from an intruder could be used not only to grant access to actors who normally would not have it, but it could deny access to people who rightly should be able to enter that area or it could hand information about individuals that are registered inside the database of the system.

In this work, we have imagined a Smart Access Monitor that grants or denies access on the basis of the rights that a user has to access a specific area. This means that the

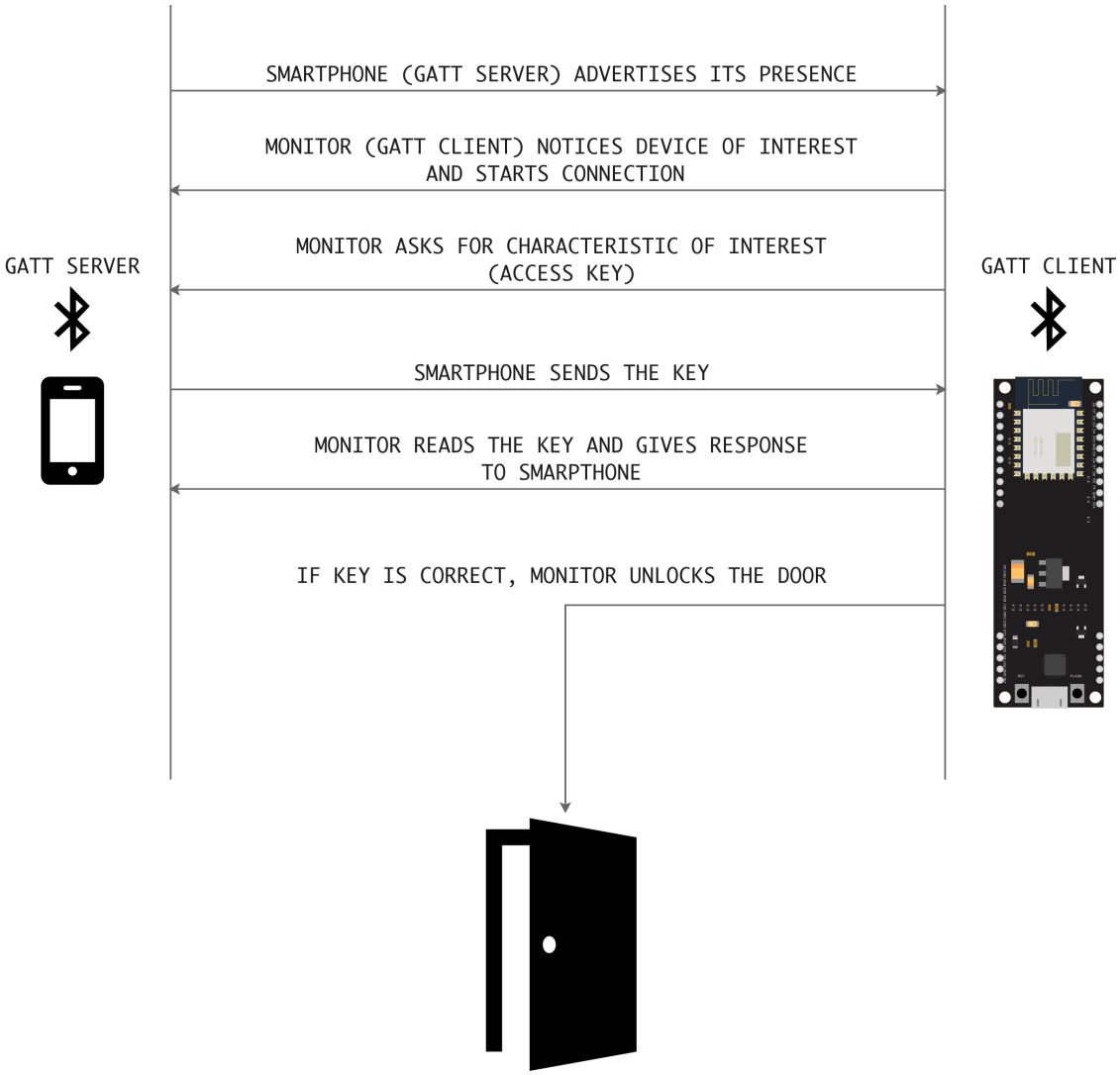

Figure 4.2. Here the architecture of the monitoring system, as well as the exchange of data between the two main parts, is schematized

user must possess a secret that is exchanged with the monitor; the monitor will check in its tables if the secret corresponds to a valid one and will take a decision accordingly. We think of the system as connected to an actuator that unlocks or keeps locked a door that separates the confidential area from the entry. A representation of the system we have devised is available in Fig. 4.2

