POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Asymmetric Verification for Control-Flow Integrity in Multicore Embedded Systems

Author: Marco Meloni

Supervisor: Paolo Ernesto Prinetto

October, 2021

Abstract

Embedded systems play an increasingly central role in modern society, thanks to the spread of the Internet of Things (IoT) and the use of smart control systems in the automotive and aeronautical domains. Their use in mission-critical or safety-critical systems makes them attractive targets for attackers. Nowadays, several of these are mixed-criticality systems, meaning that they run both high-criticality tasks (e.g., a car control system) and low-criticality ones (e.g., infotainment). High-criticality routines often employ Real-Time Operating Systems (RTOS) to enforce hard real-time requirements, while the tasks with lower constraints can be delegated to more generic operating systems. Right now, security solutions for real-time embedded systems are not as mature as the ones for general-purpose systems, which often make assumptions that do not hold in the embedded domain. Control-Flow Integrity (CFI) is a powerful security technique to protect against many kinds of binary attacks, such as Arbitrary Code Execution (ACE) like Return-Oriented Programming (ROP). Through static analysis of the application, a Control-Flow Graph (CFG) is created, describing all the allowed branches and their valid targets. At runtime, control-flow transfers are checked against the graph and possible deviations are stopped.

The aim of this thesis is to investigate CFI application in the RTOS domain, first taking a picture on what is currently under experimentation in the embedded world, and then identifying the critical issues which are still unsolved. Finally, the thesis proposes a possible solution for multicore systems running mixed-criticality tasks. By using an embedded hypervisor, predefined cores could be dedicated to only high or low-criticality tasks. This way, the complete CFI monitoring will be offloaded to the lower-criticality core, which makes it possible to implement a solid defense for high-criticality tasks while not compromising on their tight deadlines. In this proposal, the high-criticality core sends a request for CFI verification, along with the necessary information, whenever a vulnerable operation is about to happen, and stops the task only if a violation is detected. The work also presents theoretical results about a possible implementation for ARM platforms running a minimal RTOS, along with the computation of the estimated performance penalties.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The modern world is becoming increasingly dependant on embedded systems in pretty much every aspect of life. Real-Time Operating Systems (RTOS), designed for these platforms, can be found everywhere, from common household appliances, domotics, and entertainment systems, to more critical applications in transportation, industry, and several highly critical fields. Ensuring that the released products are not faulty is of utmost importance, especially for systems that can put human life in danger, like braking controllers in a car or industrial machine controllers. Failures can occur not only for faults in the operating code, but also for code vulnerabilities that could be exploited by an attacker. The more critical the system, the higher the chance of being a target for some attackers. One of the most extreme examples of this are the attacks carried out in the past few years to electric grids [19] and nuclear plants [34] [24] in cases of cyber-terrorism and cyber-warfare.

There are several classes of attacks to be aware of, especially because of the major use of C/C++ in embedded platforms [47], with their memory vulnerabilities [75]. Some of them are *dangling pointers* [5], which are mistakenly freed pointers whose use could generate unpredictable effects, and *buffer overflows* [65] [33], which are out-of-bounds writes to buffers that can write data in the adjacent stack or heap data. Some of the most dangerous attacks are the ones that allow the attacker to execute arbitrary code. This kind of exploits is usually carried out through the exploitation of *buffer overflow* vulnerabilities, which might be used to overwrite the original return addresses and redirect control-flow transfers to some injected code (e.g., *stack smashing* [63]). Although, code injection attacks have been rendered fairly ineffective thanks to the widespread implementation of Write XOR Execute policies [71]. A more effective application of control-flow redirection is presented by Code-Reuse Attacks (CRA) [58] (e.g., Return Oriented Programming [15] [39] [22] [21] [50]), which instead use sequences of code already present in the original program to execute their attack. In response to this, in 2005, Abadi *et al.* presented the concept of Control-flow Integrity [1], a solution which is expected to protect against all kinds of attacks based on the idea of control-flow redirection. The basic idea is to monitor each control-flow transfer by checking it against a predefined model of allowed transfers obtained through static analysis of the code/binary.

A large number of CFI solutions exist for general-purpose systems. For example, Reins [77] introduces CFI functionalities to Windows on an x86 platform, while O-CFI [62]

provides a proof of concept for 32-bit Windows, but their approach is applicable to 64-bit Windows as well as Linux and OS X. Works in [80] [10] [78] are just a few examples of solutions developed for Linux running on x86 platforms. Unfortunately, the scientific proposal for such solutions in the embedded world is not this rich. In fact, embedded systems often lack the same support for process and memory protection. RTOSes also have some specific challenges that need to be addressed, e.g., the possible presence of a preemptive scheduler. A scheduler introduces the same problem brought by interrupts, i.e., the possibility of unexpected control-flow transfers that fall outside of a statically-created model. Another challenge is represented by the reduced resources present in most embedded systems, which could be an hindrance for some CFI techniques (as for example, replication techniques like shadow stacks). Finally, the strict timing requirements imposed by real-time systems, especially in highly critical applications, make it imperative to reduce the performance overhead as much as possible.

