POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

PROLEPSIS: Binary Instrumentation Tool for Control-Flow Integrity in ARM and RISC-V

Author: Alessandro Iandoli

Advisor: Paolo Ernesto Prinetto
Co-Advisor(s): Gianluca Roascio

September, 2022

Abstract

The Internet of Things, also shortened with the term IoT, corresponds to an ecosystem made of web-enabled smart devices (the IoT devices), implemented as embedded systems, which collect, send and take other actions on the data that they acquire from their environment. IoT started to grow rapidly in recent years, leading to an increasing usage of embedded systems in the daily life of common people, and in critical infrastructures of companies like SCADA systems. For this reason IoT devices started to be the target of different cyber-attacks with the objective of *controlling the functionalities* of the embedded system or *exfiltrating the data* that are treated. Tipically, cyber-attacks are carried out by exploiting software vulnerabilities present in the code executed by victim system.

The software application that are executed by embedded systems are usually written with the C/C++ language, which provides total freedom to the developer in terms of interaction with hardware resources especially memory. Since memory management is all entailed to the developer, this may lead to create *memory corruption vulnerabilities* in the program, like buffer overflows or use-after-frees that can be exploited by an attacker to hijack the execution flow and possibly execute remote commands on the vulnerable systems.

Software protection techniques like *Data Execution Prevention* were introduced in order to prevent such attacks, but nowadays *Code-Reuse Attack techniques* like *Return-Oriented Programming* (ROP) or *Jump-Oriented Programming* (JOP) can bypass such mechanisms.

Such attack techniques consist in redirecting the execution flow to a chain of so-called *ROP/JOP gadgets*. A gadget consist of a small piece of code inside the vulnerable program image (few instructions), ending either with a return or with a jump instruction, that will redirect the execution flow to the next gadget in the chain. The chain is created by the attacker with the aim of executing arbitrary commands on the vulnerable system.

The countermeasure to Code-Reuse Attack is represented by *Control Flow Integrity* solutions (CFI). CFI aims at guaranteeing the correcteness of the execution flow by retrieving the set of valid destinations for each control-flow transfer instruction, and constraining that instruction to redirect the execution flow only to a valid destination in the set. The constraint is enforced by inserting instrumentation code in the proper locations in the program.

The aim of the present thesis is to provide a tool that can automatically retrieve the set of destinations for each control-flow transfer instruction, by using *binary analysis* on the program, and then instrument the program with instructions that are provided externally by the user, with the aim of making the program *resistant* to Code-Reuse attack techniques and other attacks aiming at hijacking the execution flow of a program.

The tool was written in Python and exploits the abstractions provided by the reverse

engineering framework `Radare2` for program analysis. The interaction with the `Radare2` is carried out by leveraging the primitives provided by the `r2pipe` module.

The tool was integrated with two existing CFI solutions, one for ARM platform and the other one for RISC-V.

Analyzing the results it was noticed how the solution itself, heavily affect the overhead in terms of memory costs. In particular, for RISC-V solution were defined custom instructions, for the instrumentation code, by modifying the toolchain. The possibility to define custom instructions and delegate most of the work to the hardware, significantly reduces the overhead on the final executable size for the RISC-V solution, with respect to ARM one.

Contents

Chapter 1

Introduction

The technological process and the demand for new kind of services brought to the creation of several smart devices, each one in charge of a specific function, interacting among them or with existing information systems. This kind of architecture brought to the creation of the so-called *Internet of Things* (IoT), an ecosystem of physical objects with sensors, processing capabilities, software and other technologies that connect and exchange data with other devices and systems over the Internet or other communications networks. This new kind of architecture brought to an increasing production of embedded systems, that now are employed in a wide range of applications, like medical devices or control systems architecture like SCADA applied in industrial environments.

In order to reduce power consumption and increase performances, applications running on embedded systems are executed directly on *bare metal*, meaning that there is no middleware layer between the hardware and the application software, like an operating system for desktop or mobile applications. The impossibility to rely on a middleware like an operating systems is critical for security considerations, since no native protection mechanisms are provided by default when an application is executed in an embedded system. Furthermore, IoT devices are tipically exposed on Internet in order to communicate with other devices.

These characteristics make these devices appealing targets for threat actors, given the wide attack surface offered in terms of software and hardware vulnerabilities. A cyber threat that is able to exploit a vulnerability in such kind of device might be able to: inject code in the firmware, alter his behaviour, exfiltrate sensible data manipulated by the device, make the device part of a botnet.

Design of solutions providing security features to embedded systems like Confidentiality, Integrity and Availability (CIA) must consider some constraints raised by limited hardware resources, such as:

- reduced memory and computational resources;
- lack of support for task isolation;
- non-applicability of most existing solutions to ARM-based architectures;
- lack of middleware like an operating system or hypervisor standing between hardware and software application;

The wide attack surface offered by these devices and the lack of countermeasures against attack techniques continuously evolving, contributed to the definition and development of defensive techniques. Among them, Control Flow Integrity (CFI) [1] proved to be effective to prevent memory corruption vulnerabilities. The idea of Control Flow Integrity is to have a monitor that monitors the execution flow of the monitored program and prevents branches not allowed. Before the program (and the monitor) are started, it is necessary to extract the Control-Flow Graph of the application through static analysis [3], and find what are all the allowed branches that will be provided to the monitor in order to perform security checks at execution time.

The objective of the thesis is to provide a Python written tool, referred to as *PROLEPSIS*, performing static and dynamic analysis and binary instrumentation of applications for Control-Flow Integrity purposes. PROLEPSIS takes as input custom instrumentation code that can be used to instrument the target application, allowing to integrate the tool with existing Control-Flow integrity solutions.

In particular the tool was integrated with the Control-Flow integrity solution defined here [30]. The solution is based on an external monitor consisting of a *Field Programmable Gate Array* (FPGA), a reprogrammable hardware module that interacts with the CPU in order to guarantee integrity of the execution flow.

The remainder of the document is organized as follows. Chapter 2 discusses what are the main memory corruption vulnerabilities affecting software application and the corresponding software protections, defined in order to prevent attacks exploiting such vulnerabilities. Chapter 3 discusses the state of the art of CFI solutions. Chapter 4 provides background information and terminology, needed for understanding the work of the tool. In addition it describes the solution presented in [30], the tool was integrated in. In Chapter 5, the architecture of the Python tool and how the activities are carried out are discussed. Chapter 6 illustrates the results obtained by running the tool on a given set of benchmarks, and finally, Chapter 7 concludes the work and offers suggestions for future improvements.

Chapter 2

Background

In the present Chapter, the main *memory corruption vulnerabilities* that may affect a software application are presented, and how they can be exploited by a threat actor in order to possibly obtain arbitrary code execution on the victim system. Finally, the possible *software protection techniques* that can be employed in order to counter such kind of attacks are discussed. Among them, Control-Flow Integrity is only briefly presented, while it is described more in the depth in the next Chapter.

2.1 Memory Corruption Vulnerabilities

C/C++ languages are usually the preferred languages for writing software applications for embedded systems. C/C++ provide powerful abstractions to the developer for manipulating hardware resources like memory, as developer wants to gain higher performances. For these reasons, C/C++ entails all memory management to the developer.

Most of the times, vulnerabilities are introduced due to incorrect memory management by the developer, that may be exploited by an external threat actor in order to leak data from the running application or overwrite critical data structure in the application, that can alter the program behaviour. C/C++ provide pointers to manipulate memory with no restrictions. if the developer does not follow good programming practices when writing code, he may unwillingly insert a vulnerability.

Some of the most known types of vulnerabilities are:

- **Buffer Overflow** [13][14]: vulnerability where a program when writing data in an array overruns the boundaries of the array, and as a consequence overwrites adjacent memory location.
- **Externally-Controlled Format String** [16]: present when the format string is controlled by the malicious user. It leads to format string attacks. A vulnerability that stems from the usage non-sanitized user input as format string parameter in C functions in charge of formatting, for example `printf`.
- **Buffer Over-read** [15]: vulnerability where a program when reading data from an array overruns the boundaries of the array, and as a consequence reads adjacent memory location.

- **Memory leakage** [17]: it refers to multiple allocation on the heap without any release, with a consequent heap's saturation. The heap saturation causes the system to not work anymore properly, since no more memory is available.
- **Use-after-free** [18]: Present when the memory pointed by a pointer is freed, but the pointer is not sanitized (i.e., set to `NULL`) but still references the freed memory location.

The exploitation of this vulnerabilities may lead to program crashes or to unexpected behaviour in the program. But typically the threat actor exploits them with the aim of obtaining arbitrary code execution. It means that attacker is able to inject arbitrary code, in the memory reserved to the program, and hijack the execution flow of the program to the location of the inject code.

In the next Section, it is shown how a buffer overflow vulnerability can be exploited by an attacker, and the corresponding countermeasures will be described.

2.2 Stack-Based Buffer Overflow

Stack-based Buffer overflow corresponds to a buffer overflow vulnerability affecting the the memory area reserved to the stack. As anticipated, it is present when a program writes to a memory address on the program stack, outside of the data structure pointed to by the memory address that corresponds to a fixed-length buffer.

Let us consider the following simple C program as an example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

        char buffer[30];

        strcpy(buffer, argv[1]); //here happens the overflow
        printf("buffer content= %s\n", buffer);

        printf("strcpy() executed...\n");