As the requirements have been laid, we can now discuss about the architecture of such system. As described in Fig., the monitoring system is composed of two main devices:

- A device containing the secret of the person willing to enter

- A device which looks for the secret in its tables and takes a decision

The first device can be a Smart Card, a token or a smartphone. Following the tendency in the world of IoT of interfacing devices with smartphones, due to the practicality that these devices bring with them, these have been chosen as the devices tht are carried by those who are asking for the access permission. We have developed an application for the iOS™ platform. This application acts as a BLE server. BLE is one of the most used functionalities in modern IoT devices. It is power efficient and offers functionalities which facilitate small data exchanges between two devices. In order to exchange the required information, the *GATT* protocol is used.

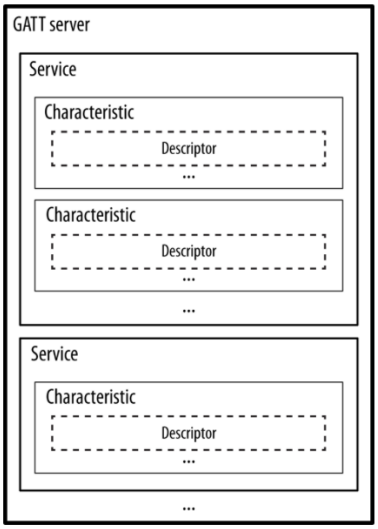

Figure 4.3. GATT server schematic, picture taken from [6]

In the GATT protocol, a server and a client are present. The server advertises its presence by broadcasting periodically some advertisement data on each channel, and if a client is interested, it will connect to the server. When connected to a Server, a client can communicate with it only by reading and writing from and to its *characteristics*. The characteristics of a server are small registers which store data inside. A set of characteristics form a *service*. A server can run one or multiple services. In our case, the smartphone will act as a *server*.

The second device is the SE*cube*™. The Open SDK of the SE*cube*™ exports some cryptographic related functions, but we have relied little on them. We have implemented though the core functionalities to communicate with a BLE server, thus making the monitor the GATT emphclient. The SE*cube*™ DevKit that we have used for this experiment does not provide any BLE module. We have resorted to use an *ESP32-WROOM* DevKit, which the SE*cube*™ communicates with via UART. At [17] it is possible to download the firmware to be flashed inside the ESP32 that exposes the BLE functionalities through AT Commands to be sent over the UART Channel.

The system then works in the following way: a user who wants to access an area first has to open the application on her smartphone to connect to the monitor. The application starts the BLE server and initiates an advertising session. It advertises a particular service name, which is captured by the monitor, that scans every 3 seconds. When it finds the service of interest, the monitor establishes a connection with the GATT server. It asks for available services and characteristics and when the characteristic of interest is found, it writes there its secret. If the secret is valid, the monitor will send back on one of its characteristics a response and controls the actuator of the lock/door accordingly.

4.3 Vulnerability and Exploitation

In order to find a vulnerability, an attacker first has to individuate a buffer that she can fill with arbitrary data, and see if that buffer is unguarder from any boundary checks. In our case, the code presents different points where buffers are created in order to store data coming from the GATT Server. Since we are relying on sending data from our mobile application, which is built on top the BLE framework for iOS provided by Apple™, we have to consider that sending a payload from the smartphone could be limited in length by the framework itself. One of the points where there are loose or no constraints at all is where we exchange the secret key from the smartphone. The key transfer happens via a BLE GATT Characteristic write, so in a secure application it is important that those lines in the code are well protected.

4.3.1 Overflowing the Unprotected Buffer

In our code, as soon as the characteristic of interest has been found, the SE*cube*™ tries to read the characteristic with the `readChar` function, which is our vulnerable function. This function sends the AT command that reads the characteristic from the GATT server. Here, a wrong assumption about the maximum length of the characteristic generates a buffer overflow vulnerability. In order to simplify the exploitation, the vulnerable function `memcpy` was used to demonstrate the issue. For security reasons, the key that is passed from the smartphone to the SE*cube*™ should be encoded in base64 characters or string hexadecimal in order not to contain any binary non-ASCII value. Although this assumption does not protect the reading function, if `readChar` had contained a `strcpy` instead of a `memcpy`, the attacker would have a much harder time in exploitaion phase. The `0x00` (NULL character) terminates a string, thus it is interpreted from `strcpy` as a null termination, and it will stop copying bytes into the buffer. There are techniques to avoid using `0x00` bytes, like XORing a register with itself and others, but the problem lies in the fact that ROP gadgets may contain `0x00` bytes within.

