POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Automatic Binary Analysis and Instrumentation of Embedded Firmware for a Control-Flow Integrity Solution

Author: Valentina Forte

Supervisor: Paolo Ernesto Prinetto

April, 2021

Abstract

The growing number of connected embedded systems has enabled the so-called IoT (Internet of Things), nowadays present in numerous scenarios of our daily life: from mobile phones to televisions, from wearable devices to surveillance systems, from medical devices to transport and industrial control systems. Since IoT devices often exercise control of critical infrastructures, they naturally become the target of cyber-attacks, which undermine to take possession not only of the data exchanged by them, but also of their control functionalities, breaking into possible software vulnerabilities present within the executed code. The goal could be gaining complete access to the device, but also altering its behavior by injecting malicious code or making it unusable.

Applications executed on embedded systems are generally written in C and C++ languages, which provide high performance, but could also introduce *bugs* and *flaws* that can be exploited to corrupt memory, since memory management is entirely entrusted to the programmer. Secure programming rules should be followed when writing C and C++ code, to avoid common problems such as pointer ambiguity, memory leakage and buffer overflow. By exploiting these coding errors, an attacker can override the contents of a memory location, whether they are local variables, data structures or return addresses of a function, to change the instructions execution flow. This is one of the basic principles of an advanced exploit paradigm, called *Code-Reuse Attack* (CRA).

CRA are implemented through attack techniques like *Return-Oriented Programming* (ROP) or *Jump-Oriented Programming (JOP)*, that harness the execution of malicious actions by reorganizing snippets of few machine instructions (called *gadgets*) already present in memory. In the common threat model, the attacker, through memory corruption, is able to force the execution of the code to be *hijacked* towards a chain of these gadgets, which in its entirety produces the execution of a malware.

Control-flow Integrity (CFI) solutions proof that it is possible to mitigate the effects of these attacks by adopting protection mechanisms that safeguard the integrity of the execution flow. Through the computation of the *Control-Flow Graph* (CFG), it is possible to determine the set of valid destinations for all machine code instructions involving a control-flow transfer (such as branches, calls and returns).

The aim of this thesis is to provide an automatic tool capable of extracting the CFG and instrumenting the binary code of the program in such a way that it is *resilient* to memory corruption problem. The tool is responsible of the offline part of a hybrid CFI technique for protecting embedded systems, which involves the presence of a *reconfigurable hardware* in the chip. The technique also provides a careful edge classification, that helps to narrow control-flow transfers needing protection. In this way, the instrumentation overhead in

terms of code memory occupation and execution time is minimized.

A Python script is developed to accomplish this expectation, with the support of `r2pipe` module that handle the interaction with the reverse-engineering framework `Radare2`.

Contents

Chapter 1

Introduction

Technological advances have contributed to the production and dissemination of modern "*smart objects*", completely changing the aspect of the world. The impressive number of connected devices, constituting the so-called *Internet-of-Things* (IoT), grows to accomplish the relentless demand to bring the objects of our everyday experience into the digital world, to optimize the services from which people already benefit. The profits offered in terms of productivity make embedded systems increasingly essential. For this reason, their proliferation is employed in an enormous range of applications, such as phones, wearable devices, surveillance cameras, medical devices and industrial control systems.

To provide significant performance in terms of execution speed, simplicity and low power consumption, many of these applications are executed on low-cost devices, so-called *bare-metal* systems. The essential characteristic is the absence of a middleware abstraction layer, like the operating system, as the instructions are executed directly on the underlying hardware. The criticality of security in these setups arises from the fact that, by default, they do not provide any native defence mechanism, exposing to obvious risks critical functionalities, communications and sensitive data. In most cases, embedded systems face technical challenges that make it even more difficult to implement solutions that are robust, safe and lightweight at the same time against any cyber threat.

Furthermore, most of the daily used devices are connected to the Internet, and this increases their exposure to cyber-attacks. The abundant hardware and software vulnerabilities present, united with weak defensive lines often adopted, make these machines an attractive target. The comprehension of the impact of cyber threats to this category of devices is crucial: injecting malicious code into the firmware or hijacking its control-flow can affect the device's performance, radically alter its behaviour, reduce battery life, steal data, render it inoperable, or even involve it in a botnet.

In [1], researchers dealt with the protection of bare-metal machines, analyzing the basic architecture and how the lack of adequate safety mechanisms can endanger the entire device (Figure 1.1). The achievement of strategies that focus on the security pillars, summarized in the *CIA paradigm* (Confidentiality, Integrity, Availability), must consider hardware limitations, mainly related to:

- limited memory size and reduced computation capabilities;
- lack of support for task isolation;

- inapplicability of most solutions to ARM-based architecture;
- lack of operating system intervention;

Moreover, many hardware security features frequently remain unused or easily bypassed and, in general, are not enough to guarantee complete system security. Even data and code section must be protected against possible memory corruption-based attack to prevent system alteration with dramatic changes.

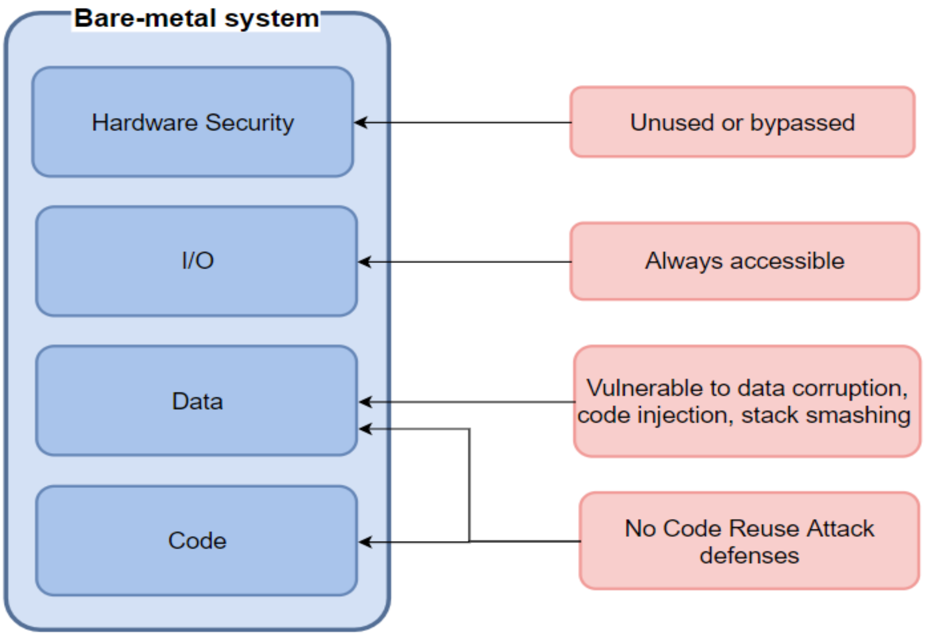

Bare-metal system
Hardware Security
I/O
Data
Code
Unused or bypassed
Always accessible
Vulnerable to data corruption, code injection, stack smashing
No Code Reuse Attack defenses

Figure 1.1: The basic architecture of bare-metal systems [1].

The exploited vulnerabilities have contributed to developing many defence techniques, among which the Control-Flow Integrity (CFI) solutions stand out [2]. The basic idea of such protection mechanisms is to have an online monitor, e.g., a piece of hardware or software, that is able to ensure the integrity of the program flow, and the fact that the application only follows predefined paths, without the possibility of redirecting it through malicious sections of the code. In order to do this, the Control-Flow Graph (CFG) [3] of the program must be extracted from the code before runtime, to correctly instruct the monitor about allowed and not allowed branches. The monitor could be the program itself, e.g., through the insertion of additional instructions to verify branches [4], or a dedicate hardware unit that computes the necessary checks.

This thesis aims to provide an automatic Python tool to extract the CFG and implement protections at binary level for ARM-based architecture. Chapter 2 analyses significant threats present at the software level and mitigations that have been studied. Chapter 3 provides state of the art relevant to Control-Flow Integrity, identifying software and hardware techniques and offering an overview of binary instrumentation tools. Chapter 4 discusses a hybrid CFI-solution for protecting embedded systems, of which the Python tool is a significant part. Chapter 5 explains the organisation and the implementation strategy of the developed tool. Chapter 6 illustrates the tool performances and the obtained results. Finally, Chapter 7 concludes the work and offers suggestions for future improvements.

Chapter 2

Background

The present Chapter aims to analyze the most common embedded software vulnerabilities, entry points to perform *memory corruption* attacks. The provided technical background concepts are essential to examine *control-flow hijacking* attacks and investigated countermeasures.

2.1 Embedded Software Security

Most of the embedded software applications are developed using *memory-unsafe* languages, such as C and C++. These languages offers high performance possibilities, but at the same time, they leave to programmers the responsibility of directly managing the memory. This opens the door to a wide plethora of unintentional errors that can be exploited to corrupt memory: it is statistically impossible to write a completely bug-free program. However, vulnerabilities are too often the result of hasty or wrong programming procedures, that leave critical infrastructures exposed. These issues are often due, for instance, to the adoption of insecure encoding practices, e.g., the usage of a code library that contains vulnerabilities, or handles memory pointers improperly.

The leading cause of misconceptions is a result of inadequate memory management by code writers. With pointer usage, the programmer is granted unrestricted memory access, and assumes the responsibility to ensure the execution of secure operations.

Major issues that can be encountered with this type of approach are:

- **Memory leakage** [5]: it refers to dynamic memory allocation without any release, with consequently heap's saturation.
- **Use-after-free** [6], also known as *dangling pointer*: it concerns handling a pointer that terminates his lifecycle and could store a memory address employed by another program.

The exploitation of these memory corruption vulnerabilities, in best cases, bring to program crashes; otherwise, in worst cases, attackers get to have unbounded memory availability and to jeopardize the entire system, altering his native behaviour.

If the application code is somehow leaked (e.g., for insecure storing and transmission policies, for employ of known and old modules, or for reverse engineering), attackers can scan every single part of the code to identify weak points to attack. They can often find authentication processes that can be defeated or bypassed, allowing them to control the device during a cyber attack. They also seek for entry point to launch *buffer-overflow attacks* [7] and other *denial-of-service attacks* against the device. Analysis of firmware images frequently reveals symbol tables and text strings, which can be used to reverse-engineer functionalities and identify weak points. In some cases, password hashes and security keys can be leaked.

Two separate issues must be treated with solid security. First, security vulnerabilities introduced during the design and development phases must be found and addressed. This range from insecure string management practices, like using `sprintf()` for copying strings instead of more secure variants like `snprintf()`, to hidden backdoors and encryption keys stored in plaintext.

Second, system security features should be included in the initial system design stage. Adoption of strong authentication, secure boot, hardware-based secure key storage, encrypted communication, and secure firmware updates are necessary security items for all devices. Otherwise, embedded software development is forced to an effortless design, because the resources available to the target device limit the ability to use higher-level programming languages, both for used memory and execution time.

Replacing these unsafe programming languages with others more reliable cannot be considered the only strategy. It would mean re-writing many lines of code and training hard the entire audience of embedded software developers. This states the necessity to apply new defence mechanisms to mitigate the effect of these kinds of attacks, such as memory protection, Address Space Layout Randomization (ASLR) [8], stack canaries [9] and Control-Flow Integrity, whose details are treated in Section 2.5.

2.2 Buffer Overflow

Attacks based on buffer overflow vulnerabilities [7] represent a major threat in a security system. Lack of input validation in the developed code with programming languages considered as "*memory unsafe*" can be exploited as a weapon to achieve the highest privileges and subvert program functionality to control the host entirely.

To plan code attack, it is sufficient to provide a sequence of input characters that overcomes boundaries of the destination buffer. The insertion of more data than the buffer can handle *overruns* the content of adjacent memory addresses, corrupting or overwriting the data held in that space [10].

An example of a famous buffer overflow vulnerability is the *HeartBleed bug* [11], found in OpenSSL: here, the insertion of malformed inputs allows to read more data than should be permissible. The presence of this fault compromises the secret keys used, the names and passwords of users and their data.

Typically, the program's control flow can be altered by performing two alternative strategies: *Code-Injection Attack* or *Code-Reuse Attack* (CRA).

2.3 Code Injection Attacks

Memory corruption vulnerabilities are not just exploited to read memory content and overwrite it, working on the target application's weaknesses. Thus, the adversary can leverage a buffer overflow vulnerability to change the stack's return address, function pointer, local variables, and heap content. The lack of data input validation allows the attacker, during the execution, to provide more data than the stack can contain. If the program function does not notify the error, the values are saved the same but overwrite and corrupt the stack (*stack smashing* [12]). Stack smashing is performed by the attacker to introduce and execute malicious *payloads* into a vulnerable computer program.

This type of attack is called *Code Injection* (Figure 2.1) and generally involves two challenges:

1. find a way to inject the malicious payload into the memory;
2. force the *Program Counter* (`PC`) or *Instruction Pointer* (`IP`) to point to the injected code.

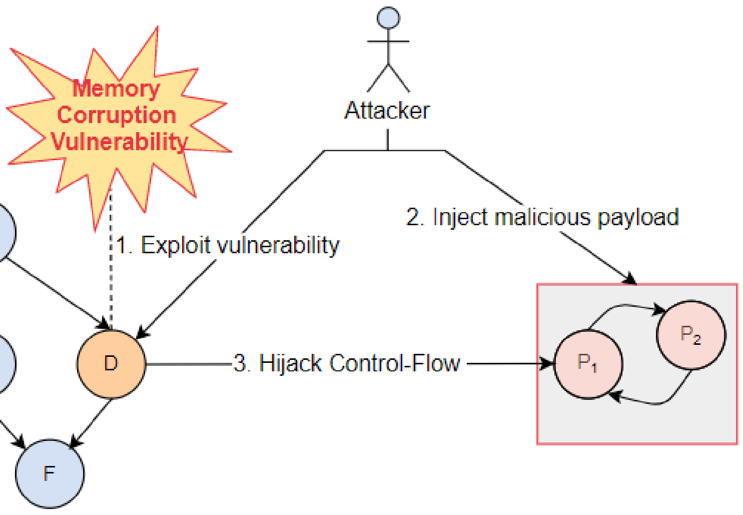


Figure 2.1: Code injection attack.

The first step is to discover an entry point in the application. The main goal is to take control of the computer system to perform any action with a sufficient level of privileges. This ability, known as *Arbitrary Code Execution* (ACE) aims to open up, for example, a command shell out of the program by introducing in memory the machine code corresponding to `system("/bin/sh")`. Once a terminal is available, every command can be launched, i.e., to steal information, disable or modify functionalities.

For the attack to be launched, the program flow must be diverted to the malicious payload, compromising the `PC`'s value. It is known that most common computer architecture keep the return address of a function at the top of the stack, to be restored at `RET` time. Above that, the local variables are stored, and among these, an input buffer could be present, or any other memory space filled with an input content. Therefore, by overrunning a memory buffer onto the stack, the attacker *overwrites* the saved return address in the stack to alter the contents of the saved `PC`, making it to point at the beginning of the payload. In this way, at return time, the execution is redirected to the point where the malware was inserted (Figure 2.1).

2.3.1 Memory Protection

To face the abovementioned threats, a mechanism to protect memory access must be introduced. The most widespread security policy is known as *Data Execution Prevention* (DEP) [13] or *Write XOR Execute* [14]. The basic idea is to mark loaded pages into the heap, stack or in other memory segments as writable (W) or executable (E), but not at the same time, making injected payload "*non-executable*". This policy can be applied with the employment of *Memory Protection Unit* (MPU) [15], that automatically process memory access requests checking, for each segment, the admission rights and, in case of violation, triggers a protection or permission fault.