        return 0;
}
```

This simple program takes as input a string from the user and copy the string inside a fixed-length buffer wide 30 bytes. The variable `buffer` is our vulnerable buffer. At execution time, `buffer` is inside the stack frame of the `main()` function. In case, the user may provide a string larger than 30 bytes: this will cause overwriting data inside the stack after the vulnerable buffer.

A critical structure located inside the stack is the *return address*. If the return address is overwritten, it is possible to redirect the execution flow to an arbitrary memory address, as can be seen in Figure 2.1. The practice of voluntarily overwriting data inside the stack is also referred to as *stack smashing*. With this technique, an attacker can also place in the stack arbitrary code, and overwrite the return address with the memory pointing to the injected code (Figure 2.2), thus obtaining *arbitrary code execution* on the victim system.

This happens because when the epilogue of the function `main()` is reached, the `RET` instruction is executed, and the corrupted return address is loaded into the *instruction pointer*, also referred to as *program counter*, containing the current instruction to be executed by the CPU.

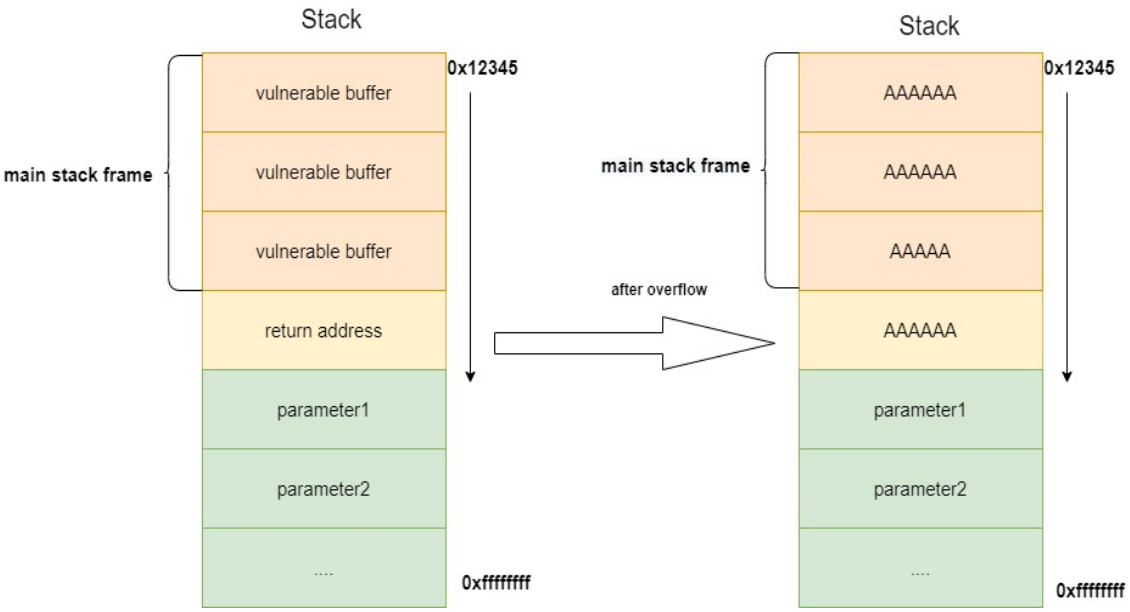


Figure 2.1: Stack smashing overwriting return address with 4 **A's**.

The injected code is usually referred to as *shellcode*, i.e., a sequence of instruction that allows to "*get a shell*" to issue arbitrary commands on the victim system. In other cases the shellcode, might perform more complex operations like connecting to a *Command and Control Framework*, also called *C2 Framework* from which the threat actor can remotely control the victim system (Figure 2.3).

As can be seen, without any software protection a memory corruption vulnerability as a stack-based buffer overflow can be trivially exploited by a threat actor. It is worth to notice how this technique relies on the fact that the stack allows to store code and execute it. This means that pages composing the stack are both writable and executable. For this reason, the first software protection technique that was introduced to counter this kind of threat is *Data Execution Prevention* (DEP).

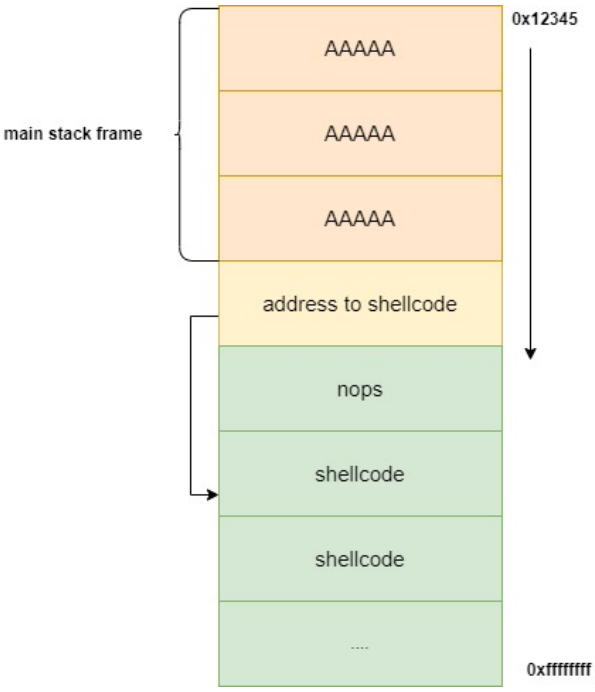


Figure 2.2: Stack smashing overwriting return address with address pointing to **shellcode**.

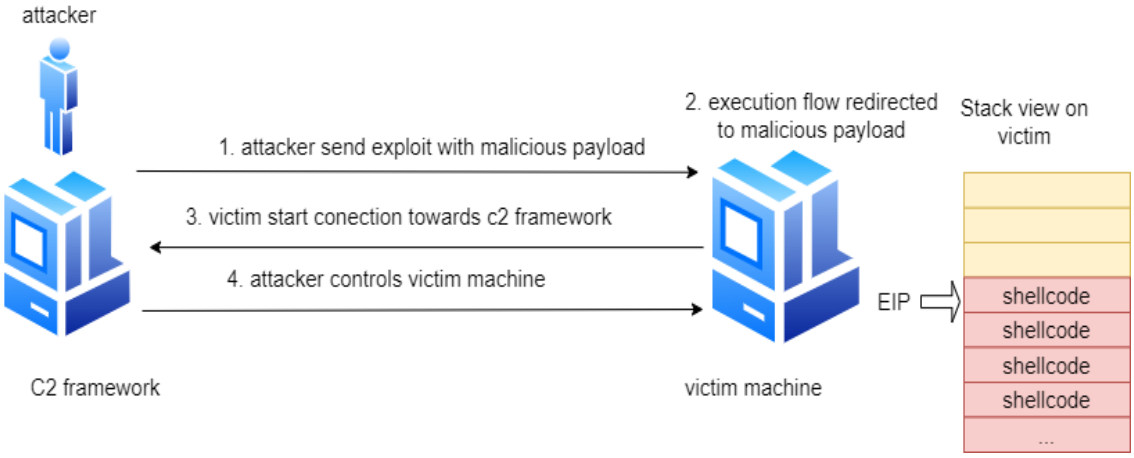


Figure 2.3: Attacker controls victim after exploiting memory corruption vulnerability.

2.3 Data Execution Prevention

Data Execution Prevention [38], also referred to as *Write Xor Execute*, is a software protection technique based on memory pages permissions. A memory page can be *Writable* (**W**), meaning that it is possible to write data on it, *Readable* (**R**), meaning that it is possible to read data from it, or *Executable* (**E**), meaning that it is possible to execute instructions stored in it.

A memory page can have a combination of this permissions: for example, it can be both writable and readable (**WR**), or both readable and executable (**RX**). With DEP, it is guaranteed that a memory page can not be at the same time writable and executable (**WX**). Therefore, with DEP, all memory pages reserved to the *stack* are marked as **WR**, writable and readable, but **not executable**. The same of course applies also to pages of other memory regions used for writing data, like the *heap*.

When there is an attempt of executing code in the stack (as was done with the shellcode in Figure 2.2) an exception is thrown, since the memory pages are not executable (Figure 2.4). In desktop application running on top of an operating system, DEP policy is guaranteed by the operating system.

For embedded systems, Data Execution Prevention policy can be implemented by a *Memory Protection Unit* (MPU) as described in [37]. For every access request to a memory page, the MPU first checks permissions against type of access, and the access request is possibly not permitted, with a protection fault triggered.

It is important to notice that the Data Execution Prevention policy only prevents the attacker from executing arbitrary code injected in the program address space. In case the attacker is able to take control of the instruction pointer, for example by exploiting buffer overflow vulnerability as described in Section 2.2, it is still possible for the malicious actor to hijack the execution flow to existent pieces of code already present in the program (since these pages will be marked as executable). Upon this concept, new attack techniques were developed, called *Code-Reuse Attack* techniques.

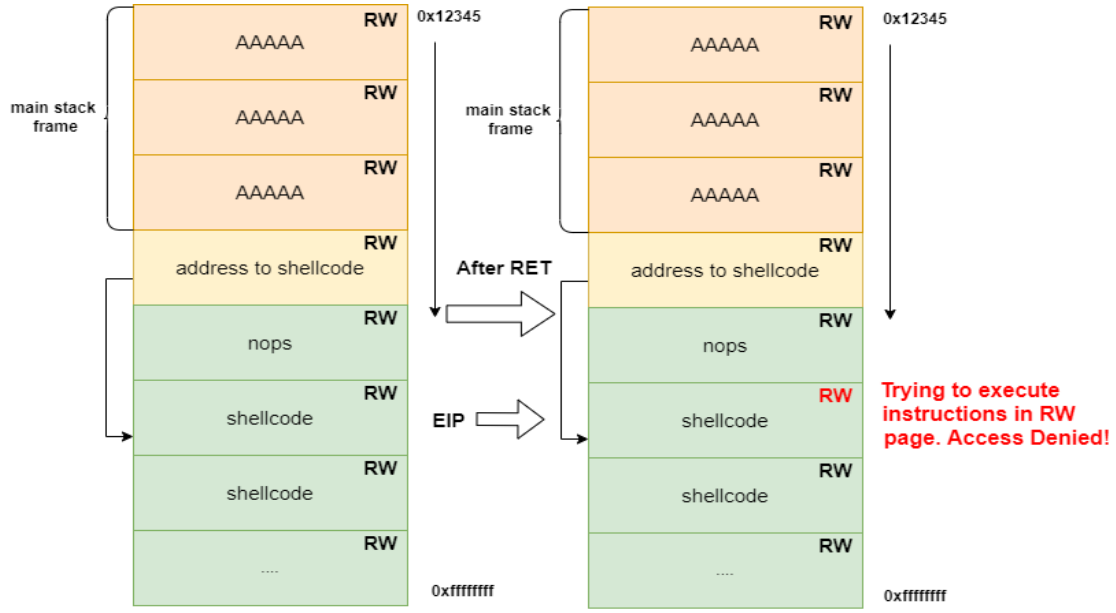


Figure 2.4: DEP prevents execution of shellcode in stack.

2.4 Code-Reuse Attack

When memory-protection defense techniques were introduced, attackers found the way to still been able to obtain ACE *arbitrary code execution* by *reusing the code already available* in the victim program instead of injecting custom code. In this way, a memory-based protection technique as Data Execution Prevention is ineffective, since the code executed is all contained in valid memory pages (according to the DEP policy, in executable pages).

This advanced exploit model is called *Code-Reuse Attack* (CRA), and consists of locating small pieces of code, referred to as *gadgets*, ending with a control-flow transfer instruction, like a jump, call or return instruction. When executed, a gadget redirects the execution to the next gadget in the so-called *gadget chain*.

As previously said, an attacker is first required to find an exploitable vulnerability that allows him/her to inject the gadget chain and to hijack the execution flow to the first gadget in the chain. The attacker can locate gadgets either in the code of the program, or in the code of external libraries the program is linked to. The attacker might exploit the presence of external libraries in order to redirect the execution flow to an entire function in a library, that may allow him to obtain arbitrary code execution more easily.

This is how *Ret-to-libc* attack work [26]: the attacker hijacks the execution flow in order to make the program execute functions inside the C Standard Library (`libc`). Typically the attacker redirects the execution flow to the `system()` function, that can execute *arbitrary system commands* passed as parameters. In Figure 2.5, it is shown how an attacker that, by exploiting a buffer overflow vulnerability, can use the *Ret-to-libc* technique in order to obtain *arbitrary command execution* on the victim system. The return address is overwritten with the address of the `system()` function, and pointer to the string `"/bin/sh"` is passed as parameter in the stack.

When the `RET` instruction is executed by the epilogue the program, `system("/bin/sh")` is called. This causes the program to start the Unix shell `/bin/sh`, allowing the attacker to execute arbitrary shell commands on the victim system.

The first attack technique implementing the Code-Reuse attack paradigm is *Return-Oriented Programming* (ROP) [33]. Subsequently, more complex Code-Reuse attack techniques were introduced, like *Jump-Oriented Programming* (JOP) [7] or *Call-Oriented Programming* (COP) [34].

2.4.1 Return-Oriented Programming (ROP)

The most traditional version of CRAs is the *Retun-Oriented Programming* (ROP) [33]. In ROP, the attacker exploits a buffer overflow vulnerability and places in the stack, after the return address, a sequence of memory addresses each one pointing to a small piece of code, located in the program address space, ending with a `RET` instruction (Figure 2.6). These small pieces of code are also called *ROP gadgets* and are usually made of few instructions. Every gadget performs a specific task, such as read/write operation from/to memory, or arithmetic/logic operations between registers.

As can be seen in Figure 2.6, the return address is overwritten with the address pointing to the first address in the chain. When the epilogue of the vulnerable function is executed, the first `RET` instruction is executed, and the first gadget address is popped from the stack and loaded in the *instruction pointer*. The instructions of the first gadget are executed

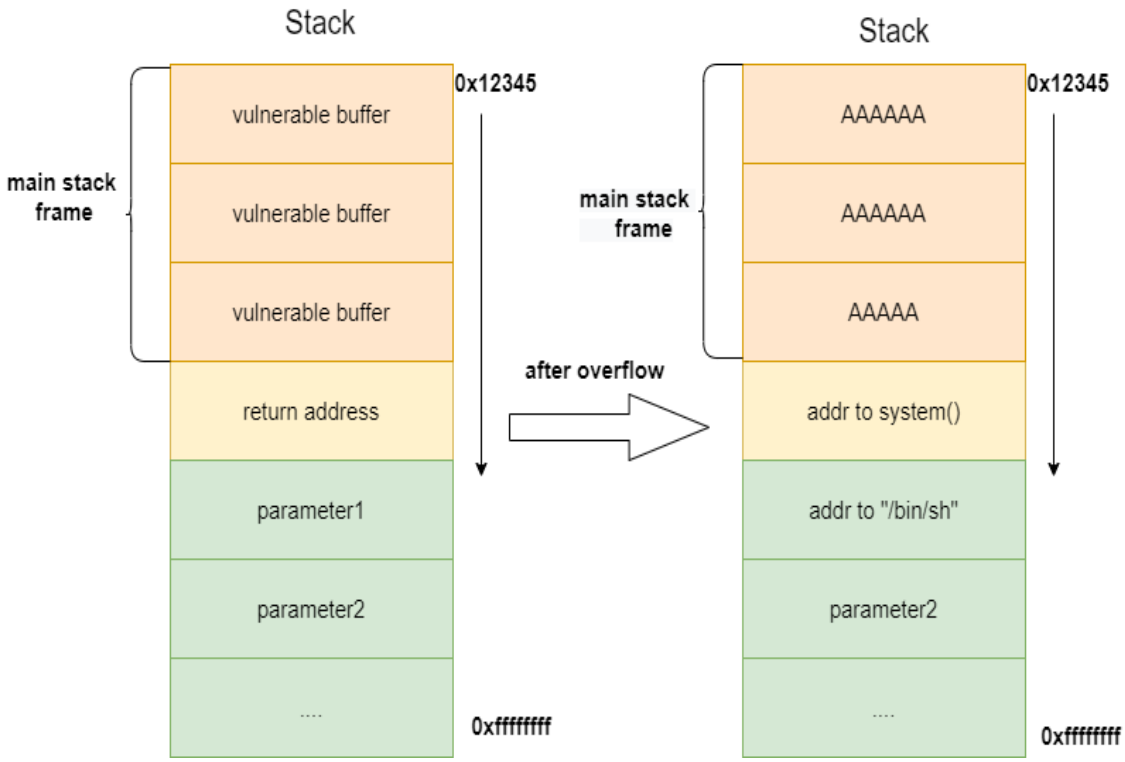


Figure 2.5: View of the stack after a Ret-to-libc attack

until final `RET` instruction in the gadget is executed that will pop the second gadget address from the stack in the *instruction pointer*. The execution continues in this way until all the gadgets in the chain are executed.

The objective of the attacker is to craft a gadget chain that, when executed as a whole, allows the attacker to execute arbitrary command on the target system. For example, the attacker may craft a *ROP chain* that will end up with a `syscall` instruction to the `execve` function (considering a Linux environment with a `x86_64` architecture): this leads the malicious actor to execute arbitrary shell commands on the target system.

2.4.2 Other CRA Techniques

As previously mentioned, ROP is not the only Code-Reuse Attack technique available. Another more powerful implementation (but also more complex) is the *Jump Oriented Programming* (JOP) [7]. Similarly to Return-Oriented Programming, the execution flow is redirected to a chain of small pieces code, but this time each one ends with an indirect `JMP` instruction instead of a `RET`.

These technique is more complex with respect to the return-based one, since `JMP` does not allow to naturally return control to the next gadget in the chain the same as with a `RET` by storing gadgets in the stack.

In order to solve the issue, a *dispatcher gadget* have to be found (increasing the complexity of the attack). Such a gadget will control the execution flow among the other

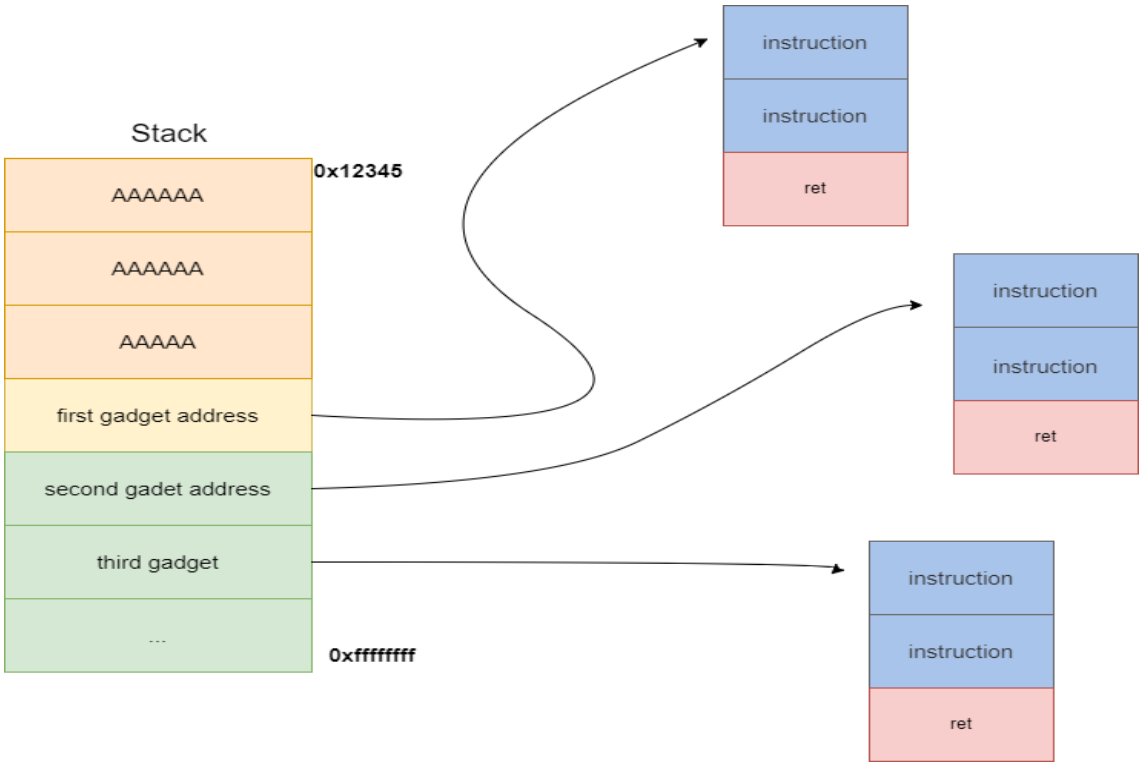


Figure 2.6: Stack view after a ROP-based attack.

gadgets (called *functional gadgets*) with the help of a *dispatch table* containing addresses to the functional gadgets. When the functional gadgets completes his execution, it will return control through a `JMP` to the dispatcher gadget, that will transfer control to the next functional gadget in the disaptch table (Figure 2.7). The execution continues in this manner until all functional gadgets are executed.

In conclusion, all these techniques are effective when the attacker is able to alter the execution flow of a program by initially corrupting the instruction pointer. In case it is prevented the attacker from overwriting the instruction pointer or it is just prevented from hijacking the execution flow such kind of attacks is mitigated.

2.5 Mitigation and Countermeasures

This Section discusses possible countermeasures to attack techniques described so far, highlighting advantages and drawbacks for each.

2.5.1 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a software protection technique based on the randomization of address space. This means that the base address of an executable, the start location of the heap, the stack, and the base address for shared libraries loaded in the

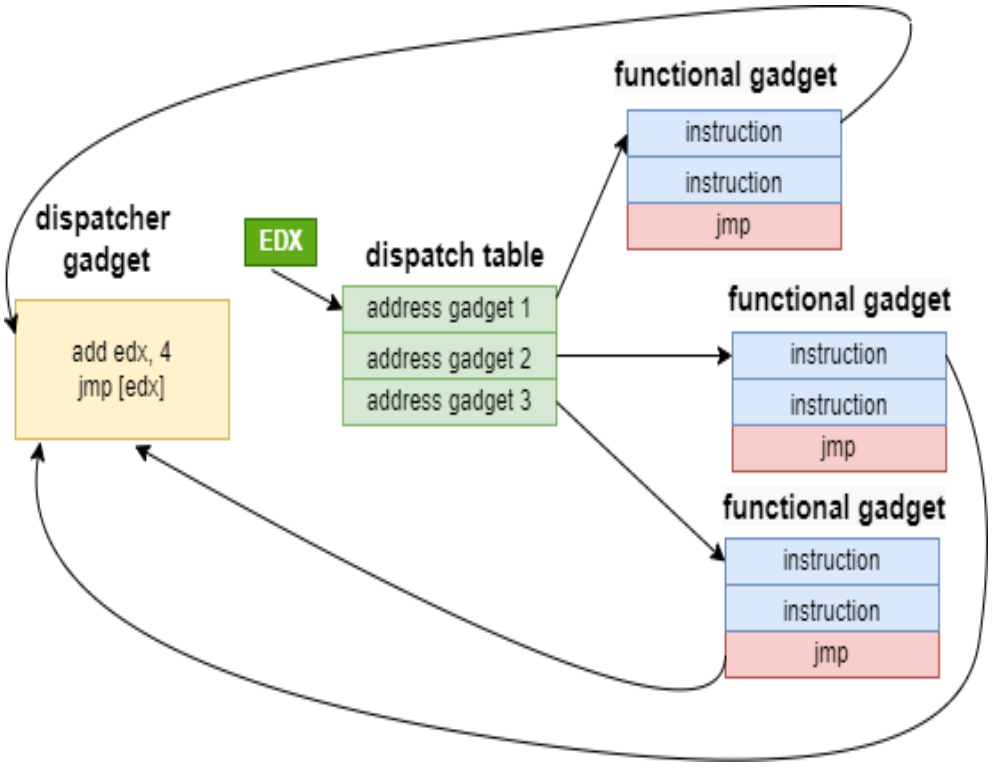


Figure 2.7: Schema of JOP-based attack.

process' address space are chosen randomly every time a process running the application is created [35].

In a 32-bit architecture, the attacker might still be able to brute-force the base address, since usually the lower 2 bytes will be set to 0 while change only the upper ones, by running the exploit multiple times until success. In a 64-bit system, a brute-force attempt would be too expensive, and it is considered very unlikely [35].

ASLR, in conjuction with DEP, proved to be effective in order to prevent the attacks previously described, since the Code-Reuse Attack paradigm relies portion of code (gadgets) in the program address space that needs to be located at a fixed/predictable location [35].

However, in the case the attacker is able, maybe by exploiting another vulnerability, to leak the base address of a library linked to the program, like the C standard library (libc), *at runtime*, CRA techniques are again available to the attacker, like *ret-to-libc*.

2.5.2 Stack Canaries

The stack canary protection technique [12] is particularly effective in order to protect the application from attacks targeting the program's stack. In particular, stack canaries aim at protecting the return address stored in the stack from been overwritten.

When a function is called, a randomly-generated value is inserted between the return address and the stack frame of the called function. Before the function returns, so before the

RET instruction is executed, the program checks the value of the canary in the stack against the original value. If there is a match, the execution continues successfully; otherwise, an exception is thrown, and the program execution is interrupted (Figure 2.8).

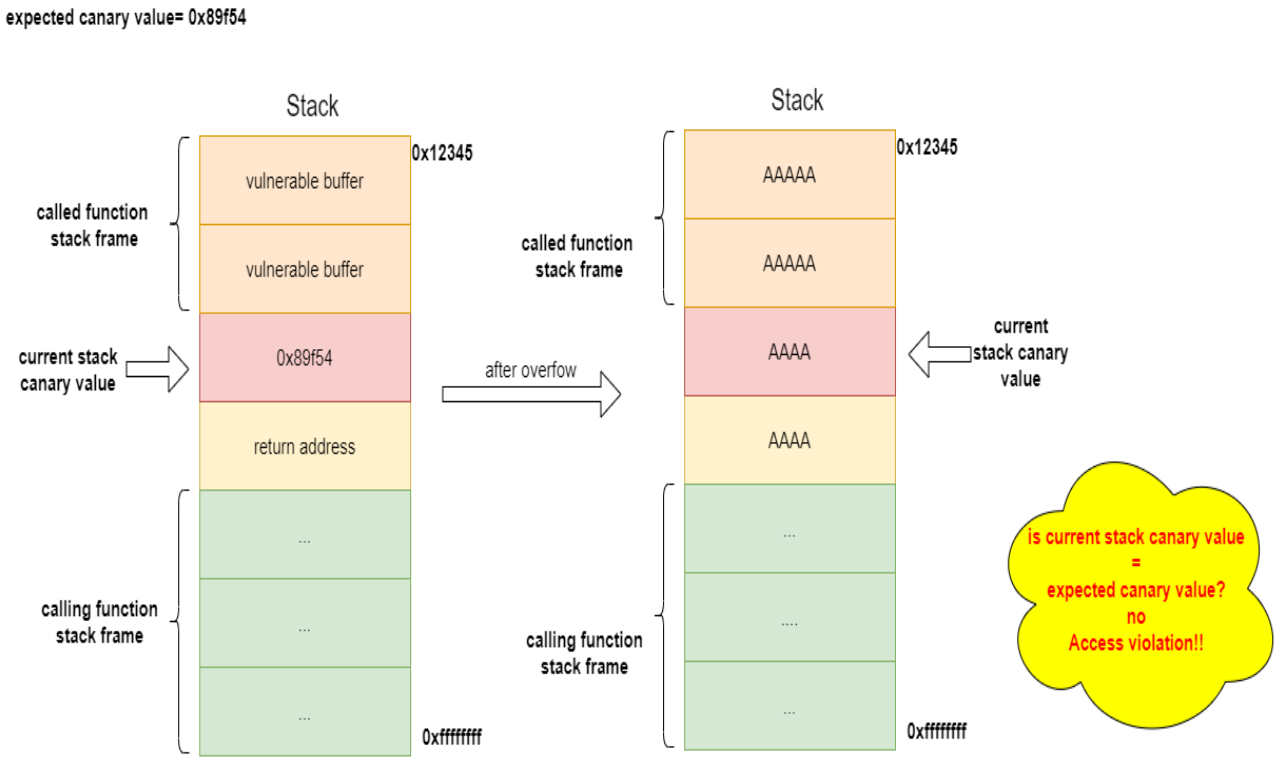


Figure 2.8: Stack canary preventing stack-based buffer overflow.

In order to bypass a stack canary, an attacker must be able to first leak the canary value at runtime, insert it in his/her malicious buffer and finally send it to the vulnerable application.

2.5.3 Control-Flow Integrity (CFI)

Control-Flow Integrity (CFI) is the most powerful protection concept for code hijacking attacks. The solution has been first proposed in 2005 by Abadi *et al.* [2].

To be applied, CFI requires to first obtain a map of all the valid destinations for each control-flow transfer instruction, like return, call or jump instruction. Once the map has been obtained, the program needs to be instrumented in order to verify, at runtime, that each control-flow instruction will transfer the execution flow to a valid destination among the ones in the map previously obtained.

In other words, the integrity of the execution flow is checked by comparing the current program path against a pre-computed model, encapsulating information about all the allowed execution paths in the program.

This model corresponds to the *Control-Flow Graph*, a higher level representation of the program extracted with the adoption of program analysis techniques.

The Control-Flow Graph is a directed graph made of:

- ***Nodes***: a node identifies a *basic block* (BB), where a basic block corresponds to a set of non-jumping instructions ending with a control-flow transfer instruction (conditional/unconditional jump, call, return,...). It is important to highlight a basic block has no other branch instruction than the ending one (Figure 2.9).
- ***Edges***: an edge corresponds to a control flow-transfer instruction and links two basic blocks. The edge corresponds to control-flow transfer instruction that terminates a *source basic block* and redirects the execution to a *destination basic block* (Figure 2.10).

Figure 2.9: Example of valid and non valid basic blocks.

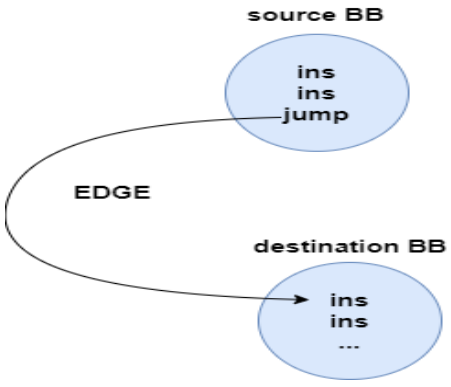


Figure 2.10: Example of edge.

These kind of checks can be either implemented by the program itself (in software) or by an hardware module external to the application. In both cases, the original software application likely needs to be instrumented in order to protect the code location at risk of control-flow hijacking attempts.

Drawbacks of Control-Flow Integrity are the complexity in the implementation and overhead that may cause on the application in terms of memory costs and performances.

Chapter 3

State of the Art

The solution aiming at enforcing *Control-Flow Integrity* property of a program consists of verifying, at execution time, that the current execution flow is compliant with the *Control-Flow Graph* of the program, previously extracted through static analysis. In the current Chapter, first the two main types of binary instrumentation are described, and existing tools performing the instrumentation will be briefly described. Later, an overview of the state of the art about Control-Flow Integrity techniques and methods is provided, presenting what are the main software-based and hardware-based solutions.

3.1 Static and Dynamic Binary Instrumentation

Control-Flow Integrity solutions guarantees the integrity of execution flow by comparing the checking the current execution path against a *pre-computed* model, encapsulating all the valid execution paths. In order to enforce the correctness of the execution flow, solutions may require to insert additional instruction, in suitable places, in the original program, in order to instrument the application to check that the currrent execution path is compliant with the valid model. The instrumentation code aims at protecting *weak* points in the program that may be exploited by attackers in order to trigger Code-Reuse Attacks.

For this reason, some of the existing CFI solutions rely on the following activities:

1. the Control-Flow Graph analysis, that allows to extract a map with all the valid execution paths that a program can follow;

2. the insertion of proper instructions in the program, e.g., before and after the execution flow changes, that check if the current path followed by the program is still in the boundaries of the map extracted at the previous stage.

The process of analyzing and instrumenting a program is called *binary instrumentation*. There exist two alternatives:

- **Dynamic Binary Instrumentation** (DBI) [27]
- **Static Binary Instrumentation** (SBI) [28]

Static Binary Instrumentation consists of analyzing the program *ahead of execution time*, by first disassembling the code and later detect the point in the program requiring instrumentation [43]. These *instrumentation points* will be used as locations for inserting the instrumentation code in the executable file. (Figure 3.1).

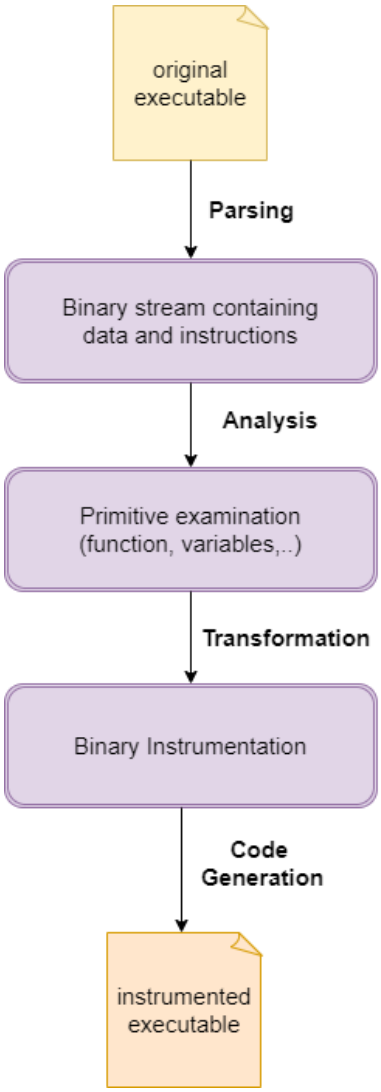


Figure 3.1: Static binary instrumentation process

On the other hand, Dynamic Binary Instrumentation relies on an external monitor that monitors the program at *execution time*. The monitor inspects one instruction at a time before being executed, and inserts the instrumentation code *on-the-fly*.

The main advantages of DBI are: (i) overhead in terms of memory costs is avoided, since the original executable is kept unmodified, and (ii) instrumentation is performed at execution time, without the need of an expensive offline process. In addition, the real-time approach adopted by DBI does not have to face the inaccuracies of disassembly engines, and does not have to care about code relocation, which have to be addressed by SBI during

the *Transformation* phase.

DBI has instead its drawbacks in the overhead introduced in terms of execution time, that may significantly affect performances. This is a particularly critical feature in real-time applications such as in the embedded domain. SBI still introduces an overhead in terms of performances, but it is only given by the additional instrumentation instructions that are inserted, that is minimal with respect to what happens with DBI, where an *execution by symbols* take place.

With both strategies, the instrumentation process follows to the following stages [42]:

1. **Parsing**: the purpose of this stage is to obtain the raw instruction stream from the executable and then send it as input to a disassembler. In addition, the content of global variables and data section are collected for further analysis;
2. **Analysis**: at this stage, the structure of the program is retrieved. The instruction stream is disassembled, and disassambled instructions are grouped in functions. In addition, branch instructions are detected, and a *Control-Flow Graph* (CFG) is produced;
3. **Transformation**: Once the CFG is available, the *instrumentation points* in the code are collected, and are augmented with instrumentation code;
4. **Code Generation**: it is the last stage of the process, where the modifications are integrated in the program in way that a final executable is produced;

3.1.1 Binary Instrumentation Tools

In literature, several binary instrumentation tools have been presented, based on advantages and disadvantages of the instrumentation technique. In the present Section, some of the most common tools following either SBI or DBI will be briefly presented.

PEBIL

PEBIL [28] (*PMaC's Efficient Binary Instrumentation Toolkit for Linux*) is a binary instrumentation toolkit that enables the building of instrumentation tools producing an efficient instrumented executables, leveraging the static approach.

PEBIL instruments the executable statically by placing a branch instruction at the instrumentation point that will redirect the execution to the instrumentation code. The instrumentation code saves the program state, and will perform the actions required by the instrumentation tool.

In order to support ISAs with variable length instructions, PEBIL takes care of relocating and transforming code for each function in order to make instrumentation code reachable from the instrumentation points.

PEBIL supports only x86 architecture.

PIN

Pin [4] as a dynamic binary instrumentation framework developed by Intel. Pin aims at providing a platform for building program analysis tools, called *pintools*. A pintool consists of three type of routines:

- **instrumentation routines**: routines that inspects the application's instructions and insert calls to **analysis routines**;