Now that we know the location of a possible memory vulnerablity in the code, we have to craft a payload that will hijack code execution in a profitable way for an attacker. Since the code verifies the secret and then decides which action to take, a trivial redirection would be to redirect code to the branch where it grants access to the user. With a ROPchain, though, severe damage can be dealt, and we wanted to show the worst impact that a well crafted ropchain has.

The `readChar` function declares three variables at its prologue. The buffer to be overflown is the `char value[32]` array. The disposition of variables inside the stack after the prologue has finalized variables declaration is represented in Fig. 4.4

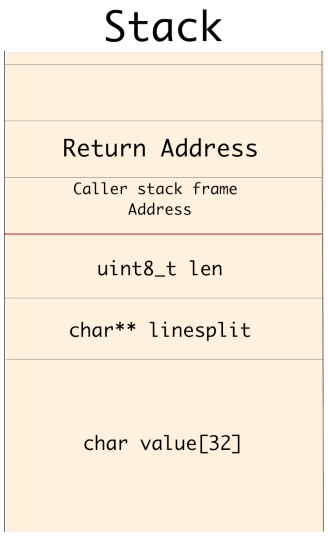

Figure 4.4. This is the layout of the stack when entering the readChar function. The buffer value is 32 bytes long and before the stack base there are two other variables.

4.3.2 Crafting the ROP Chain

The SE*cube*™ comes with a host software that can be run onto a PC and connected to the device. The host program interacts with the SDK and calls functions from it. Inside the SDK, the management and storage of data happens via records, which are allocated into flash memory for persistence. These records contain different types of information, among which there are credential informations. The `record_set` function creates or modifies a record that is identified by a type. The signature of the function is the following

```
bool record_set(uint16_t type, const uint8_t* data)
```

It requires an unsigned which identifies the type of the record and a pointer to a buffer which contains the data that is desired to be stored. A record is 32 bytes long, so data will be read for 32 bytes on. There is a particular record, which can be set from the host program, that is the *Admin PIN*. This has type 0 and is the PIN that is used to access the privileged functionalities from the host program.

Therefore, we have to call this function with `type = 0` and `data` pointing to some controllable memory location where we can read/write. By doing so, we would be able to change the Admin PIN and potentially causing maximum damage, if an intruder can reach to the host functionalities.

In ARM calling convention, in order to pass arguments to a function, they must be put into `r0-r3` registers. What we need is then:

- the address of the function `record_set`
- the address of a location in memory where we have control over the data (like a region of all 0s)
- gadgets that put the arguments inside registers
- gadget that puts the address of the function inside the `pc` register.

Luckily enough, inside the binary of the firmware flashed in the SE*cube*™, there is an instruction which performs the majority of the above-mentioned tasks.

```
pop {r0, r1, r4, r6, r7, pc};
```

It is, indeed, sufficient to put on the stack in order

- the first argument
- the second argument
- three 4-bytes values to be put in `r4`, `r6`, `r7`
- the `record_set` function address.

In Fig. 4.5 the layout of the ROPChain is showed. In order to simplify development, a simple script that automatically generates the ROP chain by analyzing the binary has been developed. The output of the program has been put then inside an array in the code of the mobile application and is then concatenated to a user manipulated string (in the benign application, the user string was controlled by a text field, which is used as the password for the authentication to the monitor; now the user field is used to pad the the first 36 bytes to fill the buffer).

4.3.3 Proof-of-Concept Attack

It is now time to show the application of the theory seen so far. In order to appreciate the correct exploitation of the program, we have to connect to the host program with the SE*cube*™ device. We have already said that the vulnerable function is the `readChar` function, which reads the characteristic that contains the key from the GATT server (the smartphone). The function prologue declares three variables:

```
    char value[32];
char** lineSplit;
uint8_t len;
```

The function reads the data coming from the UART channel and, after some filtering, it reads the data and copies it into the `value[32]` buffer with `memcpy` function. The vulnerable snippet is the following:

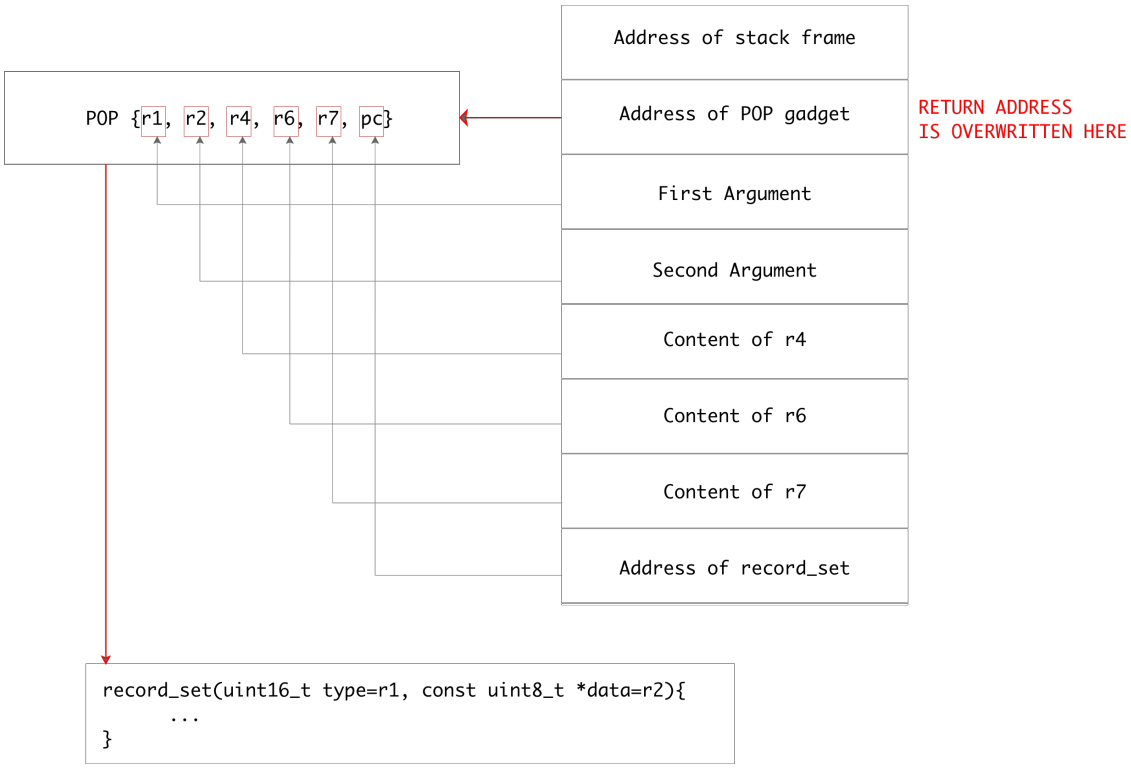

Figure 4.5. Layout of the ROP chain. Red Arrows show the flow of execution of instructions. From the POP instruction, the registers are loaded and control flow is transferred to the record_set function

```
if(!readLine(&inputBuffer, line, 0)){ // actual read
        lineSplit = split_string(line, ',', 1);
        len = atoi(lineSplit[1]);
        memcpy(value, lineSplit[2], len);
}
```

The `len` variable contains the length of the string contained in the line read from the UART channel. This means that this value is user-controlled. The `value` variable is an array of 32 bytes, so it can be overflown by simply writing more data in `line`.

As already mentioned, the `record_set` function creates or rewrites a record. Our objective is to redirect the code there, as already described. We then want to check that the record that we are going to modify is actually modified. This can be done by trying to login. A sample initialization host program is available when downloading the SDK. This sample looks for valid SE*cube*™ devices and factory initializes them, if they aren't already. It tries to login with the default Admin PIN, which is all 0s. If the PIN results invalid, it deduces that the device has already been initialized, since the next step of initialization is to set the Admin PIN to `test`.

```
l1.L1SetAdminPIN(pin_user); //set admin pin to "test"
```

In order for the SE*cube*™ to communicate with the host program, it must fall into the `device_loop` function, which listens for host commands from the USB. This loop can be reached from our BLE loop with a special password coming from the smartphone, say '`config`'. When entered into the `device_loop`. In order to prove that the ROP chain was successful, we are going to execute the following steps:

1. Initialize the device and set the password to 'test'
2. Run the BLE loop on the device
3. Send the ROP chain from the smartphone application and try to set the password to all 0s
4. Try login with all 0s again

When we execute the first step, on the host console, the following output will appear:

```
Searching for SEcube(s)
Device Found!
User PIN set to "test" successfully
Admin PIN set to "test" successfully
Initialization completed Successfully
```

We then restart the device, that enters in the BLE loop again. The iOS™ has been modified in order to include the payload after the user controlled text field. We simply put 40 characters in order to fill the buffer and overflow another pointer and an integer value (we have put `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcddddeeee`).

After the payload has been sent, we can send the device in `device_loop` function, which waits for commands from the host. We connect with our host system and try again to log in with the all 0s password. The result on the output terminal is the following:

```
Trying privileged login with all zeros
Login ok!

Logout ok!
```

The host program managed to login with a password that should have been changed. This demonstrates that the attack was successful and the password has been changed. Another and more immediate example to try would be to redirect the code directly to the branch where the device allows the user in. This would reflect immediately on the mobile application as a green writing saying that access was granted. Both examples, though, show thoroughly the impact possibilities that a ROP attack would have on IoT devices implementing similar functionalities.