However, W XOR X policies do not fully solve problems related to memory corruption: such defenses are in fact limited to preventing a code-injection attack. Contol-flow redirection is still possible if the adversary wants to execute code already present in memory, as in the case of a code-reuse attack.

2.4 Code-Reuse Attack (CRA)

After the introduction of memory-protection-based defenses, attackers found an efficient way to *hijack* the program control flow, instead of injecting new instructions in memory. In this way, no attack attempt is detected by common non-executable-page policies. The strategy of inserting external code is preempted by an approach that tends to reorganize the *existing* code to perform arbitrary calculations.

This advanced exploit paradigm is referred to as *Code-Reuse Attack* (CRA), and consists of identifying small fragments of code, called *gadgets*, that end with a *transfer instruction* of the control flow (such as jump, call or return instructions). The memory is not filled with code, but with a long sequence of *addresses*, each pointing to the beginning of a gadget. Mechanism of CRAs is summarized in Figure 2.2.

Also in this case, the starting point is a software-level vulnerability that allows overwriting the return address of the *weak* function so that after its execution, the `PC` content is corrupted to point to the first gadget in the chain to begin the attack. In selecting gadgets, the attacker has access to the code in the program's address space, and more important, in the functions' libraries linked to the application. In fact, these are common to all systems, so they constitute a huge amount of gadget source for the attacker. In other cases, the attacker could even exploit entire functions in such libraries.

This is the concept behind *return-to-libc* [16] exploits, that make use of *buffer overflow* vulnerability to redirect control flow to the C Standard Library's (`libc`) sensitive functions.

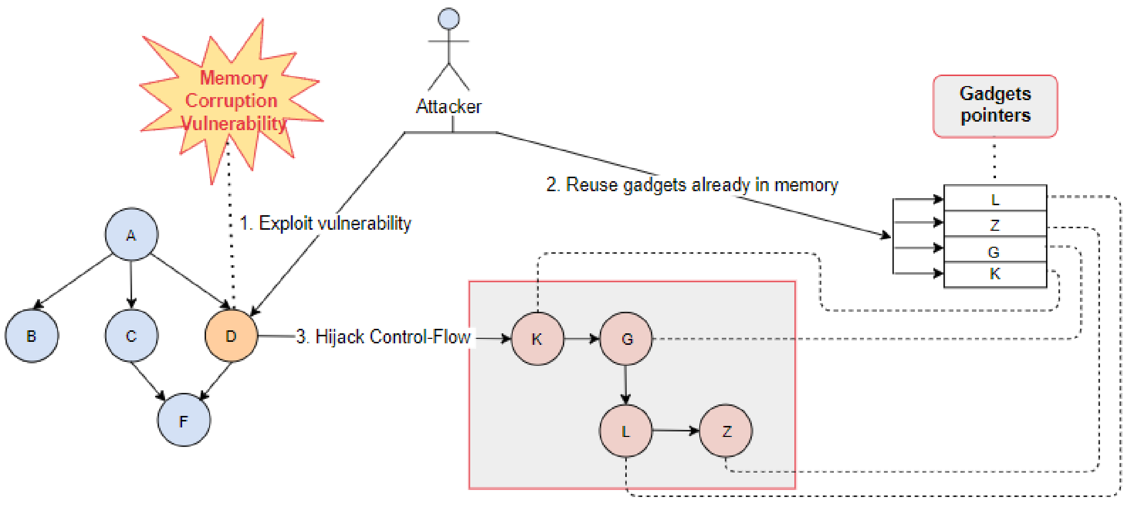


Figure 2.2: Code reuse attack.

In this way, a call function is simulated, having as parameters data inserted into the stack and controlled by the attacker.

The idea about the reorganization of code already presents in the memory segment allows generalizing the variant of this new paradigm of attack that achieves the maximum expression in *Return-Oriented Programming* (ROP) [17] [18]. Subsequently, the attack scheme extends connecting gadgets with specific features. Thus, concepts such as *Jump-Oriented Programming* (JOP) [17], *Call-Oriented Programming* (COP) [19], *Counterfeit-Object-Oriented Programming* (COOP) [20] and others [21] [22] have been introduced.

2.4.1 Return-Oriented Programming (ROP)

The most advanced and traditional version of CRAs is the *Retun-Oriented Programming* (ROP) [23], that aims to overrun the return address of the running program by corrupting the stack content (Figure 2.3).

ROP combines *gadgets* which are often limited to a couple of statements that end with *return* instruction (`RET`), to lead the program to achieve more complex executions. Each piece of code will perform a specific task, e.g., such as read/write operation from/to memory, arithmetic logic procedures between registers.

By reorganizing the gadget's chain, it is possible to completely change the task to accomplish, enabling the attacker, without any privilege, to read or write memory content, stealing information or destroying the system. To launch the attack, the adversary must first inspect the application and carefully select the ROP gadgets to use, storing the "fake" return addresses onto the stack. Then, he overwrites the return address with the first gadget's address to divert the original stream as soon as the function extracts with a `POP` instruction the value from the top of the stack.

The addresses of chosen gadgets must be arranged into the stack so that, when the `RET` instruction is performed, the first gadget's return address is used to corrupt the `PC` content, and so will switch the control to the next gadget in the chain, and so forth for the others in the sequence (Figure 2.4).

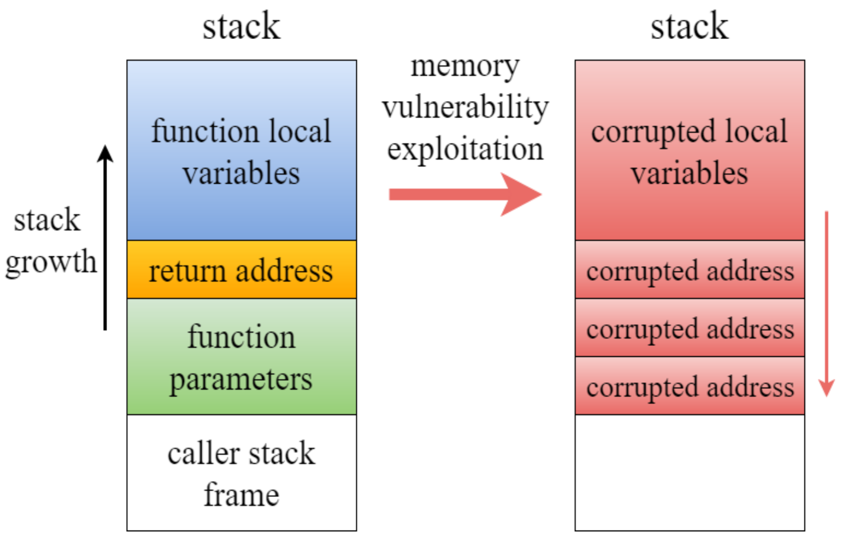


Figure 2.3: Stack corruption with fake return addresses [24].

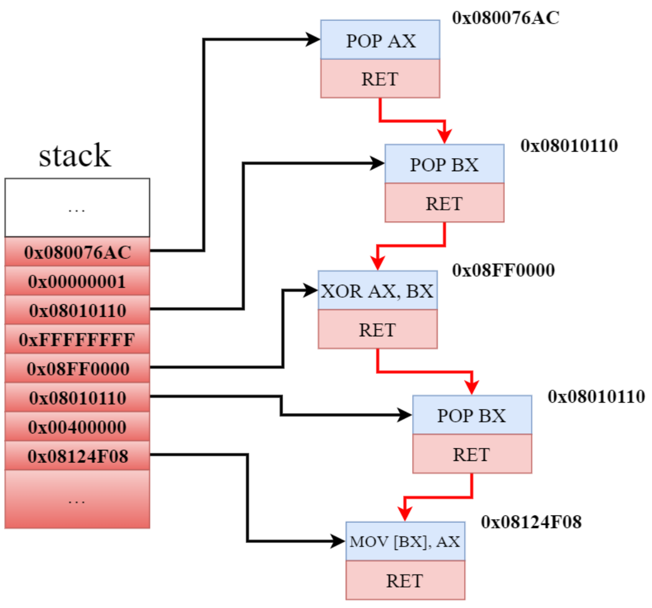


Figure 2.4: Example of Return-Oriented Programming attack [24].

2.4.2 Variants of ROP

As already mentioned, ROP is not the only method to manipulate the control-flow of an existing binary code. *Jump-Oriented Programming* (JOP) [17] is the most popular ROP-derived attack, which activates the execution of a given function through a sequence of *indirect jump* instructions. While in ROP each gadget ends with the `RET` instruction, in JOP each *functional gadget* ends with an unconditional `JMP` instruction. The addresses allowed in the normal execution of a program are collected in a *dispatch table* located in any memory section containing a vulnerability. The final `JMP` or `POP reg` instruction of the gadget is redirected to *dispatcher gadget*, that passes the control to the next gadget in the table to achieve a negative effect. After the jump, the procedure continues so that the functional gadget moves the control back to the dispatcher gadget to establish the next one to call by updating the `PC` [17].

Among other techniques of ROP variants, we have to cite *Call-Oriented Programming* (COP) [19], that involves indirect calls and *Function-Oriented Programming* (FOP) [21], that relies on gadgets built on the existing C functions. More in general, each instruction that involves the `PC` register, processing as a result of any operation, represents the starting point of altering the original program stream.

2.5 Mitigation and Countermeasures

The fight against memory corruption bugs is a great challenge. After having listed the main threats that could alter the firmware's behaviour and allow the attacker to gain control of the system, possible countermeasures are examinated in this Section, highlighting advantages and disadvantages of each in costs and performance.

2.5.1 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a procedure based on the randomization of address space. This involves that the base address of an executable, the location of heap, stack and code in a process address space, are randomly replaced in memory each time the process is created [8]. The hacker might guess or conduct a brute-force attack on critical memory addresses.

Depending on the space size and the entropy level, the difficulty level for the attacker may increase consistently. However, like stack execution prevention checks, this technique does not grant perfect security: if addresses are stored on a low number of bits, ASLR can be bypassed with brute-force attacks, repeating different addresses until success.

2.5.2 Stack Canaries

This method involves generating an arbitrary secret number pushed into the stack under the return address at each function call. Consequently, this ensures defence only for stack memory vulnerabilities. Before each function return, the canary value is checked against the original value. If no alteration is detected, program execution normally resumes, otherwise it ends immediately [9].

Stack canaries seem an efficient mechanism to mitigate any stack smashing. However, two techniques can demonstrate how to bypass them:

- **Stack Canary leaking**: if an attacker can read the content of the canary stack, this value can be used while filling the stack with corrupted values, so no alarm is generated;
- **Brute-force attack**: the canary value is determined when the program starts for the first time. If the program forks, it maintains identical values in child processes, and the attacker uses this to replicate the brute-force attack on all child processes.

2.5.3 Control-Flow Integrity (CFI)

The enforcement of *Control-Flow Integrity* (CFI) security feature has been proposed by Abadi *et al.* [2], and it is the most powerful solution against *control-flow hijacking* attacks, but also very expensive and difficult to implement.

First, the program flow is extracted through the generation of *Control Flow Graph* (CFG) [3], then different strategies are provided to encourage the program to follow it or eventually detect any violation. The CFG obtains the representation of all possible paths that could be covered by the code, in which two essential elements are distinguished: *vertices* also called *basic blocks* that are sets of statements, and *edges* that links two or more basic blocks by control-flow transfers instruction, such as jumps, calls, returns.

Once defined how execution crosses over the code, software or hardware techniques will be used to verify the legal flow by comparing the final target address of control-flow transfer instruction with the licit ones present in the CFG.

After CFI's digression offered in Chapter 3, the discussion will focus on a hybrid CFI solution for preserving embedded systems, with the implementation of *binary instrumentation* process and the monitoring through a *reconfigurable hardware*.

Chapter 3

Control-Flow Integrity: State of the Art

Procedures adopted to safeguard CFI enables to verify that program execution complies with the CFG generated before runtime. In the present Chapter, some known CFI implementations, both hardware-based and software-based, are introduced. Moreover, the Chapter gives a brief introduction to the principal binary instrumentation tools.

3.1 Software-based Solutions

The effectiveness of software-based CFI relies on two components:

1. the extraction of the information about the execution stream through the CFG generation;
2. the insertion of appropriate instructions at control-flow branches to verify the compliance with the CFG.

The objective is to compare the program's actual behaviour to a precomputed model to detect any possible deviation. The purely-software CFI methods ensure the correct traversing of possible contemplates paths extracted from CFG with code or binary instrumentation enforcement. With the insertion of reliable information into CFG, it is possible to avoid any corruption of the victim program's execution flow.

The analysis and instrumentation of binary code can follow two alternatives: *static* [25] or *dynamic binary instrumentation* (DBI) [26] [27], whose dissimilarity resides in process realisation. The static instrumentation (SBI) (Figure 3.1) modifies the binary project permanently before its execution, applying alteration directly on the file stored in memory, while in the dynamic process, the program is monitored during the execution step without the insertion of any instruction into executable, by analysing one instruction at the time via an iterative algorithm and adding the instrumented instruction on-the-fly, as shown in Figure 3.2.

The main advantage of DBI is that it avoids the code relocation problem. Moreover, the use of a more straightforward approach prevents errors related to disassembly generation or binary rewriting. However, runtime analysis requires more execution time, and it is more resource-consuming than the SBI [28]. In both cases, four steps are distinguished [29]:

1. **Parsing**: the key objective is to make the content readable and extract information about the data and code section by generating disassembly code in the static method or monitoring the stream one instruction at a time in the dynamic one. This step also collects details about data structures and code reorganisation, emphasising symbols, labels, and variables;
2. **Analysis**: once retrieved the program structure, this action enables to locate instruction, variables, function and data that can be employed to reconstruct CFG;
3. **Transformation**: the core task is to find vulnerable points, i.e., control-flow transfer instruction, and it proceeds with the relative binary instrumentation;
4. **Code generation**: it generates the final protected executable file.

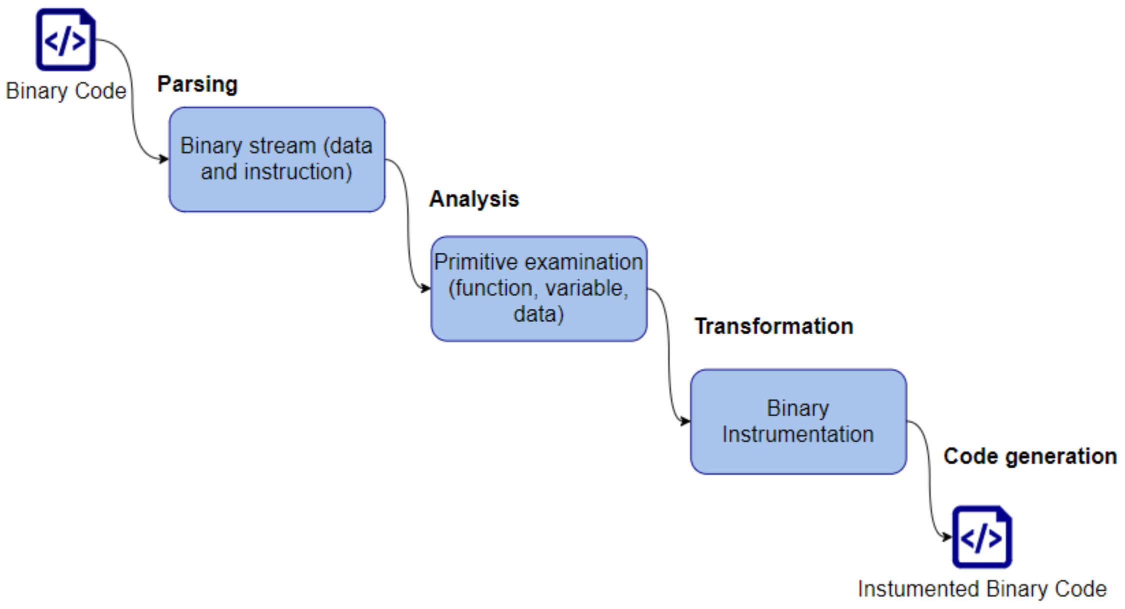


Figure 3.1: Static binary instrumentation approach.