- **analysis routines**: invoked when the program executes an instrumented instruction;
- **callback routines**: called when an event occurs;

The C/C++ API's provided by PIN can be leveraged to build pintools. PIN will use a *Just-In-Time* (JIT) compiler in order to apply instrumentation at *runtime* to the application.

Additionaly, PIN supports the *probe mode* option, which provides higher performances but has a limited set of callbacks available limiting the capabilities of the pintool.

PIN is provided for Windows and Linux platforms, but works only for program compiled for the Intel architecture, making it not suitable for embedded systems based on ARM architectures.

Dynist

Dynist [5] is a binary instrumentation and analysis framework leveraging both the static and dynamic approach for code instrumentation. Dynist provides a representation of the program in terms more higher level structures, like function's basic blocks, and the user leverages this representation in order to instrument code in *anywhere* in the binary.

Dynist emphasize *anytime* instrumentation by providing the possibility either to statically instrumented the code (binary permanently modified) or instrument it at execution time (dynamic instrumentation). The user is allowed to modify or remove instrumentation at anytime.

In addition, the analysis techniques performed allows to insert new instruction without influencing no-instrumented code, that can run at native speed, in order to minimize the overhead in performances.

3.2 Software-based Solutions

Software-based Control-Flow Integrity solutions rely on control-flow monitors that are implemented using software techniques only. In other words, checks are performed by additional code, running in parallel through another process or inside the program itself, through binary instrumentation.

In the original article describing CFI [1], authors propose a *label-based instrumentation* to have the software checking the control flow validity by itself. The approach relies on storing unique identifiers (*labels*) at the beginning of each *basic block*, where a basic block in a CFG corresponds to a set of instructions with no branch instructions in between. The instrumentation code is inserted before the indirect branch, and checks whether the destination basic block ID matches with the expected one retrieved from the CFG analysis. In case a malicious actor tries to hijack the execution flow, the attempt is detected and integrity of the execution flow is guaranteed.

In Figure 3.2, it is shown an example of how the mechanism works. In this case, the destination of the jump was labelled with ID 12345678. Before the `JMP ECX` is performed, the target label is checked in order to detect control-flow tampering attempts. In the

example if the check fails, the execution is redirected to the `violation()` function, that will stop the execution; otherwise, `ECX` is incremented and the execution is successfully redirected to the proper destination.

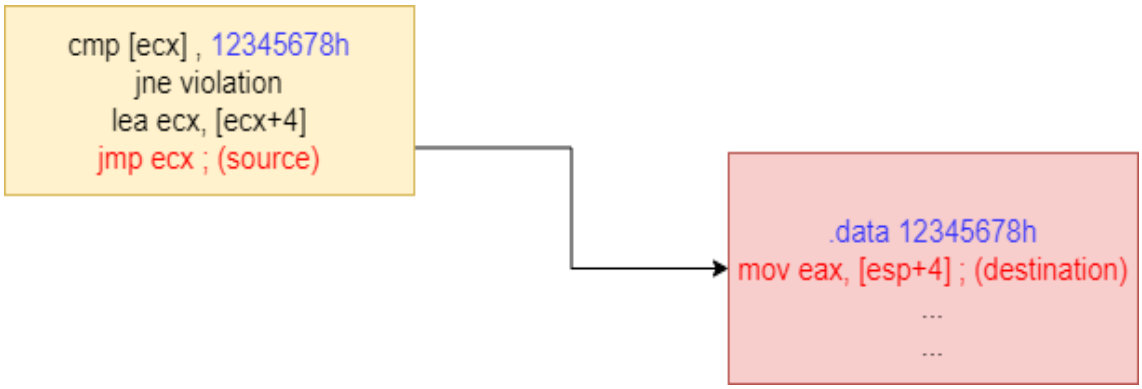


Figure 3.2: Label-based instrumentation code

Control-Flow Locking (CFL) [6] was introduced as an alternative to the label-based implementation in order to reduce the overhead introduced by the previous solution. Control-Flow locks are similar to *mutex* locks, but they are instead used to enforce validity of the application execution flow.

Some *lock code* is placed before each indirect control-flow transfer instruction. This lock code, will asserts a lock by simply changing the *lock value*, a value in memory.

The corresponding *unlock code* is placed at each destination of the control-flow transfer instruction, where the unlock code will de-assert the lock only if the current value of the lock is a valid one. In case the current lock value is different from the expected one, which implies that branch performed is not a valid. This behaviour allows to detect control-flow tampering attempts.

The unlock code must also set the lock value to an “initialization value", notifying to the next holder of the lock, that it can be acquired.

In Figure 3.3 is provided an example of application of Control-Flow Locking. First, `func1()` is called. Before returning at the end of the function, the lock code is executed. This snippet of code first checks if the lock was not set. If it was already set, this reveals a control-flow hijacking attempt, and the flow jumps to `violation()` function. Otherwise, the lock is set to a known value, (the value correspond to `key` in the example), and the execution returns to the caller. The caller, upon return, checks if the lock contains the correct `key` value. If the check is successful, the lock is unlocked (it is set to 0), and the execution proceeds normally; otherwise, it jumps again to `violation()`.

The main disadvantage of enforcing Control-Flow Integrity with software-based solutions is that requires to add to the program a large amount of instructions that introduces a significant overhead in terms of occupied memory and performances. This is particularly critical for systems with limited resources as embedded systems.

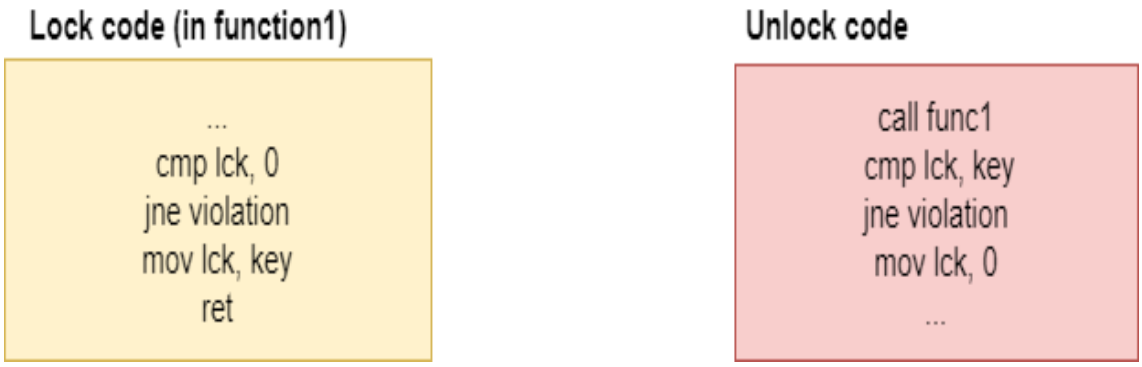


Figure 3.3: Label-based instrumentation code

3.3 Hardware-based Solutions

Hardware-assisted CFI monitor allows to avoid the losses in performances that are introduced by instrumentation routines at the software level. In order to guarantee the correctness of the control flow with hardware solutions, several challenges needs to be faced, in order to limit costs and ensure at the same time that the CFI principle is guaranteed.

The processor architecture must either modified or enriched with external modules, that are able to support CFI capabilities, increasing production costs. On the other hand, the solution is much faster in checking CFI, so that the impact on performances is negligible with respect to software-based solutions.

3.3.1 Branch Target or Instruction Encryption

One method for guaranteeing integrity of the control flow is to encrypt either the target address of a branch instruction or the target instruction itself.

The first method was proposed in [32]. It consists of inserting an additional component in the processor architecture which, before a function call, encrypts the return address before putting it onto the stack. At `RET` time, the encrypted return address gets popped from the stack, gets decrypted from the newly introduced module, and finally the address is loaded in the instruction pointer. If there was an attempt of tampering the control flow by overwriting the return address, the decryption will return an invalid address and the execution is stopped, successfully detecting the control-flow hijacking attempt. It is important to highlight that this solution does not protect indirect branches.

Slightly different technique was presented in [29]. In this case, the destination instructions of indirect branches are encrypted at *load time*, i.e., before the application is actually running. Everytime a branch instruction is executed, first the target instruction is decrypted and then it is executed by the processor. If the decryption procedure produces an invalid instruction, it means that there was an attempt of tampering the control-flow, and the execution is stopped.

The module in charge of the encryption/decryption process is joined with the help of *Physical Unclonable Function* (PUF) [25] or the *Advanced Encryption Standard* (AES) [19] algorithm. A solution adopting PUF allows to obtain better performances but suffers from

the cryptographic point of view, since the encryption/decryption procedure only consists of a XOR operation. Solutions using AES algorithm slightly decreases performances but are more resistant from a cryptographic point of view.

3.3.2 Shadow Call Stack (SCS)

Shadow Call Stack software protection, initially proposed in this paper with the name *SmashGuard* [31].

The technique requires to introduce an additional hardware module in the processor architecture, a new *hardware stack*, in which return addresses of function calls are pushed to and popped from.

Everytime a function call is performed, the return address and the current stack pointer will be both saved in the software stack (the usual memory area available allocated in the process address space), and in the newly hardware stack added to the CPU. When a return instruction is executed, both return addresses, the one in the software stack and the other in the hardware stack, are popped from the top of their stacks and are compared. If the popped addresses are equal, the execution can continue successfully; otherwise, a hardware exception is raised, detecting any attempt of hijacking the execution flow by return address corruption [9] [8] [24].

Drawback of this solution is that it is only limited at detecting stack-based attacks. Malicious actions on other memory area, e.g. heap or global data, remain undetected. In addition, inserting a module inside the processor is a very expensive solution, since requires to modify the processor architecture, even though no modification may be required to the *Instruction Set Architecture* (ISA).

3.3.3 Basic Block Hashing

Basic Block Hashing is another CFI enforcement technique that verifies the correctness of the execution flow by computing a hash of the basic block, and comparing the result with a pre-computed value. A mismatch reveals that the integrity of the control-flow was violated, and therefore the execution is interrupted.

The hardware module implementing the checks may be either strictly bounded to the processor [41], or placed between the instruction cache and the processor [10] [20], being also known as *Control-Flow Integrity Cache*.

3.3.4 ISA extensions

Another possibility to enforce Control-Flow Integrity is by extending the existing *Instruction Set architecture* (ISA) with additional instructions that perform CFI enforcement.

In [21], the authors propose HAFIX (*Hardware-Assisted Flow Integrity eXtension*), an extension of the base Instruction Set of Intel and Sparc embedded system architectures, defining additional instruction for protecting against attacks trying to break the integrity of returns from function calls.

HCFI (*Hardware-enforced Control-Flow Integrity*) [11] instead relies again on extending the ISA but also on introducing a shadow stack. HCFI improves HAFIX, as it allows to

guarantee the integrity also of indirect branches/function calls (forward edges) in addition to backward edges.

Chapter 4

Instrumentation Tool - Theory

The present Chapter is committed to provide the theory which is behind the instrumentation tool “PROLEPSIS" implemented during the work of this thesis.

The first Section aims at providing background information and the necessary terminology that is required in order to better understand the work. The next Section describes the general protection strategy and the rules of the monitor that is in charge of actually enforcing Control-Flow Integrity. In the final Section are discussed two example implementations of the CFI monitor: one as an external reconfigurable hardware module (FPGA) [30], and another one as a security IP module integrated in the processor.

4.1 Basic Definitions and Edge Classification

As already anticipated in Chapter 2, in order to guarantee the integrity of the execution flow it is required to extract and analyze the Control-Flow Graph, a directed graph made of nodes and edges, with nodes representing basic blocks and edges control-flow transfer instructions, transferring the execution flow from one basic block to another one.

Depending on the entity of control-flow transfer instruction, represented by the edge, two different kind of edges are distinguished [1]:

- ***Forward edge***: in case the control-flow transfer instruction corresponds either to a branch in the same function (*conditional or unconditional branches*) or to a function call (*call instruction*), the edge is categorized as a **forward edge**;
- ***Backward edge***: in case the control-flow transfer instruction corresponds to a return from a function call (*return instruction*), the edge is classified as a **backward edge**.

Edges can be further classified according to the source from which the target address of the branch is retrieved. In particular, an edge can be:

- ***Direct edge***: an edge identifying a jump instruction which target is constant and encoded in the jump instruction itself;
- ***Indirect edge***: an edge identifying a jump instruction which target is inside either a CPU register or a memory location.

As discussed in the previous Chapters, a malicious actor is able to tamper the execution flow by overwriting the content of memory/registers containing memory addresses to executable code. The target addresses of indirect jumps can be exploited in order to achieve such a condition. In the case of ARM-based and RISCV-based CPU architectures, the indirect jumps corresponds to that jump instructions getting the target address either from a register or from a memory location, which content may be corrupted by a threat actor.

With the retrieval of the Control-Flow Graph, it is possible to immediately retrieve the destination addresses of direct jumps, like `CALL` to a function or branches to labels, since the destination is constant and encoded in the instruction itself.

The same does not apply to indirect jump instructions, like `POP PC` or `BX lr` in the ARM instruction set or `RET` in the instruction set of RISC-V. In order to retrieve the target address of an indirect jump instruction it is necessary to retrieve the source operand from the instruction, and then trace back It's history until the target address is retrieved.

In order to achieve such a result it is necessary to retrieve the *origin-tree* of the source operand of the target. As mentioned here [30] the origin tree is a tree having as a root, the source operand of instruction generating the indirect edge, and as arms all the traces that allows to reconstruct the value stored in the source operand. In order to retrieve it, it is necessary to retrieve a slice of the program containing the instructions involved in defining the value of the source operand.

For executable running on embedded systems, it is always possible to retrieve the set of target addresses of an indirect jump instruction [30], since the executable is statically-linked to the necessary external libraries at compile time, therefore all the code is already available inside the executable file.

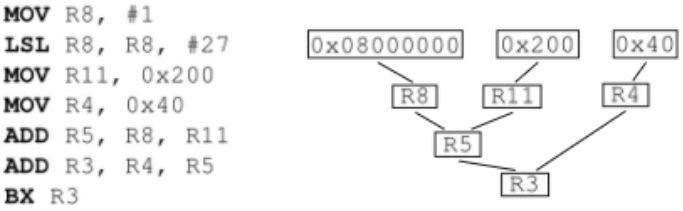


Figure 4.1: ARM code segment and origin tree of the register involved in the indirect jump [30].

For the purpose of protecting the integrity of the control-flow of the target executable, while minimazing the overhead in terms of memory costs, the edges of the CFG can be again distinguished in:

- ***Insecure edge***: an insecure edge corresponds to an edge that requires to be applied instrumentation. Insecure edges are **indirect jumps** which source operand's value depends on memory locations at risk of corruption, like the stack or the heap. An example is the `RET` instruction for RISC-V or `POP {PC}` for ARM;
- ***Secure edge***: a secure edge insteads corresponds to an edge that does not need

to get applied instrumentation. Secure edges are either **direct jumps** or **indirect jumps** which source operand's value does not depends on memory locations at risk of corruption, like the stack or the heap;

Another issue that has to be taken into account are the handling of Interrupt Requests. At execution time, the processor may receive an Interrupt Request, that will trigger the processor to transfer the execution to the corresponding *Interrupt Service Routine* (ISR) in order to serve the request. Interrupt Request are received asynchronously at anytime in the code, and can not be forecasted through Control-Flow Graph analysis.

When the processor needs to serve the Interrupt Request, first saves the current *program context* (current content of the *registers*) in memory, so that it can be restored when completes the ISR execution. Since ISRs may contain vulnerabilities exploitable by an attacker, it must be kept into account by the instrumentation process [30]. For this reason, in order to handle ISRs properly, it might be required to add instrumentation instructions both at the beginning and at the end of the routine.

4.2 Protection Mechanism

PROLEPSIS it is thought to instrument the program in a way that the CPU interacts with a *CFI monitor*, that is in charge of enforcing the Control-Flow Integrity policy.

It is possible to distinguish two stages in the protection mechanism: an *offline stage* and *online stage*. The offline stage is the one carried out by the instrumentation tool. The instrumentation tool analyzes the executable binary with binary analysis technique in order to extract the Control-Flow Graph, and then the Control-Flow Graph is further inspected in order to retrieve the insecure edges requiring instrumentation. The fundamental premise is that *it is always feasible to retrieve the destination of indirect edges and preserve the execution context when handling Interrupt Requests.*

Once are retrieved all the insecure edges, the tool generates a set of labels that uniquely identify each source/destination of every insecure edge.

For every identified insecure edge instrumentation code is inserted both before and after the jump instruction is executed. Therefore instructions are inserted *before the control-flow transfer instruction itself* and *before the first instruction pointed by the target address of the control-flow transfer instruction.*

The instrumentation instructions may consist of regular instructions already present in the Instruction Set Architecture or custom instructions (the choice depends on the CPU architecture, for example RISC-V allows to modify the toolchain and define custom instructions, while it is not the same for ARM). The instrumentation code provides the CFI monitor with the information about the current position in the program by sending the label, previously generated, identifying that code location.

Therefore, ahead of execution time, two operations must be performed:

- the monitor is provided with the list of accepted source-destination labels (identifying an insecure edge), and everytime it receives a pair, it checks if the pair is among the accepted ones embedded in it;
- the CPU is loaded with an instrumented version of the target executable, where the instrumentation instructions are executed in order to provide the monitor with the

pair source-destination identifying the current branch performed;

Subsequently, during the online stage (at runtime), the instrumented program, when an insecure branch is occuring, sends labels previously generated identifying the performed branch, by executing instrumentation code before and after the control-flow transfer instructions. The CFI monitor, when receiving information about the currently control-flow transfer operation, may detect a violation in case:

- the destination label is not valid or the pair source-destination labels received is not among the list of accepted pairs previously loaded;
- no label is received after the reception of a source label, meaning that the execution flow was redirected to a non-instrumented site that by construction can not be valid;

In case the Control-Flow Integrity monitor detects a violation, it may react in two distinct ways: stopping the execution or performing a corrective action.

4.3 CFI Monitor Implementations

As discussed in the previous Section 4.2, the instrumentation of the firmware consists of instructions sending to the monitor the information about the current location in the code before and after a branch is performed. These information consist of pairs of unique identifiers, also referred to as labels, that must be provided at load-time to the monitor in order to check the validity of the branch.

A possible implementation of the monitor is the one described in this paper [30], that consists of a reprogrammable hardware module (FPGA) interacting with the CPU with the purpose of enforcing integrity of the execution flow.

For this solution, the following types of instrumentation were identified:

1. *Backward insecure edge with a single target*: CPU transmits to the monitor the source ID before the branch and the destination ID immediately after the branch;
2. *Forward insecure edge with a single target*: same as previous one;
3. *Forward insecure edge with multiple targets*: as in 2., but inserting instrumentation code for all the destinations;
4. *Backward insecure edge with multiple targets*: as in 1., but inserting instrumentation code for all the destinations;
5. *Forward secure edge to a routine ending with a backward insecure edge with multiple targets*: it corresponds to a function call that does not require protection, since the edge is secure, but it is important to check that the return address is not modified. Therefore the CPU transmits to monitor the ID corresponding to the return address;
6. *Forward insecure edge to a routine ending with a backward insecure edge with single target*: a combination of 2 and 5. CPU sends both the ID of current location plus the ID of the return address;