4.4 Patching the Breach: FPGA monitor Setup

4.4.1 Code Instrumentation

In order to protect the code, as specified in 3.4, it is necessary to adopt code instrumentation, since this is a hybrid approach. At the time this work is being written, there is,

however, no available automated code instrumentator. Such a program should prune out all direct branches and compute the origin tree of each indirect branch value in the offline phase. Hence, we do not show binary dimension overhead here and we focused instead on the vulnerable branch, by writing code instrumentation by hand.

First of all it is necessary to identify the type of branch, among the ones listed in 3.5. In our case, the `readChar` function is called only by the `mainLoop` function, so we do not have to worry about multiple return locations for it. The vulnerable branch is a return operation at the end of the `readChar` function, so this is an *insecure backward edge with single target location*. Thus, the operations that have to be followed are:

- Mask out interrupts apart from the one that is important to us
- Mark the current branch location with a label and save it
- Store the label in the memory location `#0x60000000`

In order to accomplish this, the following assembler code instrumentation is put

```
CPSID i
PUSH {r1, r5}
MOV r1, #456
MOV.W r5, #0x60000000
STRH r1, [r5]
POP {r1, r5}
```

This is the instrumentation inserted at the source location. At the target location, we have to perform the same operations, but the `store` operation is performed at a different location.

```
PUSH {r1, r5}
MOV r1, #123
MOV.W r5, #0x60000000
STRH r1, [r5, #2]
POP {r1, r5}
CPSIE i
```

The `strh` instruction passes to the FPGA an address (second operand of the instruction) and data (first operand of the instruction). The address to which the `strh` instruction writes is the opcode for the monitor of the operation that has to be performed on the data. In this case, the operations to be performed are:

1. Store the first label
2. Store the second label
3. Look for the label pair in the Secure Edge Table
4. Raise an exception if there is a miss in the table

It is important to mention that an internal timer in the monitor starts when executing the first `store` and it waits until the next `store`. The time elapsed between these two instruction is accurately computed and if the second `store` does not arrive before the timer expires, an exception is raised. This is to avoid that the return instruction jumps somewhere in the code that does not contain instrumentation.

4.4.2 Synthesis of the Monitor

It is important to synthesize the monitor from VHDL every time that the CFG of the code changes. This is because the Secure Edge Table is synthsized within the monitor beforehand. It is in fact necessary to fill this table before prior to synthesis. Since this process is not automated yet, the table has to be filled manually. Since we have instrumented only one edge, the table will contain only a pair. With the help of a script, it is possible to write manually the label pairs and then converts it into a `.mem` file, which is used in the synthesis phase to initialize the content of a ROM.

4.4.3 Uploading the Instrumented Firmware and Monitor

In the Lattice Diamond™ tool for the synthesis, it is possible to enable the option that converts the bitstream of the design in a C array. This will be useful when we have to upload the design of the monitor inside the FPGA.

In order to have the whole system working, it is necessary to back the firmware of the SE*cube*™ with a bootloader which takes care of the preliminar configuration. The bootloader takes the C arrays generated by the synthesis tool and uses them to send bit by bit to the FPGA in order to program it. After this operation, which takes a pair of minutes to complete, the bootloader toogles the RST pin of the FPGA in order to enter the initial state of the Finite State Machine.

4.4.4 Final Results

When repeating all the steps that we have executed in 4.3.3, after having initialized the FPGA, it is possible to observe that the attack will no longer work. Indeed, the application still works normally when sending a normal payload from the smartphone application. When providing the correct password via BLE characteristic, the smartphone application will show a green message saying that authentication was successful.

We can appreciate the work of the CFI monitor when, instead, we try to send the malicious payload via the BLE application. It is sufficient to repeat the same procedure as before, in 4.3.3 and observe that it is no longer possible to login with the all 0s password, after the initialization has been performed. When repeating steps from (1) to (3), it is not possible anymore to login with all 0s. The message shown on the host terminal will be the following:

```
Searching for SEcube(s)
Device Found!
SEcube already initialized!
```

This means that when trying to send the ROP chain, the monitor interepted the attempt of code redirection and raised an exception which called a handler that reset the device, restarting the loop.