Code instrumentation enriches CFG with necessary metadata to protect critical instruction exploited by attackers to perform ROP or JOP attacks. The basic scheme [4] identifies each indirect call with a pair of *unique IDs* to identify the branch starting location (the *source*) and the destination of it (*target*). The correctness of execution depends on the runtime comparison and the verification of these IDs, essential to ensure the integrity. If there is a matching between the ID stored before the source branch and the ID inserted in correspondence to the instruction of jump target destination, it means the followed stream

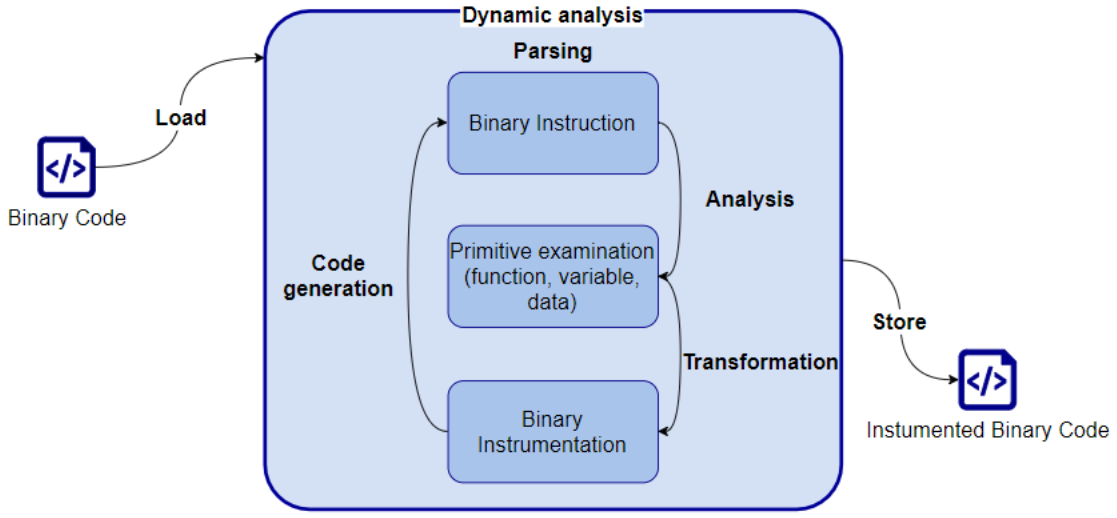


Figure 3.2: Dynamic binary instrumentation approach.

is legal and compliant with what is described by CFG; otherwise, it has been tampered with by a potential attack.

The foremost disadvantage is that the CFI software approach often appends a large amount of code and data structure that penalise execution time and memory occupation, representing a problem for systems with restricted resources. Control-Flow Locking (CFL) [30] is an alternative to the classical implementation of binary instrumentation to reduce defects due to performance overhead. It protects indirect `JMP`, `RET` or `CALL` instructions by accessing in mutual exclusion, without managing synchronisation primitives. The arbitrary *lock* ensures exclusive access to that section until the correspondent *unlock* value releases it.

The consecutive access to an occupied section triggers a system violation and suspends the execution. The final destination contains the corresponding value for *unlock*, that allows identifying direct or indirect returns from a function call and, before passing the control to the next instruction, must reinitialise starting value in such a way that it is feasible to "lock" the critical section without notifying false mistakes.

Others propose solutions like CFIMon [31], relying on Intel architecture features, and MoCFI [32], though for ARM architectures instead, which require the operating system's mediation. For this reason, it does not apply to bare-metal systems. The code instrumentation is reached through an external process that interacts with memory and other registers.

3.1.1 Binary Instrumentation Tools

Based on advantages and disadvantages of instrumentation techniques, tools have been developed to analyze the binary, detect changes in the flow and identify memory access, following the static or dynamic approach.

Among the most popular, it is undoubtedly Pin [33], a tool by Intel that monitors the user application's behaviour while it is running. The binary instrumentation framework is available for Windows and Linux, but only on Intel platforms, making it unsuitable for ARM-based embedded systems. Pin provides APIs, called *Pintools* written in C/C++, to create its analysis tool. The provided APIs identify two types of events:

- *instrumentation routines*, that define where to insert the instrumentation;
- *analysis routines*, that describe the action to be taken when activating it.

There are two implementation modes:

- *JIT*: it creates a copy of the application and edits *Just in time* (JIT), thus never executing the original code;
- *Probe*: it directly modifies the original by inserting instrumentation instructions, granting better performance but being more limited.

Pin allows us to run the program, intercept control transfer points, add instructions to the code via APIs, put the instrumented track in the code cache and then run it.

An example of a static instrumentation toolkit is PEBIL [25], which operates on x86 platforms. The main objective is to create an efficient tool capable of optimizing the instrumentation insertion times. The process is based on the addition of a jump instruction at each critical point, which transfers control to the instrumented code. The basic principle is to save the program status, perform the function to which jump, restore the status and return the control to the caller.

Dynist [34] represents a middle way between the two possible approaches, since it can instrument the code before executing it (*static instrumentation*) or during the execution (*dynamic instrumentation*). This strategy is the most efficient of those proposed, as it gathers the benefits of both methods. It provides APIs for binary analysis, rewriting and runtime patching. Investigation of CFG, functions and control-flow transfer instructions is done by static analysis, to reduce execution costs. If it needs to detect runtime cases due to system events, such as creating a thread, or if the CFG is incomplete (e.g., due to indirect jumps, calls), dynamic analysis is used to obtain greater precision.

For each basic block, edge, function and loop, it identifies the locations where to put the security code, called *instrumentation points*, and instrumentation instructions, called *code snippets*, that add information in the CFG. The dynamic phase identifies the accesses in memory, decoding the operations and addresses that regulate the program flow. In this step, it can solve the indirect jump instructions and protect the *instrumentation points*.

3.2 Hardware-based Solutions

Hardware-assisted CFI monitor overcomes performance losses introduced by routines at the software level. Inspections about the validity of the traversed path face a technical challenge to limit cost and ensure at the same time security principles.

The processor architecture must be revised to sustain CFI dedicated features necessary to handle direct access to registers or monitor data flow on the bus, raising the cost for the hardware component's addition. In contrast, the solution becomes much faster with the running checks done in parallel, in a way that the novel module can interact directly with the processor or pipeline intermediate stage.

3.2.1 Branch Target Encryption

The authors in [35] and [36] provides an adequate defence against ROP e JOP attacks with a technique based on the *encryption* of the return address of called function and the instruction of indirect jump destination, involving a key which is inaccessible to the adversary. The support of an additional hardware component, inevitable for a fast and transparent encryption procedure, does not imply the modification of *Instruction Set Architecture* (ISA). Indeed, it maintains high performance, applying a satisfactory compromise between security and costs.

The operating principle consists of encrypting the return address of a function before executing it and so before pushing it onto the stack. When the `RET` instruction is performed, the address is extracted from the top of the stack, it is decrypted, and then the processor jumps to the obtained address. If the destination has been tampered with, the decryption output results in an illegal address, leading to exception and arresting the system.

In the case of indirect jumps, instead, a *secret key* will be used to encrypt critical instructions that could be compromised to alter the program flow. During the load phase, the jump destination instructions are encoded, and when they have to be executed at runtime, they are decoded by using the corresponding secret key. Moreover, in that event, if the result hijacks the original flow, the error is notified with the consequent execution stops.

The module responsible for the encode protection is joined by the help of the *Physical Unclonable Function* (PUF) [37] or the *Advanced Encryption Standard* (AES) [38] algorithm. Proposals based on random key generation with PUF are cost-effective, because introduce fewer delay, but suffer from a cryptographic point of view, because the encode/decode procedure is implemented through a single XOR operation. Moreover, they are more exposed to *memory leakage*, so the adversary could trace back the key with a careful analysis of plaintext and ciphertext. To partially solve the problem, it could be to useful update periodically the involved key, making difficult the tracking key operation [36].

The version that uses the AES algorithm increases the delay in the achievement of the process. However, it is undoubtedly the more secure, as it does not allow the attacker to decrypt instruction if he does not possess the key.

3.2.2 Shadow Call Stack (SCS)

Shadow Call Stack [39] (SCS) enforcement guarantees system protection only against attacks that overwrite the return address stored onto the stack after call function. The idea is to create an additional stack, called *shadow stack*, that is invisible to the programmer in the user process, and is included in processor architecture, to create secure copies of the return addresses. The double-stack structure is in fact synchronized, so that at every function call, the return address is pushed onto the execution stack and then the exact value is copied onto the secondary stack.

Unlike what happens in the user process's stack, the shadow stack only stores the corresponding return addresses' value, not to impact memory occupation. At return time, the execution integrity is verified by comparing the two return addresses through a `POP` operation from the top of both stacks to detect a possible violation. If they are inconsistent, an exception is triggered, otherwise the execution proceeds normally [40] [41] [42].

The hardware support needed to realize a shadow stack may be intrusive, even if not requiring any instruction set modification. The secondary stack must be in fact placed somewhere away from the user program memory, otherwise the defense is pointless. Having it directly inside the processor is the best option, but also the most expensive, as it requires silicon modifications.

3.2.3 Basic Block Signature Verification

Another way to monitor CFI with low-power consumption has been demonstrated in [43] by enforcing CFI with primary block signature verification. Here, CFG verification involves the *hash algorithm* application, through a dedicate unit closely linked to the processor [44], or interfaced directly with the processor pipeline [45]. The term *basic block* defines a set of sequential statements that do not include any `JMP` instruction, except for the last instruction that jumps to the adjacent block in the CFG.

Once encoded, each basic block is inserted as a new entry in a memory segment inaccessible to the user process, e.g., in a *signature table* [43] [46], mapped in the address space of the target application. During the runtime step, the *signature* of the current basic block is produced and compared with the stored one and also the next block in the sequence must be checked to detect a mismatch for possible control flow alteration or to complete the process.

The realization of the CFI-cache module [43] [47] bounded to both the memory bus and the processor allows monitoring the execution stream and keeping track of basic block information. A supplementary hardware unit follows the instruction flow and compares the current target address with the pre-computed one and, if the signature violation is recognized, the system raises the interrupt. Integrity checks are done in parallel with the program execution, to optimize performance.

3.2.4 Instruction Set Architecture (ISA) Modification

According to CFG information, the extension of the original *Instruction Set Architecture* (ISA) could be a valid alternative to enforce CFI policies. The additional instructions handle the interaction with data structures, registers and labels involved in insecure control-flow transfer statements. The objective is to force, in a controlled way, the instruction flow to explore the original path determined by CFG extraction.

In [48], the ISA modification allows protecting return from called function, leaving exposed to vulnerability the control-flow transfer statements. The completeness is reached in [49], that provides security features also to calls and indirect jumps, with the assistance of `PUSH` and `POP` instruction, and those for comparing current label with the expected one. The main drawback is related to runtime overhead and costs introduced by additional instructions and used data structure stored in memory.

Chapter 4

The Investigated Solution: FPGA-based Control-Flow Integrity

The present Chapter describes the approach proposed in *"A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems"* [24], a paper of the last year authored by Prof. Paolo Prinetto and his PhD team here in Politecnico di Torino. The study provides details about a hybrid CFI technique that combines *static binary instrumentation* with the realization of CFI monitor implemented using a piece of *reconfigurable hardware* connected to the processor. This research outlines the starting point for the python tool development, discussed in Chapter 5.

4.1 Basic Definitions and Edge Classification

Before going into the details, it is essential to provide basic concepts in the context of *Control-flow Integrity.* As already anticipated in the previous Chapters, the validity check of program flow are based on the reconstruction of its structure with the use of two principal elements:

- ***Basic block*** (BB): it defines a set of instructions performed sequentially, as they are free of flow-transfer instruction, except for the last one that passes the control to a subsequent basic block.
- ***Edge***: it identifies a link between two basic blocks, expressed by a control-flow transfer instruction, through which the execution moves from a starting basic block, called *source*, to the destination one, called *target.*

Combining these two obtains the maximum expression in the *Control-flow Graph*, representing the source code reorganization in blocks and edges. The instruction ending the BB allows distinguishing the edge in:

- ***Forward edge***: it binds source BB to the next one inside the same function or to the first BB of another function;

- ***Backward edge***: it refers typically to function returns; it transfers the control from the last BB of a function to the next BB that was in sequence after the one ended with a `CALL` instruction.

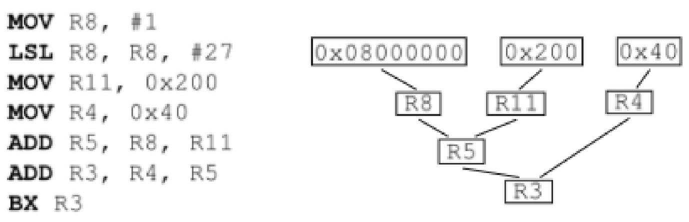
Moreover, branch statements express target destinations with an encoded label, as in the *direct edge*, or with a register content, as in the *indirect edge*.

During an attack, the goal of the attacker is to subvert the program flow, altering the code pointer stored in memory or registers. This is possible by exploiting the target addresses of indirect jumps. In ARM architectures, indirect jumps are identified as the category of instructions that modifies the *Program Counter* (PC) content with a non-constant operand, using a register to indicate the destination, or a value retrieved from memory sections that are at risk of corruption (Table 4.1).

CFG extraction shows all possible paths starting from the examined binary's entry-point. However, it is possible to immediately locate flow deviation only related to *direct* or unconditional branches. In presence of *indirect* edges, finding the destination address is a non-trivial process. It is necessary to examine the operand involved in the edge instruction, tracing back its history until its origin, and identify its final target.

The reconstruction of this so-called *origin tree* (Figure 4.1) consists of following back all the instruction that contribute to form the branch operand, and that access to memory or register values, which corruption alters the execution flow to unwanted address. In embedded system executable, where no external library is linked at runtime, and the code is all statically present in memory, the process of finding a finite number of possible destination for each branch is always feasible [24], but the complexity changes with the increasing number of instruction. The ultimate result will be a tree with a register involved in branch instruction as *root*, and all memory accesses location, constant value or value derived from external input as *leaves*. This event helps in building the register content, and insert pieces of information to CFG to assign at each edge all reachable destinations, protecting its integrity.

The initial assumption states that, in bare-metal systems, the Flash memory stores the binary file permanently, excluding the opportunity of any modification during the running phase. So, the code memory can be assumed as incorruptible.



```
MOV R8, #1
LSL R8, R8, #27
MOV R11, 0x200
MOV R4, 0x40
ADD R5, R8, R11
ADD R3, R4, R5
BX R3
```

Figure 4.1: ARM code segment and origin tree of the register involved in the indirect jump [24].