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

For preserving program context, it is instead necessary to add instrumentation code at the beginning and at the end of every Interrupt Service Routine. The code for instrumenting edges according to the case and to preserve program context when entering/exiting into/from ISRs is shown in figures 4.2 and 4.3.

In the presented solution [30], the CPU is allowed to transfer to the FPGA:

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

The opcode identifies the type of operation to be performed by the FPGA, while the 16-bit value transmitted by the CPU can have one of the following meanings:

1. a unique ID identifying a position in the code. The 3 most-significant bits are set to 0 while the ID consists of the remaining 13 bits.
2. a 16-bit value representing half of the content of a 32-bit register. It is sent by the CPU when handling ISRs.

Another CFI monitor implementation is the one defined for RISC-V by the same research team of the previous solution. In this case, the monitor is implemented as an IP hardware component integrated in the processor. The IP hardware component consist of a *Programmable Logic Array* (PLA) [36]. The behavior is quite similar to the previous solution where the program, running on the CPU, is instrumented to send labels to the monitor along with opcodes. The main difference is in the instrumentation code.

RISC-V provides the possibility to extend the Instruction Set Architecture with additional instructions. For the solution the ISA was extended with four custom instructions:

- `CFLC`: **Control-Flow Load Configuration**. A series of this instructions is executed before the instrumented program is actually executed, in order to configure the PLA to interact properly with the processor;
- `CFES`: **Control-Flow Enforce Source**. This instruction must be executed *before a control-flow transfer instruction* of an insecure edge. It sends to the monitor the source label of the edge;
- `CFED`: **Control-Flow Enforce Destination**. This instruction must be executed *after a control-flow transfer instruction* of an insecure edge. It sends to the monitor the destination label of the edge;
- `CFPUSH`: **Control-Flow Push**. This instruction must be executed at entry point of an *Interrupt Service Routine*. It allows to store the value of registers directly inside a shadow register file inside the processor;
- `CFPOP`: **Control-Flow Pop**. This instruction must be executed at exit points of ISRs. It allows to restore the value of registers previously saved in the security module through the `CFPUSH`;

Both instructions `CFES` and `CFED` automatically take care of disabling interrupts before and after the information is sent to the monitor, allowing to reduce drastically the overhead in terms of memory cost, since now only one instruction have to be inserted.

Figure 4.4 summarizes how the instructions `CFES`, `CFED`, `CFPUSH` and `CFPOP` are inserted by the instrumentator in the program.

In this solution `CFES` and `CFED` take the format of a jump instruction, but have the purpose of sending a 20-bit value to the monitor.

The value is an ID identifying the location in the code that was protected, either a source or a destination of an insecure edge.

On the other hand `CFPUSH` and `CFPOP` have the same format of *store* or *load* instruction, already defined in the original Instruction Set Architecture of RISC-V, with a different behaviour: `CFPUSH` loads the register value both in the normal stack and in the *Secure Register Stack* located in the monitor (later described). `CFPOP` perform a load both from the *Secure Register Stack* and from the normal stack. It compares the values and, if they are different, raises an exception.

As for the FPGA-based solution, the opcodes specify to the monitor what operation has to perform.

The type of instrumentations identified are the same as the ones already listed for the ARM solution, but now all of them are handled in the same way, as summarized in Figure 4.4, where `CFES` is executed before the control-flow transfer instruction, while `CFED` is executed after the control-flow transfer instruction. Only exception are the handling of Interrupt Service Routines, that are handled using `CFPUSH` and `CFPOP` as anticipated.

In both solutions, the CFI monitor relies in three internal data structures in order to check the validity of the different kind of control-flow transfer instructions:

- a ***Secure Edge Table*** to store the list of valid source-destination pairs (source and destination corresponds to IDs, where an ID refer to a position in the code);
- a ***Secure ID Stack*** used as stack for *direct function calls*, to check the validity of the return address;
- a ***Secure Register Stack*** to manage registers content before and after the ISRs execution and preserve program state.

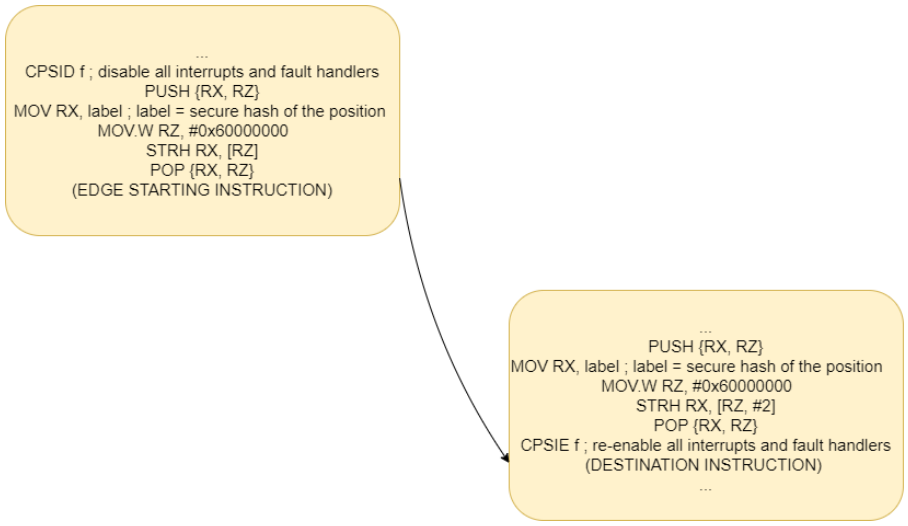
After the instrumentation process completes, at load time, the list of valid source-destination pairs will be converted in a bitstream and then "injected" in the monitor at load-time. At execution time, first *secure bootloader* is executed, taking care of configuring the monitor and setting up the interaction between the CPU and the monitor. Finally the bootloader, through a final jump instruction, redirect the execution to the entry point of the instrumented binary loaded in the CPU.

Therefore, at execution time, when the monitor receives both the source and destination IDs, it combines them through a XOR operation to index the *Secure Edge Table* containing the list valid source-destination pairs.

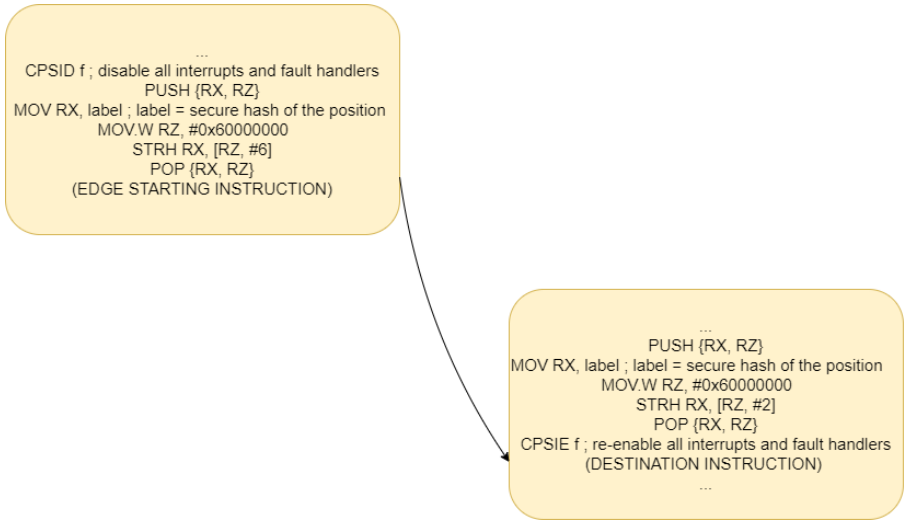
When the index points to a non-valid entry in the table, it means that a control-flow hijacking attempt was detected, so the monitor transmits an interrupt to the CPU in order to stop the execution. Otherwise, the execution continues transparently.

The check of the branch relies also on a *timer* inside the monitor, meaning that a destination ID, after the reception of the source ID, must be received before a timeout

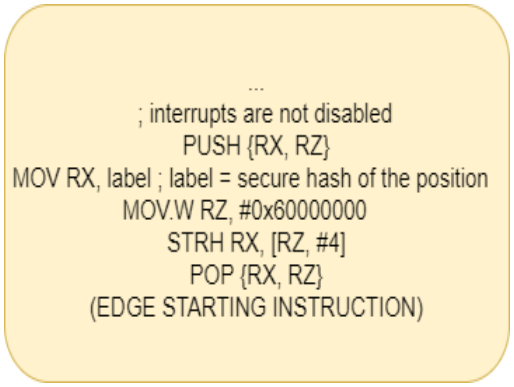
expires. If the destination ID is not received before the timeout, it implies the execution flow was redirected to a non-instrumented point in the code, that is invalid, so the CPU must be again signaled with an interrupt after timeout expiration.



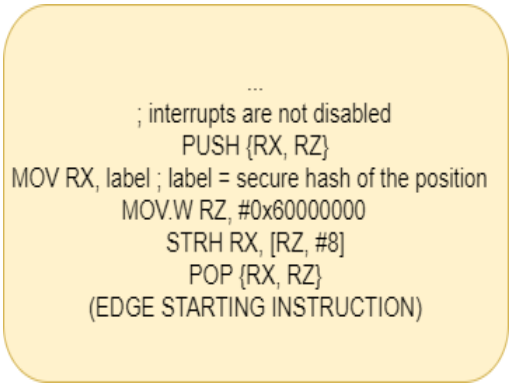
(a) ARM-FPGA instrumentation for types 1), 2), 3), 6).



(b) ARM-FPGA instrumentation for type 4).



(c) ARM-FPGA instrumentation for type 5).



(d) ARM-FPGA instrumentation for type 7).

Figure 4.2: Instrumentation code based on edge classification.

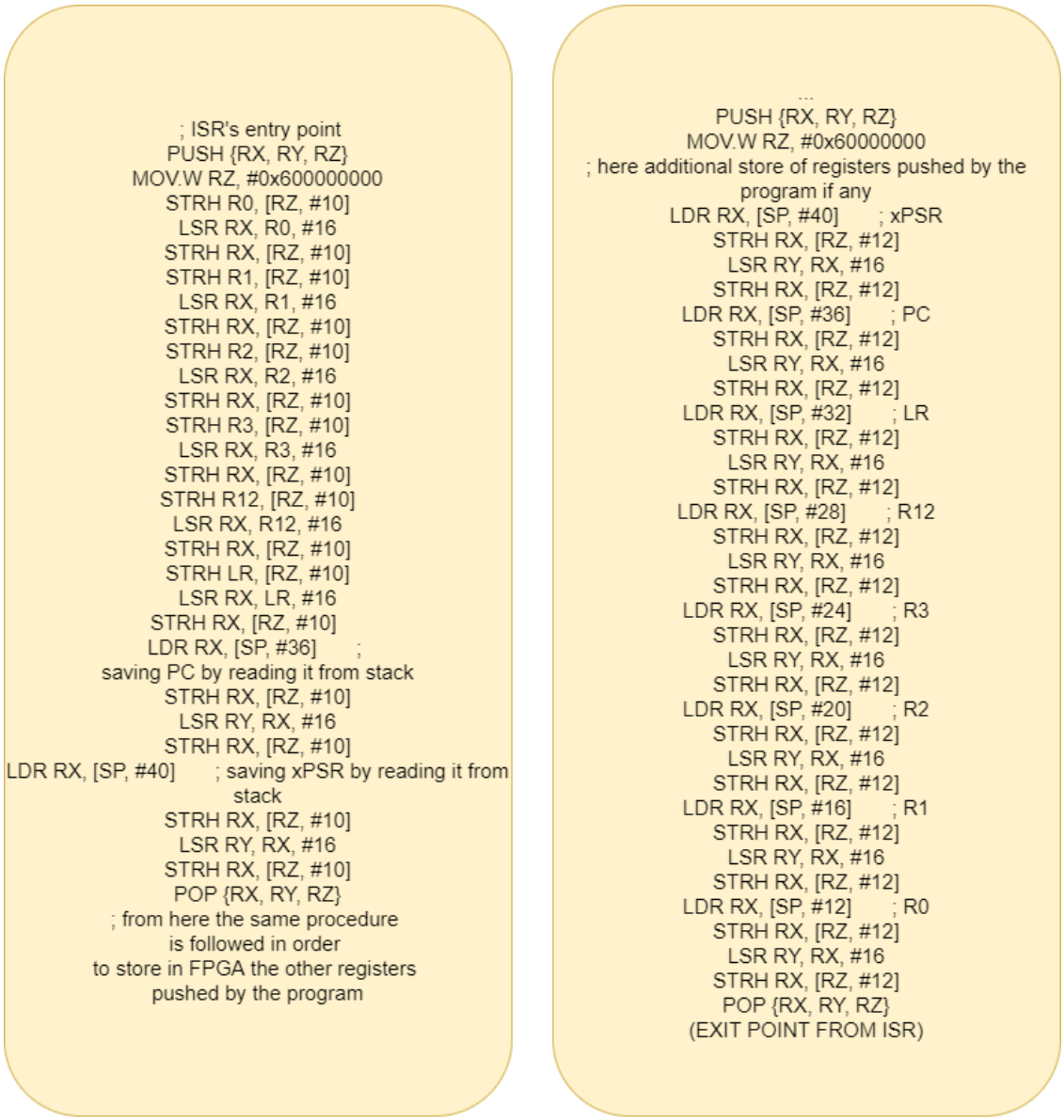
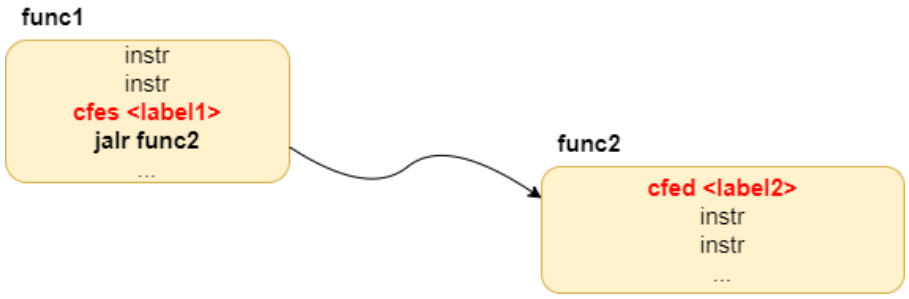
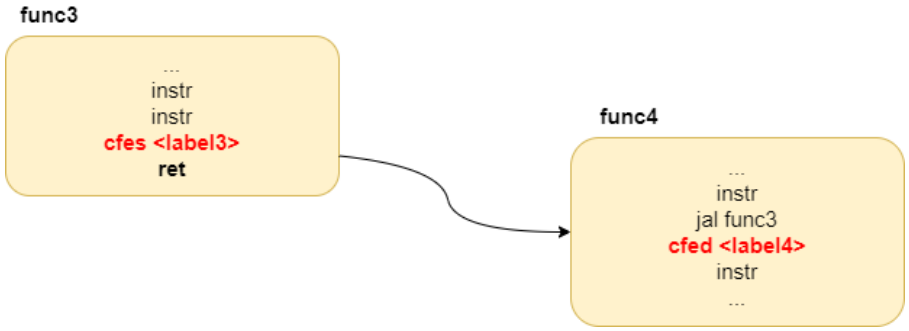


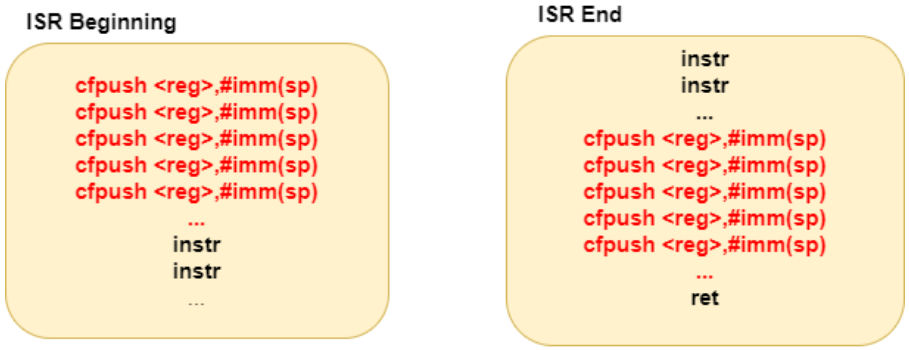
Figure 4.3: ARM-FPGA instrumentation code for ISRs.



(a) RISC-V instrumentation of forward edge.



(b) RISC-V instrumentation of backward edge.



(c) RISC-V instrumentation of ISRs.

Figure 4.4: Instrumentation with custom instructions for RISC-V.

Chapter 5

Instrumentation Tool - Implementation

The instrumentation tool, called *PROLEPSIS*, was first presented in 2021 [23]. At that moment, only ARM architecture was supported, and the tool was just a prototype with lack of stability and efficiency.

The aim of the thesis is to create a tool that is capable of automatically identifying the insecure edges in a target program, and then produce an instrumented version of the application, with the insecure edges, previously identified, protected against control-flow redirection attacks. It is in charge of implementing the instrumentation process in the *offline phase* of the solution presented in the Chapter 4. The tool is abstracted from the instrumentation code: it accepts any kind of instrumentation code, that is provided by the user as a parameter. The functionalities of the tool were expended by supporting also the instrumentation of applications compiled for the *RISC-V architecture*. In the current Chapter, the strategies adopted for implementation will be discussed, as well as the organization of the code, the data structures involved, and the stages of the instrumentation process implemented by the tool: parsing of the disassembly listing and reconstruction of the disassembly code, retrieval of insecure edges, label generation, code instrumentation.

5.1 Followed Strategies

By analyzing Control-Flow Integrity solutions, it is possible to understand that the effectiveness of the protection mechanism essentially relies on 2 factors:

1. the generation and analysis of the CFG in order to extract the edges in the program;
2. the insertion of instrumentation instructions in order to protect insecure edges from control-flow hijacking attacks;

As already mentioned, static analysis techniques do not allow the reconstruction of a complete CFG in general, as they might fail on retrieving the destinations associated with indirect jumps, and require a more in depth analysis. In addition, it is required to take

into account that the execution of Interrupt Service Routines (ISR) may happen in any point in the code, without the possibility of being tracked by the CFG.

The purpose of the tool is to extract and analyze the Control-Flow Graph, and from the CFG retrieve all possible insecure edges by also retrieving the destinations of indirect branches, and protect the execution context as described in the aforementioned paper [30]. In addition, the tool must be independent of the instrumentation code, that is provided externally by the user. The instructions still need to be compliant with the label-based protection mechanism as described in [30]. Therefore, as will be better described later, the instrumentation code requires to contain a *marker* indicating where the label value must be inserted in the code itself.

The considered instruction sets are the *ARMv7 with Thumb II extension* and *RISC-V RV32IM*. These have some peculiarities that must be taken into account during the edge retrieval process:

1. the presence of instructions that redirect the execution flow by changing the `PC` register value through operations that may involve tainted registers, which value may derive from memory locations potentially corrupted (this applies to both RISC-V and ARM);

2. unlike Intel-based architectures, the lack of proper `RET` machine instructions: by convention, ARM and RISC-V perform the return with a branch instruction to the content of the link register (ARM's `LR` or RISC-V's `RA`, return address), or by directly loading the return address in the `PC` register through a `POP` instruction or load (`LDMIA`) instruction;

3. the application of multiple instructions sets inside the same binary file: Executable files compiled against the ARMv7 thumb II architecture, may contain either instructions on 32-bits typical of the ARM instruction set, or 16-bits/32-bits instructions belonging to the Thumb instruction set;

In order to simplify the development, it was decided to keep two separate script: one for instrumenting ARM binaries and another one for RISC-V binaries. However, apart from minimal differences depending on the two different instruction sets, the algorithm followed remains conceptually the same, and can be outlined 5 different stages:

1. **`Parsing`**: it parses the file containing the disassembly of the code section (`.text`) and the disassembly of data sections (`.data`, `.rodata` and `.bss`), and are converted in two separate listing files;

2. **`CFG Extraction`**: the program is analyzed by applying a combination of static and dynamic analysis techniques in order to retrieve the Control-Flow Graph and obtain the destinations of indirect branches;

3. **`Edge Retrieval`**: insecure edges are identified, and stored in proper data structures with additional information for classification purposes;

4. **`Label Generation`**: from the edges, all the sources and destinations are associated to a given label; labels are chosen according to a given label generation algorithm;

5. `Instrumentation`: once the labels are available, the instrumentation code is applied to the locations previously identified in the code. Finally, all the sections are again merged in order to output a single instrumented assembly file, ready to be compiled again;

The `CFG Extraction` and `Edge Retrieval` phases of the instrumentation process heavily rely on the reverse-engineering framework `Radare2` [40] and the Python module `r2-pipe` [39], that allows programmatic interaction with the framework. As described in the Github repository of Radare2 [40]:

> *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...*

The tool is in fact extensively used to extract control-flow graphs and retrieve multiple high-level information about components of the program (especially functions) and for each instruction, information about their type are provided, e.g., if it is a `call` or a `ret` or a `jump` instruction.

The `CFG Extraction` stage (second phase of the process) heavily leverages the capabilities provided by the reverse-engineering framework. As previously stated, the interaction with *Radare2* in the script is performed through the module `R2pipe`. `r2pipe` provides the APIs `cmdj()` and `cmdJ()`, where the second one is extensively used in the script. `cmdj()` allows to retrieve the JSON object from Radare2, represented in Python through a *dictionary*. `cmdJ()` is another API working on top of `cmdj()` that takes the JSON object and converts it into a Python object that is finally returned to the caller.

The scripts, both for ARM and RISC-V platforms, take as input:

1. ***Target binary***: executable binary file, target of the instrumentation process;
2. ***Disassembly listings***: 2 disassembly listings retrieved through `objdump`. One is the disassembly of the `.text` section, while the other one lists `.data`, `.bss` and `.rodata` sections;
3. ***Instrumentation file***: file containing the instrumentation code to be inserted in the instrumented version program. Instrumentation code is provided in the form of a JSON array. It is important to highlight that the instrumentation instructions needs to be compliant with the label-based instrumentation strategy;

On the other hand, after the instrumentation process is applied the scripts will output the following files:

1. ***Instrumented Assembly***: an Assembly source file retrieved from parsing the disassembly listings provided as input, and augmented with instrumentation code provided as input; such a file will be later need to be compiled in order to obtain the final instrumented executable;
2. ***Insecure edge list***: A list of insecure edges that were retrieved during the *Edge retrieval* phase. Edges are classified in terms of backward and forward edges, and for each of them, some additional information is provided, like associated label or source and destination memory addresses;

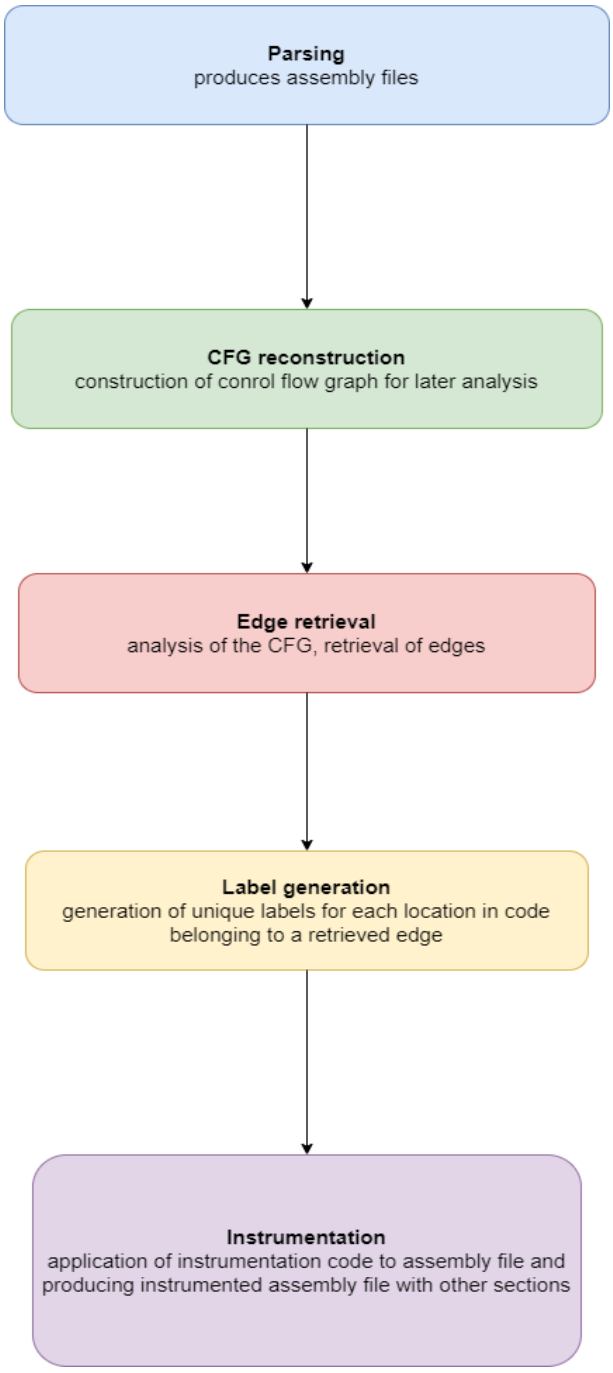


Figure 5.1: Phases of instrumentation tool.

In the following Sections, the various stages of the process will be analyzed with a greater level of details, describing the procedures followed and the data structures involved.

In particular, the internal details of the Python object defined will be first described, and later the different phases will be described and the main features highlighted.

5.2 Code Analysis

A structural analysis of the program may help in understanding the methods and strategies followed during each phase of the process. In particular, the main Python classes involved are: `Cfi`, `MultipleForwardEdges` and `MultipleBackwardEdges`.

In addition, two additional packages were defined, called `Utils` and `Protection`, where the first one provides helper functions for dealing with objects of type `MultipleForwardEdges` and `MultipleBackwardEdges` or for creating the *label map*, a Python dictionary containing the mapping between edge source/destinations and labels. On the other hand, the second one provides the functions for applying instrumentation to the Assembly code, and exposes the variables containing the instrumentation instruction, that will be loaded with the specifications provided by the user.

The class `Cfi` is the core class implementing almost all the functionalities of the tool. All the process stages, from `Parsing` to `Instrumentation` are in fact implemented in the `start()` method provided by `Cfi`. In particular, the `Cfi` class contains the following Python data structures:

- `code`: the disassembly code of the listing of the `.text` section retrieved with `objdump` from the target binary. The code is represented as a Python dictionary where each entry has as key the hexadecimal address of the instruction, and as value the instruction itself;
- `sections`: a python dictionary having as key the data section name (`.data` `.rodata` or `.bss`), and as value another dictionary that is produced by parsing the disassembly listing of the corresponding data section (again the listing is provided by `objdump`);
- `functions`: the list of functions (Interrupt Service Routines are included) retrieved by Radare2 when analyzing the executable. Radare2 provides several information for each function like: name, start address, size, number of basic blocks, **references to the function**;
- `edges_map`: a python dictionary where each entry has as key the address of the source of the edge and as value either a `MultipleForwardEdges` or `MultipleBackwardEdges` object (later described);
- `label_map`: a Python dictionary where each entry has as key the address of an instruction (belonging to an insecure edge part of edges map) and as value a unique label that must be used for instrumentation purposes;

As it is shown `Cfi` class uses two additional classes, **MultipleForwardEdges** and **MultipleBackwardEdges**, representing the different insecure edges retrieved in the program and stores them in a `edges_map`.

The class `MultipleForwardEdges` represents a a group of *forward edges* having a common source. In particular, the class wraps the following internal structures:

- ***source address***: represents the common source of the group of edges. Specifically stores hexadecimal address of the indirect call instruction;
- ***destination addresses***: contains a list of hexadecimal addresses where each one is a destination of the corresponding call instruction represented by **source address**;

On the other hand `MultipleBackwardEdges`, has the purpose of representing a group of *backward edges* having a common source. The inner structure of the class is similar to the ones used for forward edges but still slightly different. It wraps:

- ***source address***: represents the common source of the group of edges. Specifically stores hexadecimal address of the ret instruction;
- ***caller and return addresses***: contains a list of dictionaries, where each dictionary has only two items. In particular each dictionary contains a return address (destination address) to which the `ret` instruction will redirect the execution flow, and the associated call address. The call address corresponds to the address of the instruction, (right before the return address) that called the function ending with the `ret`.

Both classes provides several methods for interacting with them like, adding destination addresses or retrieving the source or the list of destination (with corresponding callers in the case of `MultipleBackwardEdges`).

As anticipated the `Protection` module provides the necessary abstraction in order to apply instrumentation the program. The module contains a variable called `types` storing a dictionary which keys corresponds to the instrumentation type (in the form of string like "type1", "type2",...) and values to the corresponding instrumentation instructions. The dictionary is retrieved by parsing the *instrumentation file*, provided in JSON format by the end user.

5.2.1 Parsing

As stated, the instrumentation is not applied directly to the executable binary but it is first required to retrieve an assembly representation, and then apply the instrumentation code there. The parsing stage take cares of producing an assembly file, to which instrumentation can be applied, from the disassembly listing retrieved with `objdump`. In particular, it is first performed the parsing of the disassembly of `.text` section, and later the disassembly of `.data` `.rodata` and `.bss` sections are parsed.

For what concern the `.text` section, the disassembly listing will provide the list of assembly functions, and for each of them a list rows providing for every instruction the following information:

- the hexadecimal address of the instruction;
- the operational code of the instruction (opcode);
- the mnemonic representation of the instruction;
- optionally a comment. Comments will be useful for resolving labels and *pc-relative load operations*;

Here, as an example, follows the disassembly listing of the exit function:

From the `.text` disassembly listing, every row is parsed and are extracted only the the hexadecimal address and the mnemonic representation, along with the comment, as a string. Address and mnemonic instruction will compose a entry of a python dictionary, named *code* in the program, where the key corresponds to the address and the value to