The CFI monitor developed at our department has proven to be a rather easy to implement solution, the requirements being only to use a device which mounts onboard a reconfigurable hardware module. The performance overhead has been measured in [24] by the authors and we report it in the table in Fig. 4.6

Benchmark	**Inputs**	**Time (no prot.)**	**Time (prot.)**	**Overhead**	**# instr. (no prot.)**	**# instr. (prot.)**	**Overhead**
SHA	Message of 100 KB	368449 μs	368457 μs	$< 0.01\%$	20453	21194	3.62%
RIJNDAEL	Message of 100 KB	1083568 μs	1083694 μs	0.01%	25471	26029	2.19%
DIJKSTRA	Matrix of 100x100 **int**	2880724 μs	2894391 μs	0.47%	20166	20912	3.70%
STRING	1331 strings (var. length)	178616 μs	180028 μs	0.79%	20060	20791	3.64%
BITCOUNT	12800 **int**	419545 μs	1233227 μs	193%	20192	20944	3.72%

Figure 4.6. Measurements performed in [24] about CFI monitor overhead

Chapter 5

Conclusions

Nowadays, the question about security in the digital world is becoming more and more relevant. This is due to the fact that the number of interconnected devices is increasing exponentially. Not only the numbers are growing, but the role that these devices are acquiring is getting more entangled with our daily lives. When we talk about computing devices, we do not only refer to personal computers or servers, but we include smartphones, watches, TVs and now also domestic appliances, like fridges, heating systems, doors, etc...

Malfunctioning of such devices not only means financial or confidential data losses, but safety issues arise. If a smart door lock does not work as intended, the house inhabitants' life is put to risk. It is true, though, that many countermeasures have been devised in the last decades against the most popular categories of attacks, along with the spread of good habits and awareness of the risks and impact of the digital world. Still, we are not completely sound. The human factor and the growing complexity of technology are two factors that likely will not disappear with time, being them intrinsic to the digital world. The average user may be educated and informed, but having systems that integrate many services, pieces of software and architectures, only professionals can deeply understand the inner workings of systems, and yet mistakes are made also by those professionals.

What I have discussed about in my thesis work is linked to this topic, but limits to the field of IoT and Embedded Systems. As already mentioned, this is the new frontier of the Digital Revolution and it is of extreme importance to take the security in the Embedded world in consideration. IoT devices, which are a subcategory of Embedded devices, are exposed to two main problems:

- They are always interconnected.
- They have limited computing resources
- Often, they have to comply to real-time constraints

The first characteristic of these devices expands the attack surface available to intruders. An attacker has the chance to connect to such systems via many protocols and from almost anywhere in the world. The second and the last characteristic combined instead pose difficult challenges to programmers and developers of such devices, since applying security by design becomes hard. While in modern operating systems we may find several security

mechanisms, product of years of research and patches, in the embedded world this is more complicated, since many times firmwares run on bare-metal, thus there are not prebuilt and well-tested libraries or security features that prevent most of attacks. Having few mega bytes of ROM imposes codebase to be concise; constraints about performance and power-efficiency put a limit in the number of instructions that a processor can execute for a given task. This automatically creates the necessity to trade between performance and security. The most common type of vulnerabilities of native codebases are memory vulnerabilities. These vulnerabilities are then exploited by attackers to take control over the execution flow or read/write arbitrarily memory. There are two main ways to exploit memory vulnerabilities:

- Code Injection
- Code Reuse

In this work, I have focused mainly on the second type. Code injection has proven to be inefficient and less expressive with respect to code reuse techniques. The latest type of Code Reuse attacks (Return-Oriented Programming, Jump-Oriented Programming, Call-Oriented Programming, etc...), have demonstrated to give extreme expressiveness (reaching easily Turing-completeness) and to be less prone to detection. The aim of this work is to present the State of the Art of a family of techniques that aim at preventing such attacks: they try to guarantee Control Flow Integrity. I have described and analyzed many of these solutions, implemented in software or hardware. The two types of implementations have their advantages and disadvantages: the first are flexible and rather easy to implement, but slow, while the second are fast but require modifications to existing circuitry.

At our department, at Politecnico di Torino, a solution has been proposed, which strives to merge the two approaches, bringing the advantages of both. This solution is theorized for embedded devices that are bundled with a reconfigurable module that communicates with the microcontroller. As statistics suggest, providing devices with an FPGA or other reconfigurable hardware is a growing trend, with obvious advantage in terms of performance and power efficiency. The solution that we describe here, which has been first presented in [24], adopts this kind of devices and implements a CFI monitor onto an FPGA. While the FPGA monitor represents the hardware part of the solution, while code instrumentation constitutes the software part. This is why we consider this as hybrid approach.