A further division between *secure* and *insecure edge* allows overcoming design restriction, by applying binary instrumentation only to branches that are at actual risk of hijacking. An edge is *insecure* if its branch operand, i.e., the final address value, is the

Assembler syntax	Instruction	Description
BL label	Branch with link	Branch to label and store the next instruction address into Link Register (LR). The instruction belongs to the direct jumps category, but LR could be altered inside the subroutine, causing an unpredictable branch
BLX Rm	Branch with link and exchange	Branch to address stored in Rm, set the address of LR to the next instruction and change the instruction set to ARM or Thumb state
BX Rm	Branch and exchange	Branch by copying the Rm address into the program counter PC and exchange, if necessary, the instruction set to ARM or Thumb mode, determined by the last significant bit (LSB) of Rm
ADD PC, PC, Rm	Add without carry	Modify PC address adding it the Rm content and the execution branches to the resulting address
MOV PC, Rm	Move content from source (Rm) to destination (PC)	Update PC address with the copy of the Rm content and the execution branches to the new address
TBB [Rn, Rm]	Table branch byte	Branch that use a table of the single-byte offsets to modify PC address. Rn identifies the pointer to the table and Rm the index into the table
TBH [Rn, Rm]	Table branch halfword	The same of TBB instruction but use a table of halfword offsets
POP PC LDMIA sp!, reglist	Pop registers from stack	Pop the address from the stack into PC, bringing to jump. This is usually used to return from a subroutine, where LR content is stored before executing the called function and restored with a copy into PC before return with one of these instructions

Table 4.1: Critical jump instruction in ARM language [50].

combination of data coming from the vulnerable memory section, whose contents could be modified through memory vulnerabilities. This implies that all the direct edges are *secure*, since the address is determined *a priori*. Also, every indirect edge that depends on a value "never leaving the code" (i.e., always stored in Flash memory or in processor registers, with no transits in data memory) is secure as well.

The reconstruction of a register's history as an argument of the branch instruction is performed by *static* code analysis. When the program runs, the processor could receive an interrupt request, triggering the execution of an *Interrupt Service Routine* (ISR). This event can happen everywhere in the code, and at any time, with no possibility of forecasting it. Since it is impossible to know in advance if and when these routines are executed, it is impossible to automatically include this information in the CFG.

Every time the processor must execute an ISR, it stores in memory the *registers* and the *current program context*, to be restored as soon as the routine ends. Moreover, ISRs could contain a vulnerable buffer used to corrupt memory and launch an attack. As a consequence, ISRs constitute a critical aspect to take into account [51]. First, the defence mechanism must preserve program state and registers, saving the content at ISR entry point. Second, the offline analyzer must scan the instructions of the ISR and protect the critical jumps as in any other function. Finally, before returning the control to the static code, the execution context must be perfectly restored. The inclusion of specific instrumentation code before and after the ISR execution ensures that the outgoing program state has remained consistent with the incoming one.

4.2 Protection Mechanism

Hardware techniques presented so far built their defensive lines directly on the underlying architecture, with the request of changing the processor design and involving a significant increase in production costs and system complexity. Instead, the presented mechanism ensures the same security results, integrating the CFI monitor on *reprogrammable hardware* (FPGA) closely connected to the processor. In other words, during the execution, the FPGA act as a *CFI monitor* that inspects the firmware flow through specific additions included in the firmware, and involving the CPU-FPGA interface (Figure 4.2).

The first step is to be done ahead of the execution, and it is about creating an instrumented binary version, which secures critical points in the code that, once compromised, may alter the original flow. The instrumentation process includes single `STORE` instructions, aiming to communicate to the FPGA the current position inside the code. Given an insecure edge, these instructions are inserted before the branch and before the first instruction of the destination site. All source-destination pairs are extracted from the CFG, so they are part of the information that the FPGA has to know in advance. For this, pairs are encoded into the programming *bitstream* of the FPGA, that is loaded in the device at the same time of the CPU Flash programming. In other words, at programming time,

- the CPU is programmed with an instrumented version of the binary firmware, containing additional `STORE` information, needed to transmit information to the FPGA at runtime;
- the FPGA is programmed with an architecture able to accept input from the CPU and verify the compliance of such data with the CFG, whose information are embedded inside the architecture itself.

At runtime, after the reset of the system, the CPU runs and without the need of any further intervention, `STORE` instructions communicate sensitive position to the FPGA monitor. The monitor detects violations in two main ways:

- when a non-valid destination label is received after a valid one, meaning that the destination is not to be paired with the source;
- when no label is received after the reception of the source label; this means the flow has been redirected to a non-instrumented site, which by definition is non-valid.

Whenever a violation is sensed, the FPGA uses a hard fault signal to stop the CPU activity, or to perform a corrective action depending on preferences.

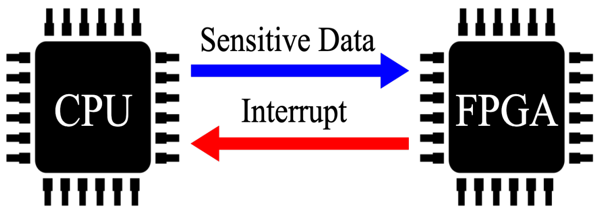


Figure 4.2: The CPU-FPGA connection [24].

The implemented strategy includes the *offline* and the *online* phase. In the former, static analysis of the code is performed, with relative binary file instrumentation to defend its vulnerable edges. The fundamental premise is that *it is always possible to establish the target of direct or indirect edge and preserve the program context during the ISR management.* To sustain this assumption, it occurs to find all of the possibly vulnerable location and determine the destination associated with them. For each of these, the CPU sends to FPGA a *unique identifier* (ID) of the BB, together with an *operational code* that identifies the nature of the value and the action to be performed on the FPGA. Operational codes are formed depending on the possible cases that may happen, i.e., on the basis of the *edge classification* presented by the solution described in [24] (Table 4.2). The inserted `STORE` instruction towards the FPGA has the aim of communicating two types of data:

1. a unique ID, equal to the the secure hash of the code location (or, if you want, of the current basic block, ended with the control-flow transfer);
2. a register content, to be sent to FPGA at entry and exit points of ISRs (for program context protection).

The found typologies are:

1. *Forward insecure edge with a single target*: the CPU emits the source BB ID before branching and then sends the target BB ID at arrival;
2. *Backward insecure edge with a single target*: same as the previous;
3. *Forward insecure edge with multiple targets*: as in 1., but applying instrumentation to all reachable destinations;
4. *Backward insecure edge with multiple targets*: as in 2., but applying instrumentation to all reachable destinations;
5. *Forward secure edge to a routine ending with a backward insecure edge with multiple targets*: the transfer action does not need protection, because the edge is secure, but it is essential to ensure that the return address is not modified. This implies that the CPU transmits to FPGA the BB's ID to which the routine must come back;

6. *Forward insecure edge to a routine ending with a backward insecure edge with single target*: CPU sends the source BB's ID before performing the transfer instruction, and the target BB ID when the previous action is completed, in order to verify the calling point identity and the legality of the destination;

7. *Forward insecure edge to a routine ending with a backward insecure edge with multiple targets*: same as the previous, but instrumenting all possible return locations;

Instead, to preserve the program context, the code sites in which insert protection are:

- The *entry point* of ISR: before executing interrupt routines, it is essential to store the register pushed automatically by an ARM processor (i.e., `R0`, `R1`, `R2`, `R3`, `R12`, `LR`, `PC`, `xPSR`) in a dedicated stack into FPGA, by inserting as many `STORE` instructions as there are registers. In addition, it follows the same procedure to save in the FPGA the other registers pushed by the program (other than those pushed automatically);

- The *exit point* of ISR: after saving registers pushed by programs, before leaving the ISR and switching the control to static code, it enters in reverse order as many `STORE` instructions before the call.

As you may notice, no `LOAD` operation is performed from the FPGA, but just `STORE`s. This is done to ensure maximum of security. The monitor inside the FPGA performs the checks about the context internally, and interrupts the FPGA if a mismatch is found.

The code entered in the instrumentation procedure for both the edges and program context protection is shown in Figures 4.6 and 4.5.

After the instrumentation, the elements obtained will be (i) the instrumented firmware and (ii) the table containing all the source-destination associations for each insecure edge. In the last offline stage, a *secure bootloader* is responsible for loading the rewritten binary version, storing into FPGA the jump table converted in a bitstream. The bootloader also correctly sets the CPU-FPGA connection, depending on the interface, in such a way to grant correct communication at runtime.

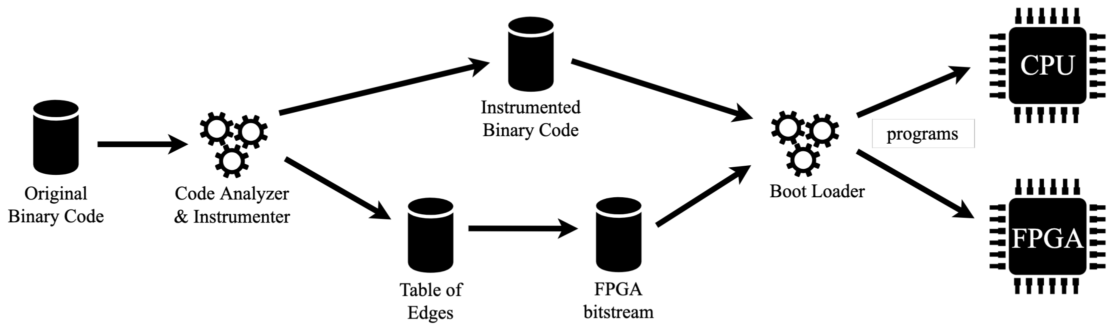


Figure 4.3: The workflow of protection mechanism [24].

4.3 The FPGA-based CFI Monitor

In general, the protection mechanism inserts a `STORE` instruction before *forward* or *backward* insecure edges, and, once reached the final target, communicates again the position. Internally, the FPGA design adopts three distinct data structures to handle different types of transfer instructions (Figure 4.4):

- a ***Secure Edge Table*** to store all of the correlation between the starting label and the arrival one;
- a ***Secure ID Stack*** used as stack for *static function calls*, to confirm the return to the exact caller point;
- a ***Secure Register Stack*** to manage registers content before and after the ISRs execution and defend the program state.

A central Control Unit processes the received sensitive information and distinguishes it based on the *opcode* received. The procedure allows examining the collected data with the expected one through a set of hardware comparators. In the presented technique, the monitor combines the source and destination ID with a XOR operation, and uses the obtained result to access the *Secure Edge Table*. To this location, it founds the details about the specific edge, which uniquely identify it.

In the implementation presented in [24], the communication with the CPU allows the transfer of:

- a 16-bit data passed on the bus;
- a 6-bit address that defines the *opcode*.

Table 4.2 shows the meaning of such opcodes.
Following 4.2, data received by FGPA can be of two types:

1. a unique ID that identifies the hash of a position in the code (13 bit);
2. a value representing half of the content of a 32-bit register (16 bit).

In the first case, the 3 top-most bits are left to 0. When received both source and target, the monitor combines them through simple XOR, resulting in a new 13-bit value. This value is used to address the internal *Secure Edge Table*, that is 8192-word large, with a 16-bit word for each entry. Entries are encoded as follows:

- the first 3 bits define the existence and identity of the edge within the CFG (Table 4.3);
- the following 13 bits report the source ID, to univocally get back to any information about the branch (in fact, this value put in bitwise XOR with the combination, gives back the target ID).

Edge Typology	Opcode	Related Action
Backward insecure edge with single target, Forward insecure edge with a single target, Forward insecure edge with multiple targets, Forward insecure edge to a routine ending with a backward insecure edge with single target	0x60000000	The received value is stored pending the arrival of the next one. In the meantime, the timer is triggered.
Backward insecure edge with single target, Backward insecure edge with multiple targets, Forward insecure edge with multiple targets, Forward insecure edge with single target, Forward insecure edge to a routine ending with a backward insecure edge with single targets, Forward insecure edge to a routine ending with a backward insecure edge with multiple targets	0x60000002	The value is combined with that previously received at 0x60000000, 0x60000006 or 0x60000008. The result allows access to the table containing all the edge information. When FPGA has to check the caller, the data's identity is verified with the value on top of the Secure ID Stack. If the timer expires before something is received, or the combination is not recognized, or the top stack does not match, an exception is triggered.
Forward secure edge to a routine ending with a backward insecure with multiple targets	0x60000004	The value is pushed into the Secure ID Stack.
Backward insecure edge with multiple targets	0x60000006	The value is stored pending the arrival of the next one. FPGA checks the caller identity. In the meantime, the timer is activated.
Forward insecure edge to a routine ending with a backward insecure edge with multiple targets	0x60000008	The value is stored pending the arrival of the next one. In the meantime, the timer is activated.
Entry point of ISR	0x6000000A	The value is relative to ISR management and so it is pushed into the Secure Register Stack.
Exit point of ISR	0x6000000C	The data validity is compared and checked with the value on the top of the Secure Register Stack. If they mismatch, an exception is triggered.

Table 4.2: Opcode received by FPGA and related action to perform

Header	Meaning
000	No edge
100	Valid edge

Table 4.3: Header code used to define the edge existence.

Therefore, if a valid entry is found, the CPU execution is not interrupted and proceeds normally. Otherwise, the monitor sends an interrupt signal to the CPU to stop the process. Moreover, if the executed routine can be called at multiple points in the code and then there are more insecure return points, the FPGA also saves the unique ID of the function's endpoint, which corresponds to the instruction immediately following that of the jump that determines the call. Hence, when it has to go back after performing the function, the monitor not only examines to return to one of the established return points, but checks to return to the exact point that made the last call.

The jumps attestation is accompanied by a *timer* that limits the IDs' waiting time. As soon as it receives the source ID, it activates the timer. This is set exactly to the time taken to perform the edge for that CPU architecture. If the target ID is not sent within the time, it means that the attacker managed to divert the flow to another code block. In this case, the violation is notified with the activation of an interrupt. In order to have all of this working, the program execution must not be suspended by any other hardware interrupt between the sending of the source ID and the sending of the destination ID, i.e., during the branch time. For this reason, the instrumentation code includes an *interrupt disable* instruction, placed before performing these `STORE` operations in order to obtain exclusive control to send the ID, jump to the destination and transmit the final ID. At destination, the instrumentation code will end up with a specular instruction re-enabling interrupts.

Another security feature considered in FPGA design is the inability to access this private resource in any way in the application. To accomplish this constraint, any reference to the FPGA, such as addresses and control signals, is removed by the instrumentation tool if found. The only authorised access is the one introduced by the `STORE` instructions added by the tool. The abovementioned secure bootloader is the only actor able to configure the FPGA before the integrity checks. Before ending its activity, the bootloader should program the MPU to make the access to itself impossible.