```
00008010 <exit>:
    8010: b508        push  {r3, lr}
    8012: 2100        movs  r1, #0
    8014: 4604        mov r4, r0
    8016: f000 fdc1   bl  8b9c <__call_exitprocs>
    801a: 4b04        ldr r3, [pc, #16] ; (802c <exit+0x1c>)
    801c: 6818        ldr r0, [r3, #0]
    801e: 6bc3        ldr r3, [r0, #60] ; 0x3c
    8020: b103        cbz r3, 8024 <exit+0x14>
    8022: 4798        blx r3
    8024: 4620        mov r0, r4
    8026: f000 fe81   bl  8d2c <_exit>
    802a: bf00        nop
    802c: 00008d3c   .word 0x00008d3c
```

Figure 5.2: Disassembly listing of text function.

the assembly instruction. The comment is still kept for resolving later pc-relative load instructions. During the parsing, the direct branch instructions must be resolved. In order to do that the comments provided by objdump are exploited. Taking the previous example of the exit function the instruction `bl 8b0c <__call_exitprocs>` will be processed and transformed into `bl __call_exitprocs`. Same happens for the instruction `cbz r3, 8024 <exit+0x14>`, that will be transformed in `cbz r3, labN` and the instruction at address `exit+0x14` will be prepended with the label `labN`.

On the other hand for the data sections less information are needed in fact every row of the listing only provides:

- an hexadecimal address;
- data bytes;

Again every data section is parsed in a way to produce a dictionary having as keys memory addresses and as values the corresponding values in memory. Depending on the platform, ARM or RISC-V, the syntax may slightly change. In the case of ARM, the conversion will looks like the one in Figure 5.3

One last step of the parsing phase is to resolve the pc-relative load instructions. In order to do that, first from the comment of the load instruction, it is retrieved the memory address in the data section. Later the dictionary representing the data section is consulted in order to retrieve from the memory address the corresponding symbol. Finally the pc-relative load is replaced with the symbol retrieved.

The final conversion of the same exit function, previously taken as an example, can be observed in figure 5.4. As it is possible to notice, the `ldr r3, [pc, #16]` is a pc-relative load instruction that finally is converted with `ldr r3, =_global_impure_ptr`.