In the second half of my thesis, I have developed a simulation of a safety critical application. The scope of this application is to simulate an attack that exploits a memory vulnerability and sends a ROP chain to redirect code execution. After having showed the effects of the attack and the impact it can have on the safety properties of the system, the currently available implementation of the CFI monitor is mounted onto the device, in order to show that the attack cannot be executed anymore.

The application simulates a Smart Access Monitor that checks that a user has the rights to access a confidential area. The monitor communicates with a smartphone, which belongs to the user and contains the secret that authorizes the user to enter the area.

This work does not aim at providing additional performance measurements, due to the fact that instrumenting a large codebase like the one of the firmware we have used is impractical without an automatic instrumentation tool. As the table in Fig. 4.6 reports, the

performance overhead is minimal for simple algorithms and benchmarks. This is justified by three main factors

- Code instrumentation is reduced to minimal, a simple `store` operation has to be performed
- Code instrumentation is inserted at most on all indirect branches and performance can be tuned by trading security
- Edge check is performed by the hardware monitor

The proof that we have given shows that it is fairly simple to mount the monitor onto the FPGA and that modifications to the program requires simple reconfiguration of the design. The design is architecture independent, the only adaptation to be done is the code instrumentation of the firmware.

At the current state, it is not possible to automate many of the passages, but it is just a matter of work and time before that these tools will be developed and the update procedure of the firmware, along with its new CFG, will also be straight forward.

Possible improvements in the direction of this thesis work may consist in implementing the solution on a production software, hence completely instrumenting the codebase. In order to achieve this, an automatic instrumentation tool has to be developed. Another upgrade to improve the updatability of the CFI monitor. Currently, an update of the monitor requires synthesizing the schematics and converting the bitstream into a C array to copy in the source code of the bootloader. This process could be made more simple by inserting the bitstream in FLASH or in ROM. Moreover, a change in the CFG requires resynthesizing the schematics of the CFI monitor; an update of the firmware requires then to reupload the entire bitstream. It would be sufficient to only update the Secure Edge Table.

The ongoing trend of new architectures and microcontrollers implementing an FPGA in their packages is what makes this solution so affordable. The rapidly increasing demand of Embedded Devices in the IoT world, and not only, is what makes solutions like this necessary. Code reuse attacks are usually very difficult to counteract, and staistfying both performance and resources constraint is crucial in the embedded world.

Bibliography

[1] Martin Abadi, Mihai Budiu, and Úlfar Erlingsson. *Control-Flow Integrity*. Tech. rep. MSR-TR-2005-18. ACM Conference on Computer and Communication Security (CCS). 2005, pp. 340–353. URL: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/.

[2] M. Alam, D. B. Roy, S. Bhattacharya, V. Govindan, R. S. Chakraborty, and D. Mukhopadhyay. «SmashClean: A hardware level mitigation to stack smashing attacks in OpenRISC». In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2016, pp. 1–4. DOI: 10.1109/MEMCOD.2016.7797764.

[3] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. «Mitigating Code-Reuse Attacks with Control-Flow Locking». In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC '11. Orlando, Florida, USA: Association for Computing Machinery, 2011, 353–362. ISBN: 9781450306720. DOI: 10.1145/2076732.2076783. URL: https://doi.org/10.1145/2076732.2076783.

[4] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. «Jump-Oriented Programming: A New Class of Code-Reuse Attack». In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: Association for Computing Machinery, 2011, 30–40. ISBN: 9781450305648. DOI: 10.1145/1966913.1966919. URL: https://doi.org/10.1145/1966913.1966919.

[5] C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hely. «A red team blue team approach towards a secure processor design with hardware shadow stack». In: *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*. 2017, pp. 57–62. DOI: 10.1109/IVSW.2017.8031545.

[6] *Chapter 4. GATT, Services and Characteristics*. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html. [Online; accessed 21-March-2021].

[7] A. Chaudhari and J. A. Abraham. «Effective Control Flow Integrity Checks for Intrusion Detection». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 1–6. DOI: 10.1109/IOLTS.2018.8474130.

[8] Linbo Chen, Jianhui Jiang, and Danqing Zhang. «Code Reuse Prevention through Control Flow Lazily Check». In: Nov. 2012, pp. 51–60. ISBN: 978-1-4673-4849-2. DOI: 10.1109/PRDC.2012.17.

[9] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. «DROP: Detecting Return-Oriented Programming Malicious Code». In: *Information Systems Security*. Ed. by Atul Prakash and Indranil Sen Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 163–177. ISBN: 978-3-642-10772-6.

[10] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. «Defeating memory corruption attacks via pointer taintedness detection». In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 2005, pp. 378–387. DOI: 10.1109/DSN.2005.36.