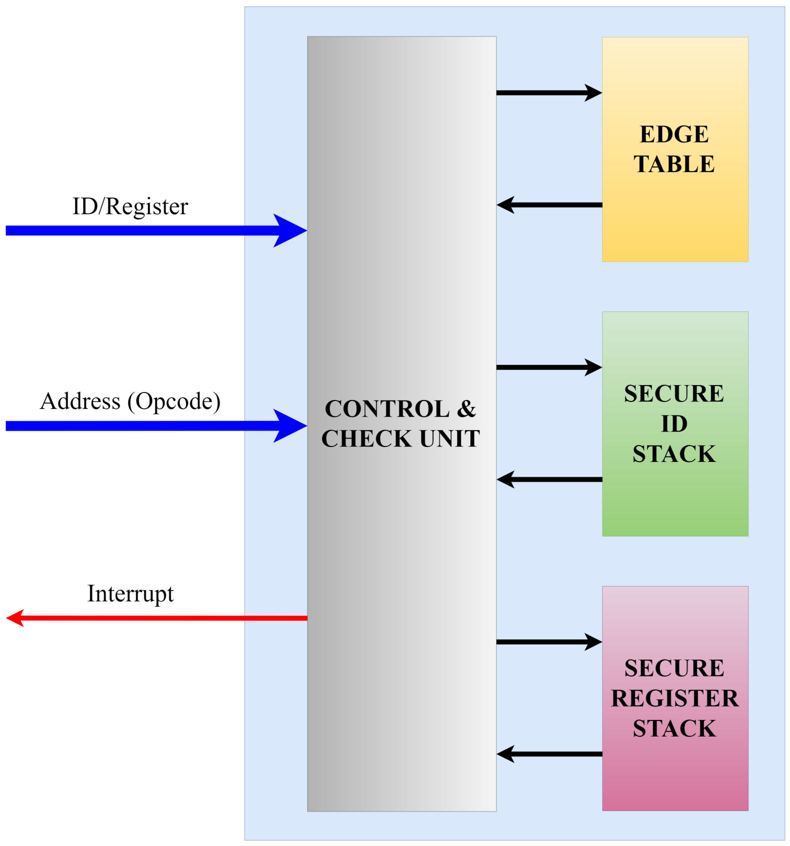
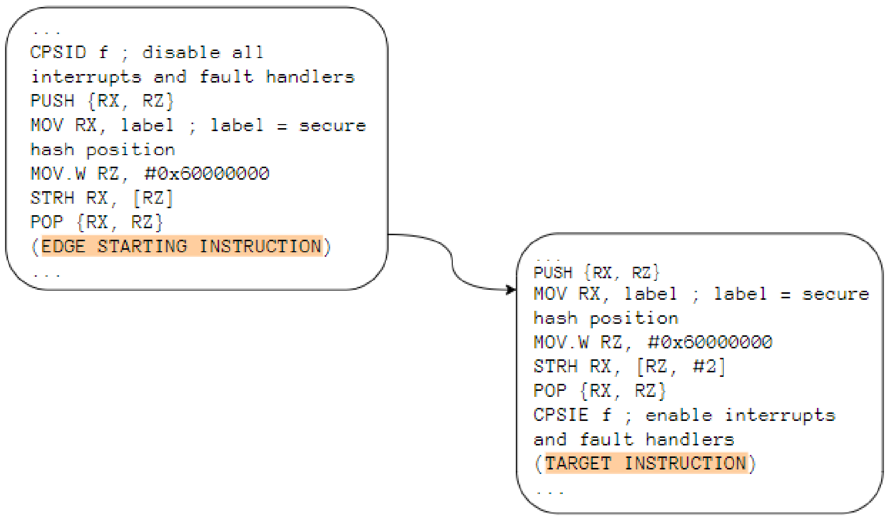


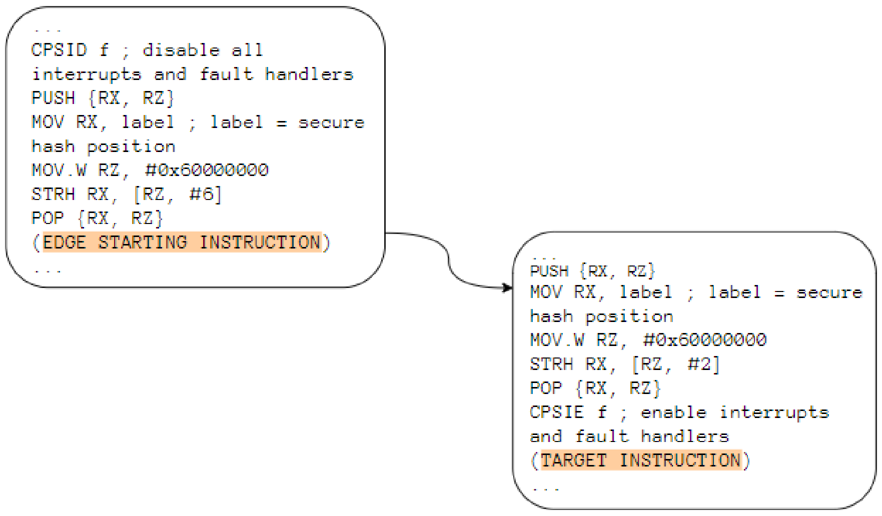
Figure 4.4: CFI monitor schema used for protection [24].



```
...
CPSID f ; disable all
interrupts and fault handlers
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ]
POP {RX, RZ}
(EDGE STARTING INSTRUCTION)
...
```

```
...
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ, #2]
POP {RX, RZ}
CPSIE f ; enable interrupts
and fault handlers
(TARGET INSTRUCTION)
...
```

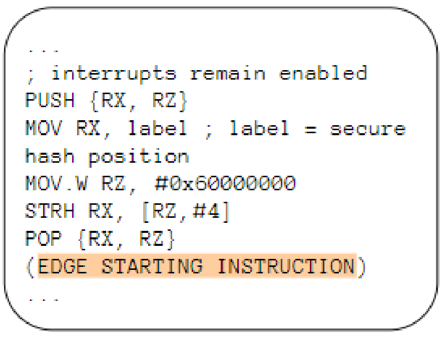
(a) Instrumentation for types 1), 2), 3), 6).



```
...
CPSID f ; disable all
interrupts and fault handlers
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ, #6]
POP {RX, RZ}
(EDGE STARTING INSTRUCTION)
...
```

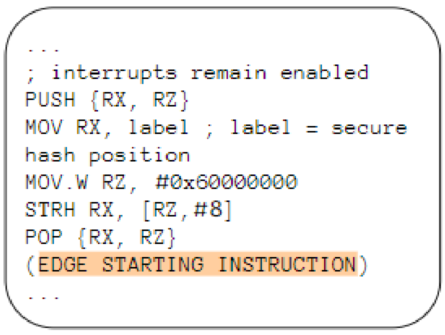
```
...
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ, #2]
POP {RX, RZ}
CPSIE f ; enable interrupts
and fault handlers
(TARGET INSTRUCTION)
...
```

(b) Instrumentation for type 4).



```
...
; interrupts remain enabled
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ,#4]
POP {RX, RZ}
(EDGE STARTING INSTRUCTION)
...
```

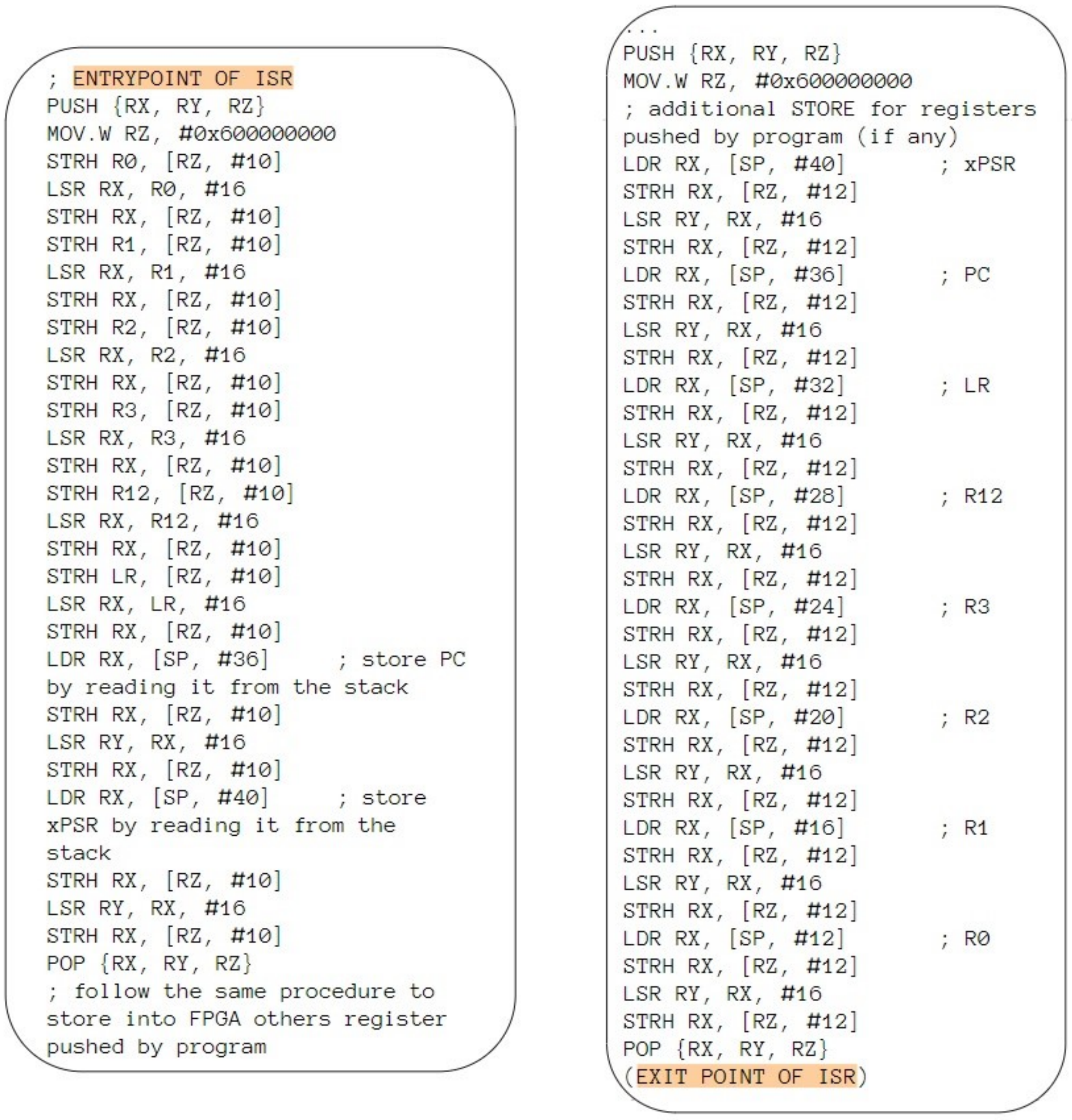
(c) Insturmentation for type 5).



```
...
; interrupts remain enabled
PUSH {RX, RZ}
MOV RX, label ; label = secure
hash position
MOV.W RZ, #0x60000000
STRH RX, [RZ,#8]
POP {RX, RZ}
(EDGE STARTING INSTRUCTION)
...
```

(d) Instrumentation for type 7).

Figure 4.5: Instrumentation code based on edge classification.



```
; ENTRYPOINT OF ISR
PUSH {RX, RY, RZ}
MOV.W RZ, #0x600000000
STRH R0, [RZ, #10]
LSR RX, R0, #16
STRH RX, [RZ, #10]
STRH R1, [RZ, #10]
LSR RX, R1, #16
STRH RX, [RZ, #10]
STRH R2, [RZ, #10]
LSR RX, R2, #16
STRH RX, [RZ, #10]
STRH R3, [RZ, #10]
LSR RX, R3, #16
STRH RX, [RZ, #10]
STRH R12, [RZ, #10]
LSR RX, R12, #16
STRH RX, [RZ, #10]
STRH LR, [RZ, #10]
LSR RX, LR, #16
STRH RX, [RZ, #10]
LDR RX, [SP, #36]     ; store PC
by reading it from the stack
STRH RX, [RZ, #10]
LSR RY, RX, #16
STRH RX, [RZ, #10]
LDR RX, [SP, #40]     ; store
xPSR by reading it from the
stack
STRH RX, [RZ, #10]
LSR RY, RX, #16
STRH RX, [RZ, #10]
POP {RX, RY, RZ}
; follow the same procedure to
store into FPGA others register
pushed by program
```

```
...
PUSH {RX, RY, RZ}
MOV.W RZ, #0x600000000
; additional STORE for registers
pushed by program (if any)
LDR RX, [SP, #40]        ; xPSR
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #36]        ; PC
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #32]        ; LR
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #28]        ; R12
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #24]        ; R3
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #20]        ; R2
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #16]        ; R1
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
LDR RX, [SP, #12]        ; R0
STRH RX, [RZ, #12]
LSR RY, RX, #16
STRH RX, [RZ, #12]
POP {RX, RY, RZ}
(EXIT POINT OF ISR)
```

Figure 4.6: Instrumentation code for ISRs.

Chapter 5

Automatic Tool Features

The goal of the thesis is to provide an automatic tool to support the hybrid solution presented so far. It mainly focuses on realising a Python script that deals with the binary instrumentation process during the offline phase. The following Chapter presents the implemented strategy stages, from the disassembly file analysis to instrumentation scheme, the code organisation, the used data structures and the cooperation with the reverse-engineering framework `Radare2`.

5.1 Adopted Strategies

The analysis of the procedures used for control-flow integrity shows that the effectiveness of the protection mechanisms lies essentially in:

- the extraction of the CFG to establish the correct edges correlation;
- the insertion of the instrumentation code used to protect critical points checked at runtime.

As explained before, the static analysis accuracy does not allow reconstruction of the complete CFG. Mainly, it fails to trivially identify the destinations of indirect jumps, and a more in depth analysis is required. Moreover, it should be taken into account that the execution of ISRs cannot be foreseen by the CFG as a static code analysis generates this.

The objective of the implemented tool is not only to extract the manifest CFG from the firmware, but to enrich it with the missing information to *identify the target of all reachable jumps*, even indirect ones, and protect the program context during the ISRs, as explained in [24]. The described features constitute the starting point of the script development, focusing on realising part of the *offline* phase, and primarily dealing with the automatic process of analysis and code instrumentation.

Before going into the details of the solution, it is necessary to point out that the considered instruction set is the ARM ISA, which is adopted the vast majority of IoT devices on the market. The challenges faced to accomplish a secure solution applicable to any ARM microcontroller are mainly related to:

1. the presence of instructions that alter `PC` value through operations that involve dangerous registers, which content is composed of values coming from memory potentially corrupted;

2. the lack of proper `RET` machine instructions: unlike x86 architecture, that ends the execution of a function with a `RET` instruction, ARM routines terminate with branch statements whose final destination depends on the link register (`LR`) value, like `BX LR`, or from the return address loaded from the top of the stack with a `POP` or load multiple (`LDMIA`) instruction, or from branch statements handled by unconditional direct jump;

3. the application of multiple instructions set inside the same binary file: principally, the instruction can belong to the proper ARM set, with 32-bit instructions, or the Thumb set, in which most of the instructions are encoded on 16 bits, with few others on 32 bits.

All these factors form the basis on which each stage of the script has been developed. Thus, the central aspect to focus on is overcoming the static analysis limitation due to the call graph's partial recovery by reconstructing all possible `PC`-based instruction targets. The tool implementation outlines 5 stages:

1. ***Parsing***: it examines the file containing the disassembly instructions of all the functions employed in the source code by translating it into an assembly file;

2. ***Extraction***: it outlines the general program stream exploring all the performed function calls and retrieves the CFG of those functions that include indirect edges;

3. ***Reconstruction***: it is responsible for determining the storing locations where indirect branch operands transited (registers, memory), tracing their history, and therefore collecting the instructions that contributed to the value;

4. ***Recognition***: it identifies the edges and classifies them based on specifications;

5. ***Instrumentation***: it applies the instrumentation statements based on the discovered edge typologies.

Parsing
P
Extraction
E
Reconstruction
R
Recognition
R
Instrumentation
I

Figure 5.1: Stages of strategy implementation.

The script activity enjoys the support of the external module `r2pipe` [52], that handles the communication with the reverse-engineering framework `Radare2` (`r2`) through pipes. As mentioned on the Radare2 Github repo [53]:

Radare project started as a forensics tool, a scriptable command-line hexadecimal editor able to open disk files, but later added support for analyzing binaries, disassembling code, debugging programs, attaching to remote gdb servers...