```
00018d58 <impure_data>:
   18d58: 00000000  andeq r0, r0, r0
   18d5c: 00019044  andeq r9, r1, r4, asr #32
   18d60: 000190ac  andeq r9, r1, ip, lsr #1
   18d64: 00019114  andeq r9, r1, r4, lsl r1
   18d68: 00000000  andeq r0, r0, r0
   18d6c: 00000000  andeq r0, r0, r0
   18d70: 00000000  andeq r0, r0, r0
   18d74: 00000000  andeq r0, r0, r0
   18d78: 00000000  andeq r0, r0, r0
   18d7c: 00000000  andeq r0, r0, r0
   18d80: 00000000  andeq r0, r0, r0
   18d84: 00000000  andeq r0, r0, r0
   18d88: 00000000  andeq r0, r0, r0
   18d8c: 00000000  andeq r0, r0, r0
   18d90: 00000000  andeq r0, r0, r0
   18d94: 00000000  andeq r0, r0, r0
```

(a) Disassembly listing of `.data` section fragment.

```
impure_data:
  .word 0x00000000
  .word 0x00019044
  .word 0x000190ac
  .word 0x00019114
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
  .word 0x00000000
```

(b) Assembly representation retrieved after parsing of `.data`.

Figure 5.3: data listing conversion.