[11] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. «HCFI: Hardware-Enforced Control-Flow Integrity». In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. New Orleans, Louisiana, USA: Association for Computing Machinery, 2016, 38–49. ISBN: 9781450339353. DOI: 10.1145/2857705.2857722. URL: https://doi.org/10.1145/2857705.2857722.

[12] J. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. Si Merabet, and M. Timbert. «CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity». In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, pp. 529–536. DOI: 10.1109/DSD.2018.00093.

[13] L. Davi, M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. «HAFIX: Hardware-Assisted Flow Integrity eXtension». In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744847.

[14] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. «MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones». In: (Jan. 2012).

[15] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. «Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks». In: *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, 49–54. ISBN: 9781605587882. DOI: 10.1145/1655108.1655117. URL: https://doi.org/10.1145/1655108.1655117.

[16] A. De, A. Basu, S. Ghosh, and T. Jaeger. «FIXER: Flow Integrity Extensions for Embedded RISC-V». In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 348–353. DOI: 10.23919/DATE.2019.8714980.

[17] *ESP32 AT firmware - GitHub*. https://github.com/espressif/esp-at. [Online; accessed 21-March-2021].

[18] ZhiJun Huang, Tao Zheng, and Jia Liu. «A Dynamic Detective Method against ROP Attack on ARM Platform». In: *Proceedings of the Second International Workshop on Software Engineering for Embedded Systems*. SEES '12. Zurich, Switzerland: IEEE Press, 2012, 51–57. ISBN: 9781467318532.

[19] Zhijun Huang, Tao Zheng, Yunxiu Shi, and Ang Li. «A dynamic detection method against ROP and JOP». In: *2012 International Conference on Systems and Informatics, ICSAI 2012* (May 2012). DOI: 10.1109/ICSAI.2012.6223219.

[20] Y. Lee and G. Lee. «Detecting Code Reuse Attacks with Branch Prediction». In: *Computer* 51.4 (2018), pp. 40–47. DOI: 10.1109/MC.2018.2141035.

[21] Yang Li, Zibin Dai, and Junwei Li. «A Control Flow Integrity Checking Technique Based on Hardware Support». In: Oct. 2018, pp. 2617–2621. DOI: 10.1109/IAEAC.2018.8577547.

[22] IHS Markit™. *The Internet of Things, a movement, not a market*. https://cdn.ihs.com/www/pdf/IoT_ebook.pdf. [Online; accessed 28-november-2020]. 2017.

[23] N. Maunero, P. Prinetto, and G. Roascio. «CFI: Control Flow Integrity or Control Flow Interruption?» In: *2019 IEEE East-West Design Test Symposium (EWDTS)*. 2019, pp. 1–6. DOI: 10.1109/EWDTS.2019.8884464.

[24] Nicolo Maunero, Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. «A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems». In: Apr. 2020, pp. 1–10. DOI: 10.1109/DTIS48698.2020.9081314.

[25] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu. «Control Flow Integrity Based on Lightweight Encryption Architecture». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.7 (2018), pp. 1358–1369. DOI: 10.1109/TCAD.2017.2748000.

[26] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. «Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions». In: *Journal of Computer Virology and Hacking Techniques* 14 (May 2018), pp. 1–18. DOI: 10.1007/s11416-017-0299-1.

[27] *SEcube Open Source SDK*. https://www.secube.eu/resources/open-sources-sdk/. [Online; accessed 21-March-2021].

[28] Hovav Shacham. «The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)». In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[29] Y. Shi and G. Lee. «Augmenting Branch Predictor to Secure Program Execution». In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 10–19. DOI: 10.1109/DSN.2007.19.

[30] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. «Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity». In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6. DOI: 10.1145/2897937.2898098.

[31] *Top 10 Technologies That Will Drive the Future of Infrastracture and Operations*. https://www.gartner.com/en/documents/3970841/top-10-technologies-that-will-drive-the-future-of-infras. [Online; accessed 15-March-2021]. 2019.

[32] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. «On the Expressiveness of Return-into-libc Attacks». In: vol. 6961. Sept. 2011, pp. 121–141. ISBN: 978-3-642-23643-3. DOI: 10.1007/978-3-642-23644-0_7.

[33] Yubin Xia, Y. Liu, H. Chen, and B. Zang. «CFIMon: Detecting violation of control flow integrity using performance counters». In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* (2012), pp. 1–12.

[34] Craig Young. *Tripwire Research: IoT Smart Lock Vulnerability Spotlights Bigger Issues*. https://www.tripwire.com/state-of-security/featured/tripwire-research-iot-smart-lock-vulnerability/. [Online; accessed 25-february-2021]. 2020.