During the research, the tool has been widely used both through the `Cutter` [1] version, offering a graphical interface, and `r2pipe` [2] for interaction in *"quiet mode"* directly in the developed script. The offered API receives as a parameter a string executed as a command on the `r2` console and returns as result strings with information about binary analysis. The application requires the installation of `Radare2` on the system on which the Python code runs.

Moreover, the script has 2 launch modes. Both of them perform the entire procedure, receiving as parameter the `<disassembly_listing>` and the `<file_elf>`, but the second one allows to produce a statistic file with the addition of the `"-report"` flag. The generated `report.txt` file includes details about examined files, such as the list of direct function calls, secure jumps, insecure jumps and their resolution and protection schema applied during the instrumentation process. More attention will be given to the various steps' internal structure in the coming Sections, explaining the methodologies and data structures used.

5.2 Code Analysis

The code structure let to better understand strategies and methods adopted in each stage. The practical workflow organisation involves the use of three classes: `Cfi`, `BasicBlock` and `Protection`. The first one, `Cfi`, is the main class and handles the cooperation with the others. It contains data structures necessary to store:

- the disassembly code of the file received as input, in a dictionary that has the hexadecimal address as key and the corresponding instruction as value;
- the association between the direct jump, identified by a hardcoded label (dictionary value), and the analogous position in the generated file, recognised by a unique row (dictionary key);
- the functions traversed starting from the program entry point thought the manifest direct edges;
- the ISRs routines inside the disassembly file;
- the info about basic blocks recovered by CFG reconstruction, in a dictionary that has the BB ID as key and the BasicBlock object as value;
- details on insecure edges, such as the address, function name which belongs to, register involved;

[1] Cutter, the graphical user interface for Radare2, https://cutter.re/

[2] r2pipe, module to script Radare2

- the specifications required for the instrumentation process.

Moreover, the `Cfi` class manages cooperation with the `r2pipe` module. When the main creates the `Cfi`'s instance, the constructor tries to establish the connection to the pipe using the `ELF` filename, related to the disassembly input file, as a parameter of the statement `r2pipe.open(ELF_filename)`. Once completed, the session is created and saved as a class attribute called anywhere in the code. Before the script terminates the execution, `Cfi` closes the connection with the complementary operation `r2.quit()`.

The `BasicBlock` class is defined as a container of helpful information to reconstruct a register's history as an operand of an indirect jump. The essential BB elements are:

- a unique ID, correlated to the line number where the BB is located in the assembly file;
- the name of the function the BB belongs to;
- the list of the incoming edges;
- the list of the outgoing edges;
- the instructions included in the BB;
- a boolean value to mark as *"visited"* the BB when traversing it, and as *"not visited"* when the search ends. It is helpful to avoid infinite loops during the origin tree reconstruction.

Other class attributes keep track of the history of the register involved in the jump and processed by internal methods to derive the final destinations.

The last one is the `Protection` class, whose attributes are the JSON array received from the `Cfi` class and an `option` attribute that manages a switch case to select the type of edge for the instrumentation process.

The UML diagram summarizing the class scheme of the tool is in Figure 5.2.

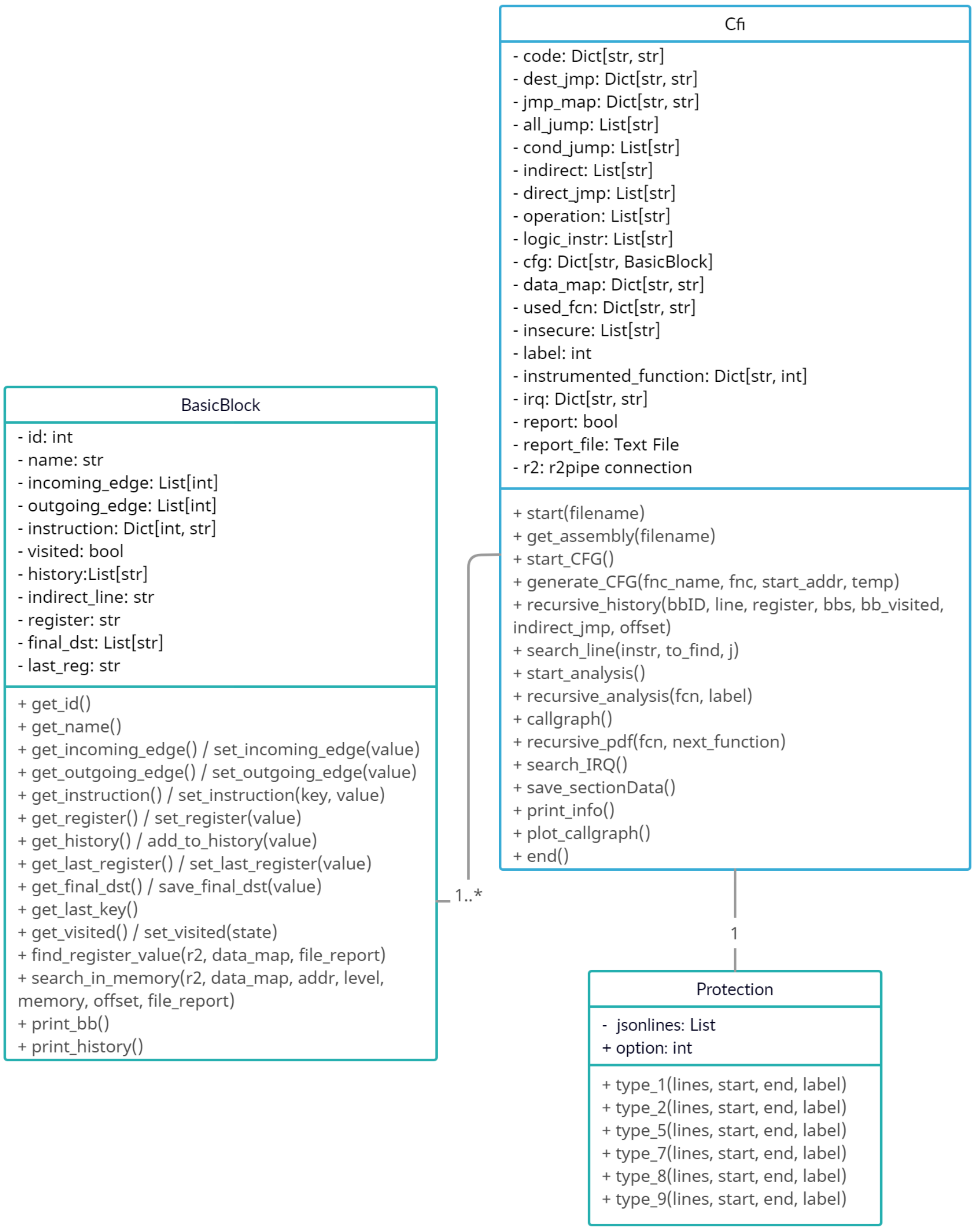

Cfi
- code: Dict[str, str]
- dest_jmp: Dict[str, str]
- jmp_map: Dict[str, str]
- all_jump: List[str]
- cond_jump: List[str]
- indirect: List[str]
- direct_jmp: List[str]
- operation: List[str]
- logic_instr: List[str]
- cfg: Dict[str, BasicBlock]
- data_map: Dict[str, str]
- used_fcn: Dict[str, str]
- insecure: List[str]
- label: int
- instrumented_function: Dict[str, int]
- irq: Dict[str, str]
- report: bool
- report_file: Text File
- r2: r2pipe connection
+ start(filename)
+ get_assembly(filename)
+ start_CFG()
+ generate_CFG(fnc_name, fnc, start_addr, temp)
+ recursive_history(bbID, line, register, bbs, bb_visited, indirect_jmp, offset)
+ search_line(instr, to_find, j)
+ start_analysis()
+ recursive_analysis(fcn, label)
+ callgraph()
+ recursive_pdf(fcn, next_function)
+ search_IRQ()
+ save_sectionData()
+ print_info()
+ plot_callgraph()
+ end()
BasicBlock
- id: int
- name: str
- incoming_edge: List[int]
- outgoing_edge: List[int]
- instruction: Dict[int, str]
- visited: bool
- history:List[str]
- indirect_line: str
- register: str
- final_dst: List[str]
- last_reg: str
+ get_id()
+ get_name()
+ get_incoming_edge() / set_incoming_edge(value)
+ get_outgoing_edge() / set_outgoing_edge(value)
+ get_instruction() / set_instruction(key, value)
+ get_register() / set_register(value)
+ get_history() / add_to_history(value)
+ get_last_register() / set_last_register(value)
+ get_final_dst() / save_final_dst(value)
+ get_last_key()
+ get_visited() / set_visited(state)
+ find_register_value(r2, data_map, file_report)
+ search_in_memory(r2, data_map, addr, level, memory, offset, file_report)
+ print_bb()
+ print_history()
1..*
1
Protection
- jsonlines: List
+ option: int
+ type_1(lines, start, end, label)
+ type_2(lines, start, end, label)
+ type_5(lines, start, end, label)
+ type_7(lines, start, end, label)
+ type_8(lines, start, end, label)
+ type_9(lines, start, end, label)

Figure 5.2: UML class diagram.

5.2.1 Parsing

The code instrumentation consists of the inclusion of specific statements corresponding to the critic *control-flow transfer* instructions. The re-writing binary operation starts from parsing its *disassembly listing* (`".list"` file format), which contains the complete disassembly of the section `".text"` of the firmware. Besides providing readable details about the binary code, this file also contains the disassembly of common library functions used, as the linking is of course static. Thus, it is possible to investigate them and protect against possible attacks derived from memory-corruption vulnerabilities.

Each line of the disassembly file holds 3 parameters: from left to right, it reports the hexadecimal address, the machine language instruction at that location and finally, the mnemonic of the instruction (some lines also have a comment as fourth value, generated automatically). At this stage, the script reads the code listing and converts it into an *assembly* file. This step is crucial, as it is more challenging to apply the instrumentation process in the disassembly file: in fact, each instruction insertion would imply the inclusion of the hexadecimal address that must be calculated consistently, respecting both the alignment and the width of the instruction. To avoid further complications, it works directly on the assembly file.

The translation step processes each row to extract only the assembly instruction via suitable *regular expression* (`regex`[3]). Moreover, the associated hexadecimal address is stored as a unique key in a dictionary not to lose the connection to these pieces of information and facilitate its use with direct access.

The `regex` construction considers instructions on 16 bits and 32 bits inside the same file. Specifically, `regex` applied differs for the match with:

- name of the function;
- branch instruction;
- instruction ending with a comment;
- statement on 16 bits or 32 bits.

In most cases, the single goal is to isolate the address and the instruction, while in the presence of branch instruction, it is also necessary to resolve linked hexadecimal address thought explicit *label* generation. All of these operations lead to the creation of assembly file `"out.s"`, whose text results transformed as in the following examples (Figure 5.3):

[3]Regular Expression 101, website used to create and test regex, `https://regex101.com/`

```
080801d4 <frame_dummy>:
 80801d4:  b508       push {r3, lr}
 80801d6:  4b03       ldr  r3, [pc, #12] ; (80801e4 <frame_dummy+0x10>)
 80801d8:  b11b       cbz  r3, 80801e2 <frame_dummy+0xe>
 80801da:  4903       ldr  r1, [pc, #12] ; (80801e8 <frame_dummy+0x14>)
 80801dc:  4803       ldr  r0, [pc, #12] ; (80801ec <frame_dummy+0x18>)
 80801de:  f3af 8000  nop.w
 80801e2:  bd08       pop  {r3, pc}
 80801e4:  00000000   .word 0x00000000
 80801e8:  2000ca90   .word 0x2000ca90
 80801ec:  0808b714   .word 0x0808b714
```

(a) Disassembly code fragment.

```
frame_dummy:
   push {r3, lr}
   ldr  r3, [pc, #12]  ;  (80801e4 <frame_dummy+0x10>)
   cbz r3,lab0_frame_dummy
   ldr  r1, [pc, #12]  ;  (80801e8 <frame_dummy+0x14>)
   ldr  r0, [pc, #12]  ;  (80801ec <frame_dummy+0x18>)
   nop.w
lab0_frame_dummy:  pop  {r3, pc}
   .word 0x00000000
   .word 0x2000ca90
   .word 0x0808b714
```

(b) Assembly code fragment.

Figure 5.3: Code listing conversion.

5.2.2 Extraction

The *extraction* stage represents the core of the script, together with the reconstruction one. The aim is to track the observed firmware's general flow starting from the *entry point* and traversing the so-called *Global Call Graph* (GCG). The result is a graph that has (i) the entry function as root, identified by the entry point address, (ii) the *direct call* functions as internal nodes and (iii) the routines that end the execution passing the control to the caller as leaves. The term *"direct calls"* only refers to the `BL label` type instructions, as, after performing the routine to which jump, they must return the control to the next instruction after the executed branch statements. Direct calls detected by the unconditional jump are considered safe because they never return to the caller.

The same procedure is valid for ISRs routines, with the difference that it is not possible to establish the exact point in which they will be called. The graphic representation of the GCG is exported in `".png"` format if it is chosen to launch the script in *report* mode.

The GCG extraction allows reducing the number of functions to only parse those that are performed. A recursive algorithm cross over each routine directly called from the entry point. This step finds the indirect branch instructions and *direct calls* to other functions that constitute the subsequent internal nodes of the graph. The search continues recurring on the newly-found nodes, until it reaches a function that does not include direct calls. This last one represents a leaf of the graph, and generally ends with a `BX LR` instruction, to return the control to the caller, or ends with an unconditional jump, as shown in Figure 5.4.

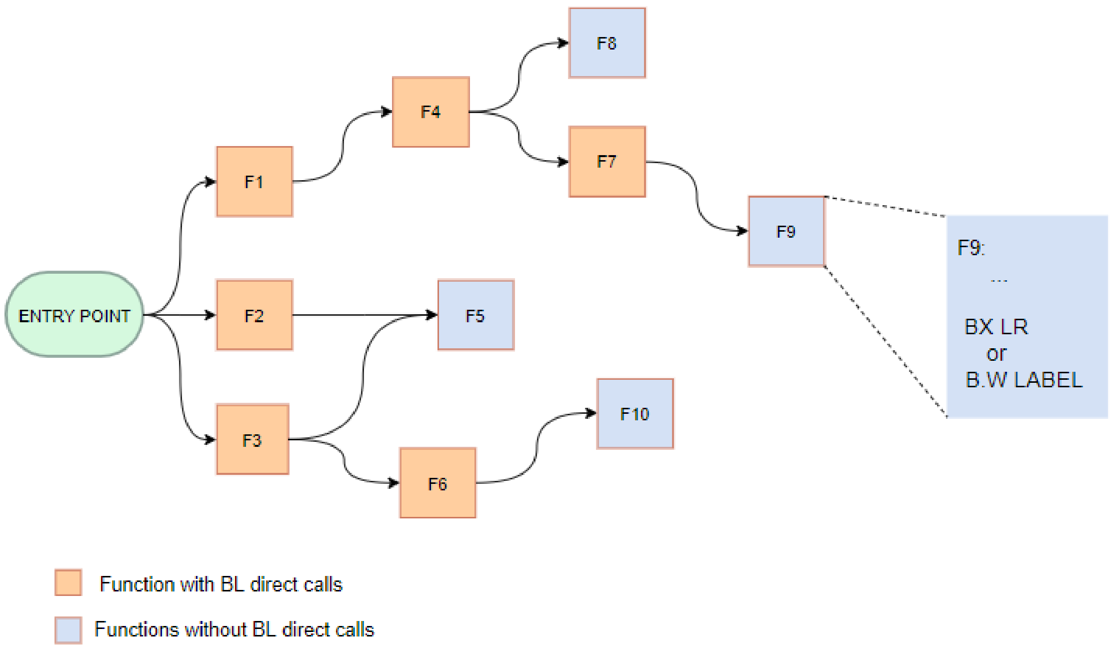

F8
F4
F7
F1
F9
F9:
...
BX LR
or
B.W LABEL
ENTRY POINT
F2
F5
F10
F3
F6
Function with BL direct calls
Functions without BL direct calls

Figure 5.4: Global call graph representation.