```
00008010 <exit>:
    8010: b508        push  {r3, lr}
    8012: 2100        movs  r1, #0
    8014: 4604        mov r4, r0
    8016: f000 fdc1   bl  8b9c <__call_exitprocs>
    801a: 4b04        ldr r3, [pc, #16] ; (802c <exit+0x1c>)
    801c: 6818        ldr r0, [r3, #0]
    801e: 6bc3        ldr r3, [r0, #60] ; 0x3c
    8020: b103        cbz r3, 8024 <exit+0x14>
    8022: 4798        blx r3
    8024: 4620        mov r0, r4
    8026: f000 fe81   bl  8d2c <_exit>
    802a: bf00        nop
    802c: 00008d3c  .word 0x00008d3c
```

(a) Disassembly listing of `exit` function.

```
exit:
  push  {r3, lr}
  movs  r1, #0
  mov r4, r0
  bl  __call_exitprocs
  ldr r3, =_global_impure_ptr
  ldr r0, [r3, #0]
  ldr r3, [r0, #60]
  cbz r3, lab0
  blx r3
lab0:   mov r0, r4
  bl  _exit
  nop
  .ltorg
```

(b) Assembly representation retrieved after full parsing of the `.text` section.

Figure 5.4: data listing conversion.

5.2.2 CFG Extraction

In this phase, the executable binary is analyzed in order to retrieve abstract representations of the program in particular the Control-Flow Graph. This phase is mostly carried out directly by the reverse engineering framework *Radare2*. The framework will apply several static and dynamic analysis techniques, and once finished, it allows to be queried in order to retrieve the result of the analysis. For example it is possible to automatically retrieve the Control-Flow Graph of a function in image format (figure 5.5) or JSON format, more useful for scripting purposes.

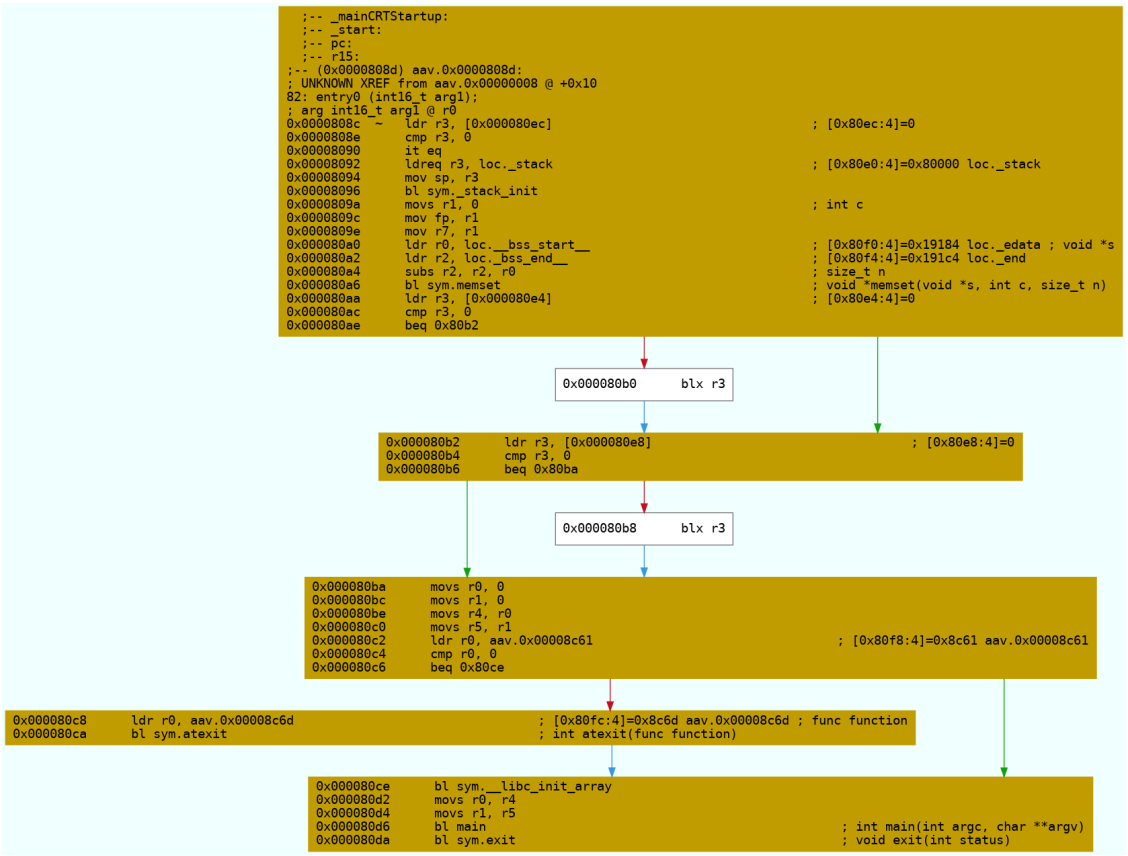


Figure 5.5: Control-Flow graph of a function provided by Radare2.

What is particularly useful, for the purposes of the script, are the information provided for every function in the program.

With the help of the API `cmdJ()` provided by r2pipe, it is possible to immediately retrieve from Radare2 a list of Python objects each one representing several information about a function. The object will contain the following fields, among the others:

- ***name***: provides function name;
- ***offset***: the relative memory address pointing to the beginning of the function;
- ***size***: the space in bytes occupied by the function;
- ***codexrefs***: a list of control-flow transfer instructions redirecting the execution flow to code location in the function itself;

As we be later described in the next subsection the `codexrefs` property will be particularly important in order to identify all the callers of a function. Every python object inside the `codexrefs` field will consists of the following properties:

- ***addr***: provides the address of the control-flow transfer instruction (this can be a memory address part of a different function in the code);

- *type*: the type of control-flow transfer. It can be `CALL` for identifying a call instruction or `JUMP` for identifying jump instructions;
- *at*: the destination address of the control-flow transfer instruction at the corresponding address `addr` (this is a memory address part of the function);

The following image (figure 5.6) summarizes better the relationships and meaning of the different properties. In the example function `func2()` calls `func1()`. For `func1` Radare2 will make available, through R2-pipe, a Python object representing the function, that will contain (in the `codexrefs` property) the call from `func2()` to `func1()`. These information are critical in order to extract all the edges from the Control-Flow Graph.

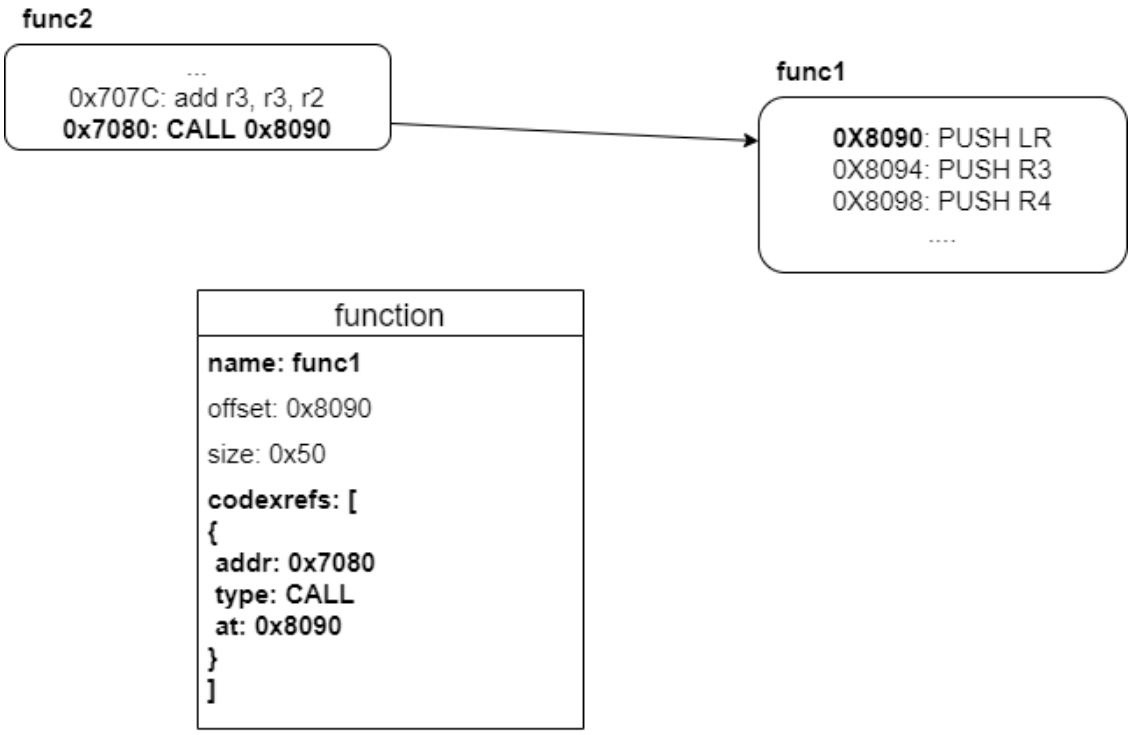


Figure 5.6: function object extract from Radare2 embedding calls to the function.

Another interesting object that can be extracted from Radare2 are Python objects representing instructions. For every instruction Radare2 allows to retrieve the type that can be `call` for direct call `rcall` for indirect call to register or `ret` for return instructions. It is important to notice that also instructions like `pop pc` will be considered as type `ret`. This allows to use the same algorithm for potentially any CPU architecture.

Another useful information provided by Radare2 are the targets of indirect call instructions. Given an instruction of type `rcall`, the list of targets will be provided in the property, named `refs`, of the object representing the indirect call instruction.

5.2.3 Edge Retrieval

The edge retrieval stage is the core of the script, where the information provided by Radare2 are exploited in order to extract the list of edges requiring protection.

Before proceeding with the description of the algorithm adopted, it is important to highlight that the indirect call instructions are all considered as insecure even if some of them may not be. It was decided to adopt this conservative approach because it was not always possible to understand if the target was actually retrieved from insecure memory regions or not.

In addition, it was noticed that the number of indirect forward edges was significantly lower with respect to the total number of insecure backward edges. For this reason the optimization on indirect forward edges in any case will not provide significant benefits in terms of memory costs and performances.

In order to retrieve the insecure edges it necessary to first collect all the functions of the program provided by Radare2. Once all of them are available the edge retrieval phase can start.

For every function retrieved the following operations are performed:

1. the list of instructions is retrieved and stored in a list;
2. the list of caller instructions of the current function is retrieved through a recursive algorithm, along with the next instruction the called function will return to (later better described);
3. from the list of instructions retrieved at point 1 the indirect call instructions, along with their targets, are retrieved. Every edge is inserted in a dictionary having as key the source and as value the set of destinations (the value will be an object of type `MultipleForwardEdges`);
4. from the list of instructions retrieved at point 1 the insecure return instructions, are retrieved. For every pair return instruction-destination instruction a backward edge is added to the same dictionary of point 3 (along with the caller instruction). In this case to the dictionary will be eventually added objects of type `MultipleBackwardEdges` as value and again the source as key;

As it is described the step 2 of the algorithm requires a recursive function in order to retrieve the function callers, and consequently the destinations of return instructions. This is required because in a program it may happen that a function `function1()` perform a call to `function2()` and `function2()` perform a jump to `function3()`. This result in `function3()` returning directly to `function1()`. This behaviour is summarized in figure 5.7.

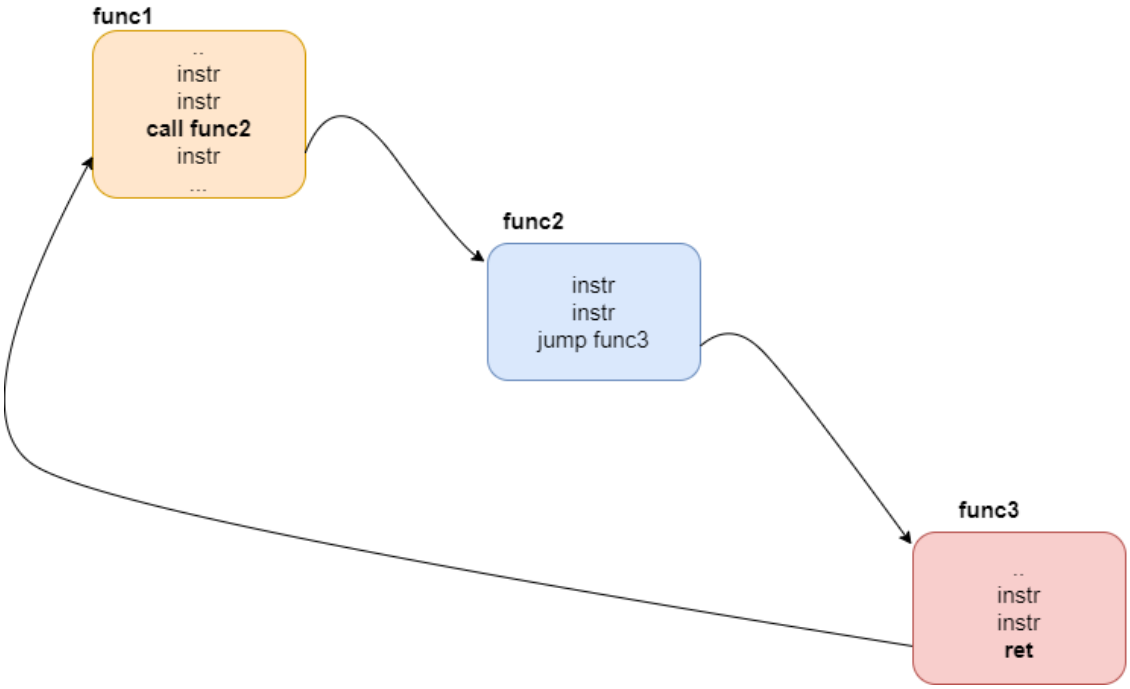


Figure 5.7: call followed by jump and by ret.

In order to retrieve all the caller and destination instructions the following recursive algorithm was implemented:

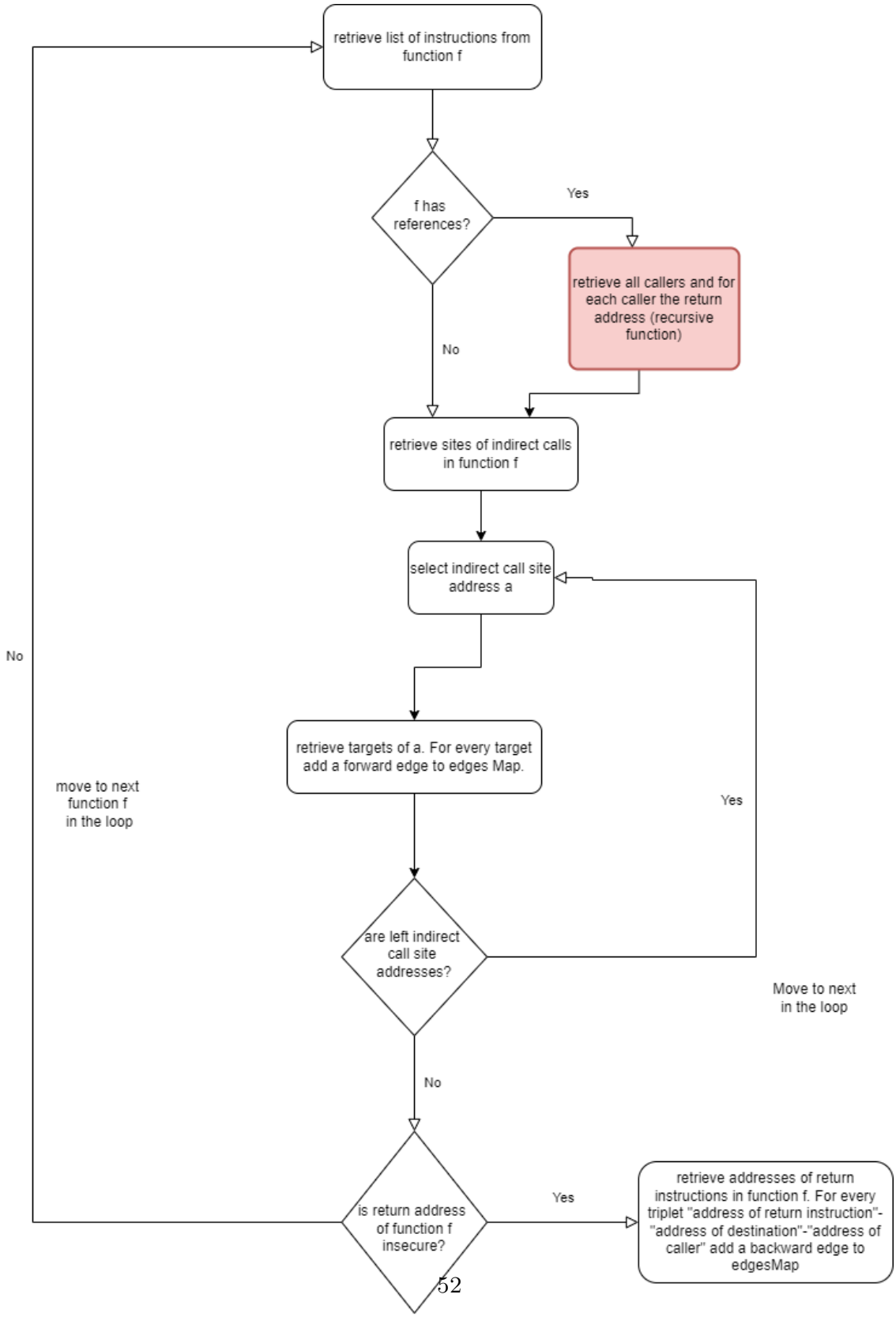
1. given a function first the `codexrefs` property is analyzed (described previously in the CFG extraction subsection) and are retrieved two lists: one containing calls to the function and another one containing jumps to the function;

2. the list containing calls is mapped to a list containing objects with address of the call instruction and address of instruction the execution flow will return to after the call. This list is then appended to a bigger list initialized to be empty before the recursive procedure starts;

3. for every jump instruction in the second list (retrieved at point 1) the corresponding function is retrieved and the algorithm proceeds recursively on that function;

Figures 5.8 and 5.9 summarize the behaviour of the two algorithms. The red block in figure 5.8 indicates when the recursive algorithm, represented in figure 5.9, is called during the overall edge retrieval process.

It is important to highlight that in order to define if the return instructions of a function are insecure or not, it is checked if the return address is ever stored in the stack or any other writable memory region, that may be potentially corrupted. This check is dependent on the targeted Instruction Set Architecture since the name of the register holding the return address of course will be different.


retrieve list of instructions from function f
f has references?
Yes
retrieve all callers and for each caller the return address (recursive function)
No
retrieve sites of indirect calls in function f
select indirect call site address a
retrieve targets of a. For every target add a forward edge to edges Map.
are left indirect call site addresses?
Yes
Move to next in the loop
No
is return address of function f insecure?
Yes
retrieve addresses of return instructions in function f. For every triplet "address of return instruction"-"address of destination"-"address of caller" add a backward edge to edgesMap
No
move to next function f in the loop

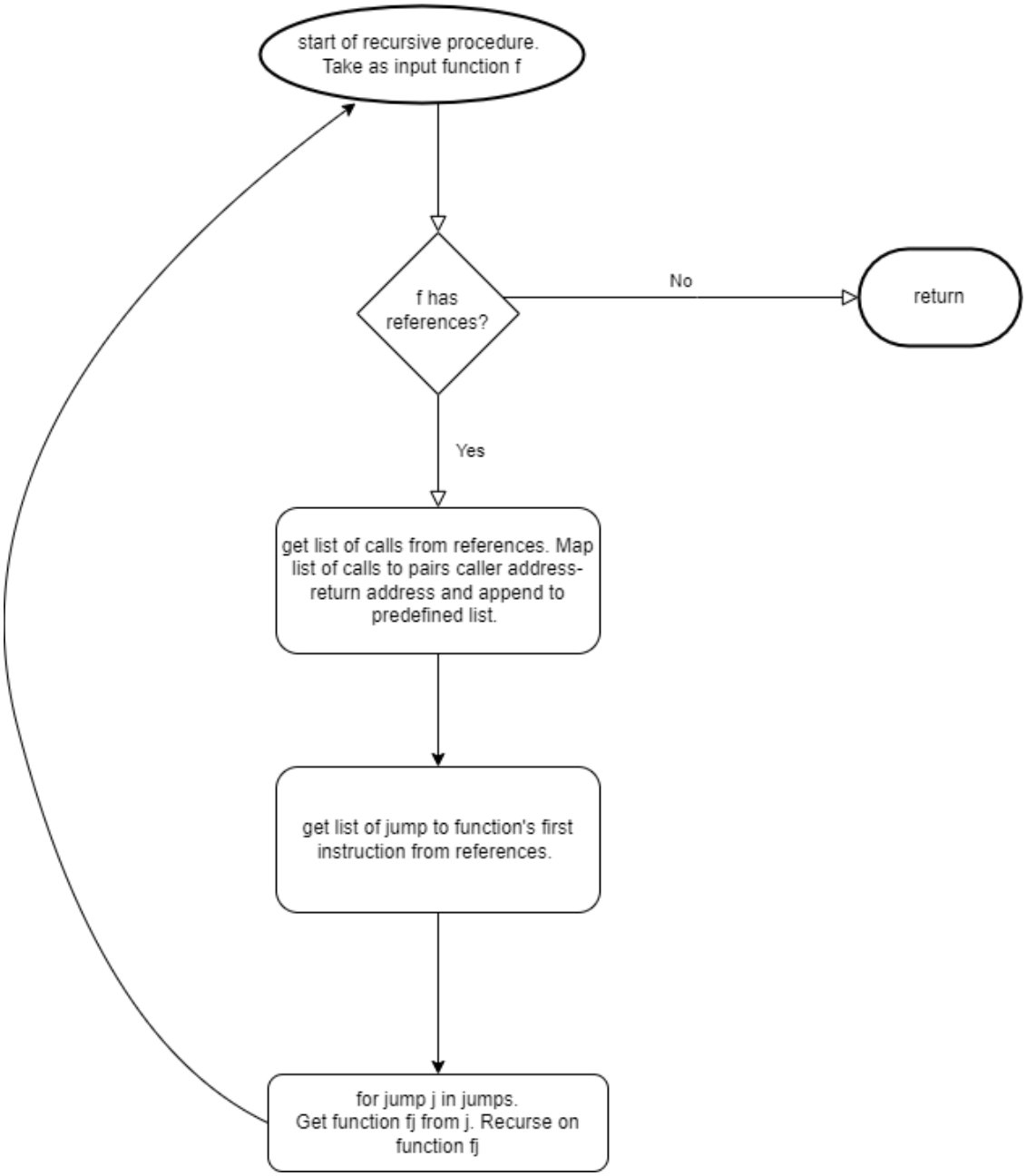


Figure 5.9: recursive algorithm for retrieving the list of addresses to call instructions and the corresponding return address.

5.2.4 Label Generation

During this phase the *edges map* created during the `edge retrieval` phase is taken as input. From the map are extracted all the memory addresses (either corresponding to a

source or destination of an edge), possible duplicates are filtered out and the remaining memory addresses are stored in a list. As a final step the list of addresses is passed to a label generation algorithm that will assign a unique label to each memory address. A label corresponds to an integer that can be represented on 13 bits. This a constraint that must be respected in order to make the solution compatible with the FPGA-based solution described in the previous Chapter. The label generation algorithm will not be described since it is out of the scope of the thesis, but was directly provided by Paolo Prinetto's research team as Python module to be added to the tool.

5.2.5 Instrumentation

During the last stage of the overall process, the assembly code, retrieved during the `parsing` phase, is modified by adding instrumentation instruction to the instrumentation points. First are instrumented the forward edges and later the backward edges. The algorithm is separated in two parts:

1. first the `MultipleForwardEdge` objects from the edges map are retrieved;
2. for every `MultipleForwardEdge` object it is checked if there are multiple destinations or not and based on that a given type of instrumentation is applied;
3. then `MultipleBackwardEdge` objects are retrieved from the edges map;
4. for every `MultipleBackwardEdge` object it is checked if there are multiple destinations or not and based on that a given type of instrumentation is applied;

Once the assembly code is augmented with the instrumentation code the the resultant assembly code is written to a file.

As a final step the list of forward and backward edges in the edges map are converted to JSON objects and written to file. In figure 5.10 an example of content of the file containing forward and backward edges in JSON format.