In compliance with the *ARM Architecture Procedure Call Standard* (AAPCS) [54], functions can be performed by direct calls with `BL` instructions or indirect calls with `BLX` instructions. The `BL` instruction forces the execution of a routine specified by a *label*, copying its address in `PC` and storing into `LR` the return address, which is equal to the next instruction in sequence. In the easier case, it passes the control to the called function, performs the instructions and returns to the caller with `BX LR` statements. It is possible to state that `BX LR` *is a secure edge if* `LR` *is not modified from the branch to the moment in which the return address is restored.* If `LR` suffers alteration, it could be tampered with to hijack the flow anywhere, so it needs protection.

Not all the function calls follow this schema. In other cases, the return address is pushed onto the stack before branching. If the function requires more than three parameters to work, the first subroutine instruction will be `PUSH` or `STMDB` instructions, involving of a variable number of registers (`R4-R9`, `R10`, `R11`) and the return address contained in `LR`. Before passing to the callee, it restores the register values and writes return address in the `PC`.

The `BLX` instruction defines a forward indirect edge to an address stored in a register and whose target destination cannot be determined with static analysis. The program flow can be altered if the register is formed on values from the memory section at risk of corruption. In general, the adversary exploits the instructions with PC as the final destination by updating the address with a constant value, received input or data coming from memory to hijack the execution to unpredictable locations. It is necessary to recover the *"origin tree"* of the register involved in the indirect edge to determine all the reachable targets.

Once found all the indirect jumps, the tool proceeds with the CFG generation, only for the functions that include the branch, in such a way to provide with helpful data structure the algorithm responsible for tracking history and limit the operation only to passed function. The recovery operation identifies the BB chain that forms the function, saving for each BB (i) a unique ID, (ii) the list of incoming and outgoing edge and (iii) the instructions. All of these connection between BBs determine the CFG for each function (Figure 5.5).

The general idea is to extract CFG *"on-demand"*, so the tool does not create it for all functions in the file, but only for those that must be processed for the production of the origin tree. Thus, it begins to trace the history, parsing the function's instruction containing the insecure edge, and *"demand"* the CFG for the caller function only if the end condition is not reached. In this way, it avoids instantiating BB objects that will never be passed as not involved in the origin tree.

Consequently, the current step and the reconstruction one are strictly dependent, which increases the algorithm complexity. However, the adopted strategy is more convenient than performing the extraction for all the functions related to memory occupation and execution times.

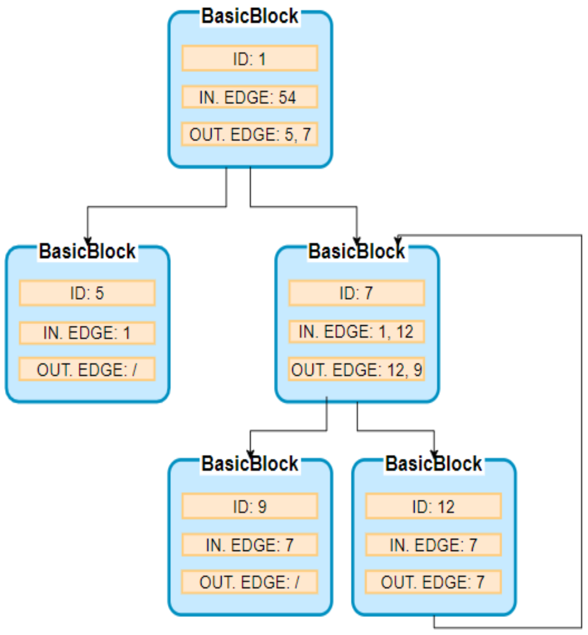


Figure 5.5: Chain of BBs obtained from CFG's reconstruction.

5.2.3 Reconstruction

The reconstruction of the origin tree follows the theory illustrated in the previous Chapter. First, the script analyzes the indirect jump instruction to locate the register whose content is of interest. It *traces back* its history until it reaches all the locations of memory or constants that make up the result. If the resulting address is constructed from a code memory address, it is not considered insecure, since the code memory is assumed as incorruptible, while if the final address belongs to the data section, it is considered potentially insecure and it needs protection. The main branch typologies found in firmware disassembly are:

- `BLX Rm`;
- `BX LR` (if `LR` is involved in others operation before return to its caller);
- `TBB[PC, Rm]` or `TBH[PC, Rm]`;
- `LDR.W PC,[Rn, Rm, LSL #imm]`;

The last two instructions usually translate the `switch-case` statements. In `TBB` or `TBH`, it copies directly to `PC` the value of the register and can get all the destinations through `R2`, as they are all as the outcoming edge of the BB that contains the only `TBB` statement. While in the case of `LDR`, the value of `PC` is formed by `Rn`'s address pointing to a table to which is added `Rm`, representing the offset. Then it needs to find the values of these two registers to reconstruct the value of `PC`. The alternative is to recover only the value of `Rn`, which is the starting address of the table and identifies all those contained in it as possible values. This choice is made because the offset `Rm, LSL #imm` indicates that the function is within a loop, and as `Rm` changes, it is possible to have access to several addresses in the table, if not all.

In other cases, the tracking process depends on the complexity of the code. The task is executed by a recursive algorithm that starts from the jump instruction and goes back to the previous BBs checking the incoming edges, looking for the first instruction that uses that register to save the result. The crossed blocks are marked as *visited* until the exploration ends, to avoid infinite loops during the process. The instruction is saved in a list that is defined as an attribute of class BB. Before proceeding, it updates the register to be searched using an operand to produce the current instruction result.

The analysis studies the different ARM instructions encountered, from arithmetic to logical operations, move, load and store operations. If the jump depends on a register whose value is loaded from the stack via *Stack Pointer* (`SP`), the tool must keep track of all the operations in which `SP` is involved, i.e., `PUSH`-like and `POP`-like instructions, to know the location to which `SP` points and recover its contents. In that case, the register to look at is the one pointed by `SP`, and the reconstruction operations consider how it is increased or decremented according to the `PUSH`/`POP` encountered.

The search continues within the current BB until all instructions have been read and then resumes its path to the previous BB. The access happens directly considering that the current BB's incoming edge coincides with the previous BB's ID. Once the BB is changed, the same considerations are repeated with every recursive call until the termination condition is reached, generally expressed by `LDR Rm, [PC, #imm]`. This format is used to load in `Rm` a constant value stored in the `.text` section, whose position is relative to the current

`PC`. Therefore, it is the starting point for recover the final register value to which jump by following the flow of found history instructions.

If the termination condition is not found in the BB of the function that contains the jump, as mentioned before, it is necessary to generate the CFG of the caller function. Then, it resumes the search from the jump instruction and follows the path defined backwards. For simplicity, we have defined a limit of requests of the CFG to a maximum of 2, after which the script raises the alarm. Otherwise, it becomes too complicated to reconstruct history. In general, the source tree's complexity is closely related to the complexity of the firmware code written by the programmer. The *extraction* and *reconstruction* procedure is summarized in Figure 5.6.

Thus, the tool processes the history in reverse order. The BB method reads every statement in the history list, isolates operands, and *"emulates"* the ARM statement, saving the final record at the registry used to contain the result. If there are `LDR` or `STR` instructions, the method interacts with a copy of the data map to recover the indicated location or store the value that could be accessed later.

If the termination condition points to a code memory address, it is unnecessary to continue to process the other instruction because it brings a secure position in the code. While if it leads to a data memory address, the script must recover all the achievable destinations.

5.2.4 Recognition

After identifying all the target destinations, it is necessary to classify each edge depending on defined typologies. The mechanism follows the GCG as initially, but this time the flow contains additional information recovered from indirect jump resolution. For each critical edge encountered, the tool creates a JSON element that includes, as fields, edge *type*, *label*, the position of the *source* instruction and the related *target* instruction of the branch.

5.2.5 Instrumentation

The last step is the instrumentation process applied to assembly statements obtained in the first stage. The additional instructions are inserted by parsing the JSON array previously created. The *"type"* value constitute a filter for choosing the appropriate instrumentation code based on edge classification, as shown in Figure 4.5 and 4.6.

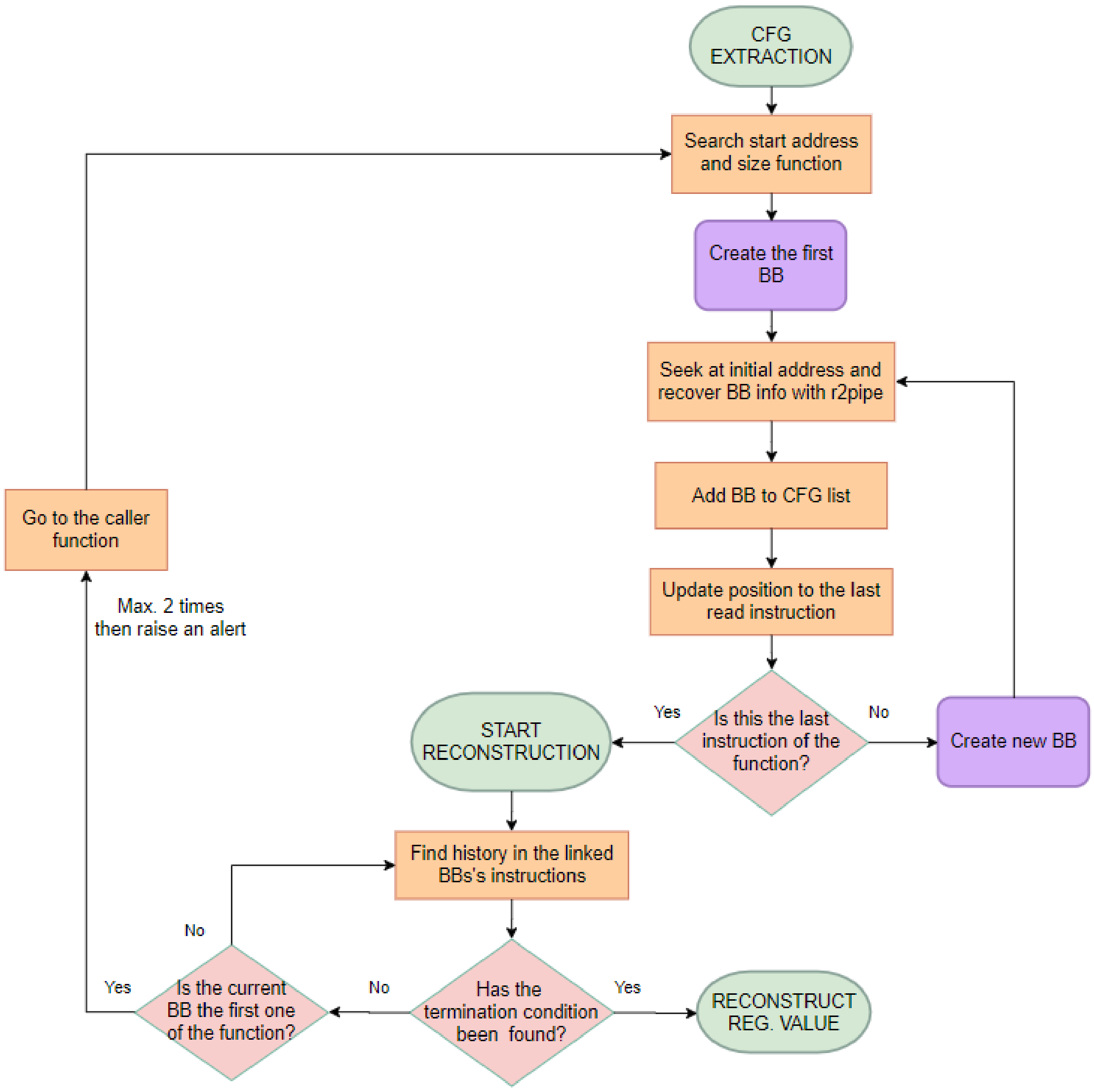

CFG EXTRACTION
Search start address and size function
Create the first BB
Seek at initial address and recover BB info with r2pipe
Add BB to CFG list
Update position to the last read instruction
Is this the last instruction of the function?
Yes
No
Create new BB
START RECONSTRUCTION
Find history in the linked BBs's instructions
Has the termination condition been found?
Yes
No
RECONSTRUCT REG. VALUE
Is the current BB the first one of the function?
Yes
No
Go to the caller function
Max. 2 times then raise an alert

Figure 5.6: Flowchart of Extraction and Reconstruction procedure.

Chapter 6

Experimental Results

The present Chapter offer some results extracted from the performance analysis of the Python tool. The division in 5 stages of the entire procedure allows isolating and studying each of these singularly to estimate the applied algorithm's efficiency.

The Python tool `Radon`[1] evaluates the code complexity by:

- *Cyclomatic complexity*: it corresponds to the number of decision blocks that determine the number of linearly independent paths and assigns a rank from A *simple block* to F *very complex block*;
- *Raw metrics*: it gives details about the total number of lines of code (LOC) and logic lines of code (LLOC);

Table 6.1 collects the main value of these metrics for each class to have a general overview. The main class `Cfi` has greater average complexity than the other two. Rank C, which means *"slightly complex blocks"*, is attributed to the fact that more complex methods can be found there. In particular, the dependence is more significant in those dealing with the CFG's *extraction* and the *reconstruction* of the jump origin tree. The presence of numerous decision blocks used to find the proper instruction processed, derive the register on which to make the next recursive call, and verify the termination condition, dramatically increases the method's complexity, but is unavoidable for collecting the right instructions.

Class	N. Blocks	LOC	LLOC	Avg. Complexity
Cfi	17	1047	920	C
BasicBlock	27	433	401	A
Protection	9	150	64	A

Table 6.1: Code analysis.

[1]Radon's documentation, https://radon.readthedocs.io/en/latest/

Besides, the *reconstruction* phase is closely linked to the extraction phase of CFG. The method examines all the instructions included in its path, going back until it reaches the root. In the best case, the termination condition is within the function. Therefore, the set of instructions and the process of reconstruction of the final value can be obtained quickly. In the worst case, it will demand the extraction of the caller's CFG, and it will have to repeat the whole procedure.

All this strongly depends on the firmware code structure and how the calls to the other functions are defined in the firmware source code. If the code is linear enough, the disassembly will hardly contain indirect jump instructions to be solved. For example, if in the `main()` function, the programmer handles calls to other functions within a `switch-case` statement, this will be automatically translated through an indirect jump instruction access to the jump table.

Another costly operation in terms of execution times (but not complex at the code structure level) is the *parsing* step. At this stage, the script must read all the file lines, filter the data, and rewrite it into another file. The use of appropriate `regex` simplifies finding instructions in each row, but the resolution of addresses through labels at the jumps adds additional costs. The execution time of the stage, in general, undoubtedly depends on the number of lines to process.

The other classes, `BasicBlock` and `Protection`, are instead composed of effortless methods concerning the activities they perform. For example, the *instrumentation* process reads the information obtained from the previous step and inserts the correct instructions based on the jumps' classification by making direct access to the position where it goes to perform the writing operation.

The script validity has been demonstrated by testing different benchmarks and comparing the obtained results (Table 6). These benchmarks have been chosen from the MiBench online library [2], which provides common benchmarking programs to test embedded systems' performance. The Table reports the number of instructions composing the binary of the firmware after compilation and with no protection, the execution times of each of the tool steps (in seconds), the number of direct calls and of insecure edges found, and the final number of machine instruction after the application of the protection.

[2]http://vhosts.eecs.umich.edu/mibench//

Benchmark	Instr. (no prot.)	Parsing (s)	Direct Call	Extraction (s)	Insec. Edge
BITCOUNT	20554	9.0035167	177	6.1345312	11
DIJKSTRA	20529	8.3118812	186	7.2268513	11
SHA	13663	7.6490014	127	4.6812931	14
RIJNDAEL	25685	9.7907316	197	13.937783	11
CRC	20320	8.3461010	178	6.3364535	10
STRING	12960	7.2736033	127	4.4640792	14

Benchmark	Reconstruction (s)	Recognition (s)	Instrumentation (s)	Instr. (prot.)
BITCOUNT	2.0416360	0.9700785	0.1179562	21366
DIJKSTRA	1.9285040	0.9283237	0.0988015	21327
SHA	1.5535502	0.2436774	0.1224835	13959
RIJNDAEL	2.4204537	1.0149302	0.1239938	26494
CRC	1.8562357	0.8226949	0.0969530	21042
STRING	1.5635721	0.1432315	0.0667889	13217

Table 6.2: Experimental results.

Benchmark's instrumentation process was lanched 100 times for each program to derive more accurate average values. Note that, for instance, the *parsing* phase's execution time grows linearly with the number of input file instructions, as well as the increase of direct calls also raises the extraction time mostly related to the determination of GCG, obtained through a recursive algorithm. This trend relies on the number of functions created and used by the programmer, plus the library functions invoked. Instead, *recognition* and *instrumentation* operations are among the simplest in code structure, and execution times are almost negligible compared to those at other stages.

The Python library `memory_profiler`[3] allows obtaining further analysis by monitoring the process line by line and outlines how the various methods' memory profile varies. The result is shown in a plot to highlight the points where there is greater use of memory. For simplicity, the profile examined is an extract of the overall result (Figure 6.1), as they are the operations to require more memory. The initial curve grows in correspondence with the increase of the extracted instructions from the disassembly and memorized in the dictionary; then, it has an almost constant track during the analysis of the GCG, and the BB's extraction of the functions contain the indirect jumps. The peaks, instead, are in correspondence with the examination of the BB and the paths' reconstruction. The part repeated after the dotted line is related to the request of extraction of the CFG of the calling function and its reconstruction, which, as we see, has the same trend as the previous phase.

[3]Memory Profiler Github page, `https://github.com/pythonprofilers/memory_profiler`

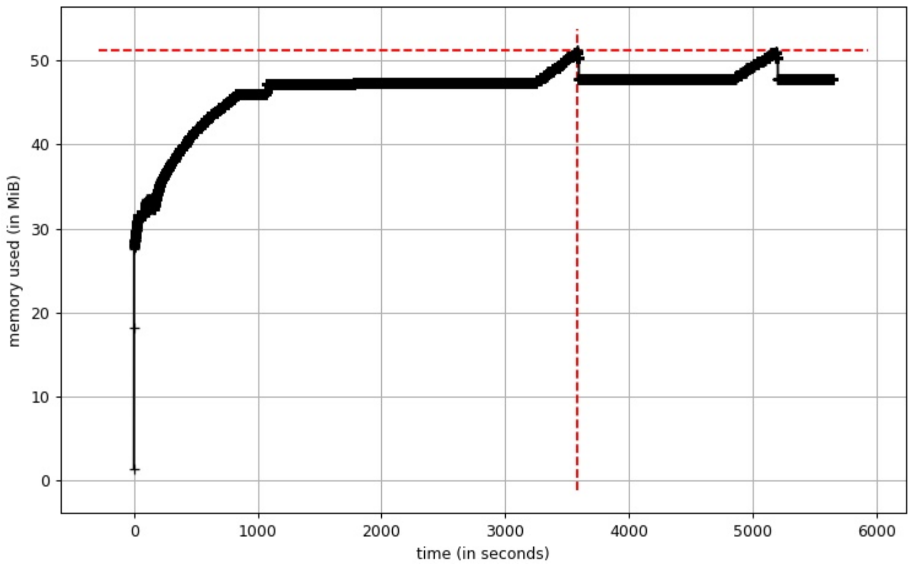


Figure 6.1: Plot of memory profiler.

Chapter 7

Conclusions and future works

This thesis aims to provide an automatic Python tool capable of extracting the CFG for those functions containing control-flow transfer instruction and instrumenting the program's binary code to safeguard its activity from memory corruption vulnerabilities. The paper *"A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems"* [24], published by Prof. Paolo Prinetto and his PhD team at Politecnico di Torino, constitute the starting point of the script development. The research focuses on realising part of the *offline* phase of a hybrid CFI technique for protecting embedded system, which merges static binary instrumentation with the realisation of a CFI monitor summarised in FPGA.

The objective of the implemented tool is not only to extract the manifest CFG from the firmware, but to enrich it with the missing information, derived from *static binary analysis*, to *identify the target of all reachable jumps*, even indirect ones, and protect the program context during the ISRs. The protection mechanism is implemented at the binary level for ARM-based architecture and so suitable for the vast majority of IoT devices on the market.

Starting from the disassembly code analysis, the tool reconstructs the CFG for those function that contains *insecure* control-flow transfers, i.e., those whose final location depends on a value that has even partially transited through areas of memory at risk of corruption. Compared with others' solution, the developed script offers a strategy for finding the indirect edges' target. Implementing a recursive algorithm allows tracing the origin path of the involved register to estimate all the valid destination target addresses. According to edge classification, once source-destination pairs have been identified, the enforcement proceeds with the instruction-level instrumentation on assembly representation.

The study proves that it is always possible to reconstruct the origin tree of a jump to identify the memory locations that make up the destination address. The results extracted from the Python tool performance analysis show that the average complexity is mainly related to the fact that there are methods more elaborated than others. While some of these are composed of effortless methods concerning their activities, like recognition and instrumentation phases, others functions are strictly conditioned by other methods' activity. In particular, the dependence is more significant in those dealing with the CFG's *extraction* and the *reconstruction* of the jump's origin tree. So, execution time and memory consumption rely on the depth of the origin tree. If the register's history involved in a branch is related to few instructions, the final target destination can be obtained quickly. Otherwise, it requires a further CFG extraction to achieve the desired outcome.

Thus, the realisation of the entire procedure, the execution time, and the memory use strongly depend on the complexity of the programmer's code, the library functions used and how the disassembly code translates the program instructions. The use of Python allows building a robust, flexible and portable application. Besides, it offers the ability to take advantage of external modules, such as `r2pipe`, to interact with the `Radare2` reverse-Engineering framework and analyse the firmware binary in simple steps. The main disadvantage is that Python uses large amounts of memory for application implementation, but it is a good trade-off compared to other programming languages' performance.

The tool is tested on different benchmarks with positive results, but improvements are expected in the future. First, it is worth to optimise the most complex methods to improve performance for the running time and the memory used. Moreover, the process that operates on the origin tree data must be extended to all possible ARM instructions, as the tool now identifies just a reduced set composed by the most common instructions and *emulates* the operation to find the final result. For example, the set should include all the logic operation and all the arithmetical ones that work with *flags*, even if this kind of statements are hardly ever involved in the branch origin tree.

Acknowledgements

I would like to thank Professor Paolo Prinetto for giving me the great opportunity to work on this thesis. Infinite gratitude goes to Gianluca Roascio and Nicolò Maunero for having guided and encouraged me with patience in developing the project and the final elaboration, and for having always been super professional and helpful.

A special thanks go to all the people who have joined me on this troubled but satisfying journey. Thanks to all the family and my fantastic parents for the endless love and support. I would like to thank Angela, Elena, Gaia and Marina for always making me feel at home and having shared joys and desperation on alternate days. Thanks to my boyfriend, Luigi, for always being there and not making me lose (entirely) mental health. I also thank my little Lucio for the study breaks and for the affection shown despite the distance.

Without you there, this adventure would not have been the same.

Bibliography

[1] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. «Protecting bare-metal embedded systems with privilege overlays». In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 289–303.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. «Control-flow integrity principles, implementations, and applications». In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[3] K. S. Kumar and D. Malathi. «A Novel Method to Find Time Complexity of an Algorithm by Using Control Flow Graph». In: *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*. 2017, pp. 66–68. DOI: `10.1109/ICTACC.2017.26`.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. «Control-flow integrity». In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM. 2005, pp. 340–353.

[5] *CWE-401: Missing Release of Memory after Effective Lifetime*. `https://cwe.mitre.org/data/definitions/401.html`. [Online; accessed 03-March-2020]. 2020.

[6] *CWE-416: Use After Free*. `https://cwe.mitre.org/data/definitions/416.html`. [Online; accessed 03-March-2020]. 2019.

[7] *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*. `https://cwe.mitre.org/data/definitions/119.html`. [Online; accessed 28-October-2019]. 2019.

[8] H. Shacham, M. Page, B. Pfaff, EJ. Goh, N. Modadugu, and D. Boneh. «On the effectiveness of address-space randomization». In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 298–307.

[9] C. Cowan, S. Beattie, RF. Day, C. Pu, P. Wagle, and E. Walthinsen. «Protecting systems from stack smashing attacks with StackGuard». In: *Linux Expo*. 1999.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. «Buffer overflows: Attacks and defenses for the vulnerability of the decade». In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. IEEE. 2000, pp. 119–129.

[11] *The Heartbleed Bug*. `https://heartbleed.com/`. [Online; accessed 03-March-2020]. 2020.

[12] A. One. «Smashing the stack for fun and profit». In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[13] Microsoft Support. *A detailed description of the Data Execution Prevention (DEP).* `https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in`. [Online; accessed 05-March-2020].

[14] PaX Team. *PaX Non-Executable Pages Design and Implementation.* `https://pax.grsecurity.net/docs/noexec.txt`. [Online; accessed 05-March-2020]. 2003.

[15] O. Stecklina, P. Langendörfer, and H. Menzel. «Design of a tailor-made memory protection unit for low power microcontrollers». In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES).* IEEE. 2013, pp. 225–231.

[16] *Getting around non-executable stack (and fix).* `https://seclists.org/bugtraq/1997/Aug/63`. [Online; accessed 05-March-2020]. 1997.

[17] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. «Jump-oriented programming: a new class of code-reuse attack». In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM. 2011, pp. 30–40.

[18] H. Shacham et al. «The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).» In: *ACM conference on Computer and communications security.* New York. 2007, pp. 552–561.

[19] A. Sadeghi, S. Niksefat, and M. Rostamipour. «Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions». In: *Journal of Computer Virology and Hacking Techniques* 14.2 (2018), pp. 139–156. ISSN: 2263-8733. DOI: `10.1007/s11416-017-0299-1`. URL: `https://doi.org/10.1007/s11416-017-0299-1`.

[20] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. «Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications». In: *2015 IEEE Symposium on Security and Privacy.* 2015, pp. 745–762. DOI: `10.1109/SP.2015.51`.

[21] Y. Guo, L. Chen, and G. Shi. «Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications». In: *2018 IEEE Conference on Communications and Network Security (CNS).* 2018, pp. 1–9. DOI: `10.1109/CNS.2018.8433189`.

[22] L. Deng and Q. Zeng. «Exception-oriented programming: retrofitting code-reuse attacks to construct kernel malware». In: *IET Information Security* 10.6 (2016), pp. 418–424.

[23] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. «Return-oriented programming: Systems, languages, and applications». In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), p. 2.

[24] N. Maunero, P. Prinetto, G. Roascio, and A. Varriale. «A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems». In: *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS).* IEEE. 2020, pp. 1–10.

[25] MA. Laurenzano, MM. Tikir, L. Carrington, and A. Snavely. «Pebil: Efficient static binary instrumentation for linux». In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS).* IEEE. 2010, pp. 175–183.

[26] Z. Huang, T. Zheng, Y. Shi, and A. Li. «A dynamic detection method against ROP and JOP». In: *2012 International Conference on Systems and Informatics (ICSAI2012)*. 2012, pp. 1072–1077. DOI: 10.1109/ICSAI.2012.6223219.

[27] Z. J. Huang, T. Zheng, and J. Liu. «A dynamic detective method against ROP attack on ARM platform». In: *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*. 2012, pp. 51–57. DOI: 10.1109/SEES.2012.6225491.

[28] D. Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. no starch press, 2018, pp. 224–263.

[29] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. «From hack to elaborate technique—a survey on binary rewriting». In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–37.

[30] T. Bletsch, X. Jiang, and V. Freeh. «Mitigating code-reuse attacks with control-flow locking». In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM. 2011, pp. 353–362.

[31] Y. 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. DOI: 10.1109/DSN.2012.6263958.

[32] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.R. Sadeghi. «MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones.» In: *NDSS*. Vol. 26. 2012, pp. 27–40.

[33] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, et al. «Analyzing parallel programs with pin». In: *Computer* 43.3 (2010), pp. 34–41.

[34] AR. Bernat and BP. Miller. «Anywhere, any-time binary instrumentation». In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 2011, pp. 9–16.

[35] Y. Li, Z. Dai, and J. Li. «A Control Flow Integrity Checking Technique Based on Hardware Support». In: *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. 2018, pp. 2617–2621. DOI: 10.1109/IAEAC.2018.8577547.

[36] 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.

[37] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. «Silicon physical random functions». In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 148–160.

[38] J. Daemen and V. Rijmen. «AES proposal: Rijndael». In: (1999).

[39] H. Ozdoganoglu, CE. Brodley, TN. Vijaykumar, and BA. Kuperman. «Smashguard: A hardware solution to prevent attacks on the function return address». In: *Technical Report* (2000).

[40] 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.

[41] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer. «Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems». In: *IEEE Embedded Systems Letters* 10.3 (2018), pp. 87–90. ISSN: 1943-0663. DOI: 10.1109/LES.2018.2819983.

[42] A. Francillon, D. Perito, and C. Castelluccia. «Defending embedded systems against control flow attacks». In: *Proceedings of the first ACM workshop on Secure execution of untrusted code*. ACM. 2009, pp. 19–26.

[43] 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.

[44] W. Wang, M. Liu, P. Du, Z. Zhao, Y. Tian, Q. Hao, and X. Wang. «An Architectural-Enhanced Secure Embedded System with a Novel Hybrid Search Scheme». In: *2017 International Conference on Software Security and Assurance (ICSSA)*. 2017, pp. 116–120. DOI: 10.1109/ICSSA.2017.14.

[45] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A. Sadeghi. «LO-FAT: Low-Overhead control Flow ATtestation in hardware». In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3062276.

[46] M. Milenković, A. Milenković, and E. Jovanov. «A framework for trusted instruction execution via basic block signature verification». In: *Proceedings of the 42nd annual Southeast regional conference*. 2004, pp. 191–196.

[47] 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.

[48] L. Davi, M. Hanreich, D. Paul, A.R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. «HAFIX: hardware-assisted flow integrity extension». In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM. 2015, p. 74.

[49] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. «Hcfi: Hardware-enforced control-flow integrity». In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM. 2016, pp. 38–49.

[50] J. Yiu. *The definitive guide to the ARM Cortex-M3*. Newnes, 2009.

[51] 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.

[52] Radare2 Team. *Radare2 r2pipe GitHub repository*. https://github.com/radareorg/radare2-r2pipe. 2017.

[53] Radare2 Team. *Radare2 GitHub repository.* https://github.com/radare/radare2. 2017.

[54] ARM Limited. *Procedure call standard for the ARM architecture.* http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf. 2009.