```
...
  {
    "type": "forward",
    "source": {
      "source_addr": "0x8ae0",
      "source_label": 716364
    },
    "destinations": [
      {
        "dest_addr": "0x8030",
        "dest_label": 222873
      }
    ]
  },
  {
    "type": "backward",
    "source": {
      "source_addr": "0x8ae6",
      "source_label": 216529
    },
    "destinations": [
      {
        "dest_addr": "0x80d2",
        "dest_label": 305699,
        "caller_addr": "0x80ce",
        "caller_label": 305699
      }
    ]
  },
  {
    "type": "forward",
    "source": {
      "source_addr": "0x8c82",
      "source_label": 141943
    },
    "destinations": [
      {
        "dest_addr": "0x8044",
        "dest_label": 904573
      }
    ]
  },
...
```

Figure 5.10: forward and backward edges in JSON format.

Chapter 6

Experimental Results

In the current Chapter, results obtained by running PROLEPSIS for different *benchmarks* are presented.

In particular, it was chosen to select benchmarks from the *Embench IoT* project [22], stored in a Github repository containing some test program that can be built over multiple embedded platforms. Among them, the architecture of our interest are ARM Cortex-M (having Thumb II ISA) and 32-bit integer set of RISC-V.

Tables 6.1 provides the number of edges retrieved for each considered benchmark, both when the benchmark was built for the ARM architecture and for the RISC-V one.

(a) Results from running PROLEPSIS on ARM.

Benchmark	total edges (approx.)	insecure edges	overhead	.text section overhead	total overhead on ELF
aha-mont64	164	17	476 B	14.0%	1.11%
crc32	130	15	436 B	32.3%	1.01%
edn	202	18	448 B	11.1%	0.98%
matmult-int	179	20	544 B	25.4%	1.25%
nettle-aes	197	23	720 B	15.7%	1.32%
nettle-sha256	221	32	904 B	13.6%	1.88%
slre	445	42	1284 B	22.1%	2.76%
statemate	461	16	480 B	6.6%	0.93%

(b) Results table for RISC-V

Benchmark	total edges (approx.)	insecure edges	overhead	.text section overhead	total overhead on ELF
aha-mont64	156	11	52 B	0.98%	0.53%
crc32	110	9	48 B	2.42%	0.56%
edn	177	9	28 B	0.44%	0.21%
matmult-int	154	11	12 B	0.38%	0.12%
nettle-aes	165	15	48 B	0.57%	0.19%
nettle-sha256	200	20	88 B	0.68%	0.47%
slre	422	28	72 B	0.76%	0.48%
statemate	450	10	36 B	0.46%	0.02%

Table 6.1: Experimental results.

It must be considered that benchmarks compiled following the instructions provided in the Embench Github repository [22] do not contain Interrupt Request handlers. The

presence of Interrupt Request handlers may increase the values for the overhead.

It is possible to notice how the overhead in RISC-V executables is lower with respect to the one on ARM executables. This happens because, as anticipated in the previous Chapters, for the RISC-V instrumentation code were defined custom instructions, allowing to perform the same operations performed on ARM but with one single instructions, drastically reducing the size of the instrumentation code.

Another important considerations is that the disassembly and recompilation operations might cause a loss of data from the original binary. In order to produce the disassembly listing, later parsed, `objdump` is used: during the disassembly, that command might not able to properly disassemble code, especially when code and data overlaps. This can be observed especially on RISC-V, where in some cases the overhead is expected to be higher.

Chapter 7

Conclusions and Future Work

The objective of the thesis was to provide an instrumentation tool, written in Python, that is able to automatically perform the following operations starting from an executable binary:

1. take the executable binary as input and convert it to a compilable assembly file;
2. apply a set of static and dynamic analysis techniques to the binary, by exploiting the primitives provided by a reverse engineering framework, in order to retrieve the location in the code requiring instrumentation;
3. insert instrumentation instructions in the code location identified at the previous step;

The tool allows to specify the instrumentation code by inserting it in a JSON file passed as input to the tool itself.

The instrumentator was integrated with two existing CFI solutions, one for ARM and the other one for RISC-V. For the CFI solution instrumenting the RISC-V binaries, custom instructions were defined.

The tool was executed against a set of benchmarks and were shown the results in terms of overhead and insecure code locations identified.

Even if the tool proved to work for ARM and RISC-V binaries, there is still room for improvements.

As already described, the tool uses Radare2 as an underlying reverse engineering framework for applying static and dynamic analysis techniques. During testing, the tool proved to be effective in retrieving backward edges, but likely was not able to retrieve all forward edges. Radare2 is heavily exploited for retrieving the targets of indirect jump instructions, but seems to fail in retrieving some of them. It was noticed how the *angr* reverse engineering Python library succeded, in some cases, where Radare2 failed. An improvement would be the implementation of a custom dynamic analysis procedure that is able to retrieve the targets of indirect jumps, that is invoked when Radare2 fails in retrieving them.

Acknowledgements

I would like to thank Professor Paolo Prinetto for giving me the opportunity to work with the team members of his research group.

Special thanks go to Gianluca Roascio for having guided me during the project development and the elaboration of the final document, and for been always available and helpful.

Finally i would like to thank the friends, my family and my girlfriend having supported me along the path.

Bibliography

[1] 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.

[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] Frances E. Allen. «Control Flow Analysis». In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] *CWE-121: Stack Based Buffer Overflow*. https://cwe.mitre.org/data/definitions/121.html.

[14] *CWE-122: Heap Based Buffer Overflow*. https://cwe.mitre.org/data/definitions/122.html.

[15] *CWE-126: Buffer Over-read*. https://cwe.mitre.org/data/definitions/126.html.

[16] *CWE-134: Use of Externally-Controlled Format String*. https://cwe.mitre.org/data/definitions/134.html.

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

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

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

[20] 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.

[21] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. «HAFIX: Hardware-Assisted Flow Integrity eXtension». In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744847.

[22] Embench. *Embench-Iot repository*. https://github.com/embench/embench-iot. 2019.

[23] Valentina Forte, Nicolò Maunero, Paolo Prinetto, and Gianluca Roascio. «PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity». In: *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 2021, pp. 1–6. DOI: 10.1109/ICECCME52200.2021.9591080.

[24] 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.

[25] 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.

[26] *Getting around non-executable stack (and fix)*. https://seclists.org/bugtraq/1997/Aug/63. 1997.

[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] 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.

[29] 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.

[30] Nicolò Maunero, Paolo Prinetto, Gianluca Roascio, and Antonio 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)*. 2020, pp. 1–10. DOI: 10.1109/DTIS48698.2020.9081314.

[31] 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).

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] A.K. Sharma. *Programmable Logic Handbook: PLDs, CPLDs, and FPGAs*. McGraw-Hill handbooks. McGraw-Hill, 1998. ISBN: 9780070578524. URL: https://books.google.it/books?id=UH4eAQAAIAAJ.

[37] 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.

[38] 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.

[39] Radare2 Team. *R2pipe repository.* https://github.com/radareorg/radare2-r2pipe.

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

[41] Weike Wang, Muyang Liu, Pei Du, Zongmin Zhao, Yuntong Tian, Qiang Hao, and Xiang 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.

[42] 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.

[43] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. «A Platform for Secure Static Binary Instrumentation». In: *SIGPLAN Not.* 49.7 (2014), 129–140. ISSN: 0362-1340. DOI: 10.1145/2674025.2576208. URL: https://doi.org/10.1145/2674025.2576208.