This thesis goes through a massive research into the current panorama for the application of CFI in the RTOS world. An analysis of the current state of the art has been done, and through that, a definition of the main principles for an ideal solutions have been defined. The most important issues that still need to be solved are also presented.

The core of the thesis is a proposal for a solution for the case of mixed-criticality systems. These are systems that handle both high-priority or high-risk tasks, which could potentially have serious consequences on safety, and lower-criticality ones, which generally have laxer requirements. An example could be represented by an automotive application, where it is possible to have, on the same platform, both the car control system (which could be responsible for the braking system, ABS, etc) and the infotainment system. The first one need to be extremely reliable and respect strict deadlines, while the second one can accept some possible performance loss. For these kind of situations, a solution for multicore systems has been hypothesised, which assigns the workloads of different criticalities to different cores, and make the lower-criticality core handle the CFI verification for the higher-priority ones. This architecture has the main goal of reducing the impact on schedulability created by the overhead needed by the CFI checks. All of this is done by using an embedded hypervisor implementing *static partitioning*, which makes it possible to assign cores and memory to selected systems and making them able to communicate through interrupts and a configurable shared memory space. In this way, the RTOS running the high criticality task would instead be able to operate with tight deadlines, with just the overhead from sending the necessary information to the monitoring core.

The proposed solution is theorised for ARM processors, as this is the most common family the embedded field [26], and by using the Bao hypervisor [59] alongside FreeRTOS [41]. Of course, the same theoretical principles could be applied to different RTOSes or hypervisors.

1.1 Outline

The remainder of the thesis is organized as follows. Chapter 2 briefly introduces the basics of security in the embedded domain and goes into details for laying down the basis needed for this thesis, by explaining the concepts of Control-Flow Integrity, Real-Time Operating Systems and Embedded Hypervisors. Chapter 3 presents our research done on

the current state of the art relating CFI solutions for embedded systems, defining the ideal characteristics and the open issue in the current research. Chapter 4 describes our proposal for a solution aimed at mixed-criticality systems. In Chapter 5, a theoretical evaluation of the introduced overheads and timings is presented. Finally, in Chapter 6 the work is concluded, summarising the results and introducing ideas for future work.

Chapter 2

Background

2.1 Security in the Embedded Domain

The security of embedded systems is a very popular topic nowadays, as everyday billions of these devices collect and exchange data about our lives. The limited resources they have put the community up against the challenge to adopt anyway secure protocols to handle every level of operations. Embedded systems often lack proper security protocols regarding authentication, communication, and privacy protection. Some examples of this are the widespread use of standard passwords and the lack of protected firmware updates. The adoption of strong authentication, secure updates, encrypted data transfer/communication and physical hardening are thus paramount to ensure safety on embedded platforms [67]. Accurate selection of components is also important, since the inclusion of outdated or insecure software/hardware components could potentially compromise the security of the whole system.

Still nowadays, most of the embedded software, especially at the low level, is written in C or C++ [47]. This is because of the powerful features of these languages, which allow very low-level management of resources and memory. Unfortunately, they do not implement any native protection of memory, making systems vulnerable to several attacks [65] [5] [70]. Historically, the most common kind of exploited vulnerability is the possibility of a *buffer overflow* [20]. Since the C language does not offer any native mechanism for boundary checks, writing data to an input buffer can turn in going over its limits, ending up with overwriting the subsequent data on the memory, e.g., the stack. When done deliberately as an attack, this is called *stack smashing* [63].

The reduced resources and possible real-time requirements of embedded systems pose even further limits, by making the use of very secure and powerful but resource-heavy OSes and memory protection techniques sometimes undesirable or unacceptable. Embedded platforms often do not have the same kind of hardware security functionalities that general-purpose systems have. For example, most embedded systems do not have a Memory Management Unit (MMU) and thus they are not able to use Virtual Memory as a way to isolate adequately the address spaces of their processes. A Memory Protection Unit (MPU) is becoming more common in some embedded platforms (like the most recent ARM series), which does allow improved memory protection potentiality. Still, not all RTOSes support them correctly and are making use of them to the fullest yet.

2.2 Binary Attacks

The term *binary exploitation* refers to the process of exploiting the vulnerabilities in a compiled application to generate maliciously unexpected results from the program. This can translate to several classes of attacks, but in our case the focus will be on the case of *memory corruption*. By abusing vulnerabilities in the program that allow an attacker to read or write data that should be protected, it becomes possible to get higher privileges, obtain sensible data, damage important data, and so on. Among the various possible effects, the investigation of this thesis is mostly interested about the possibility of hijacking a program's control-flow to execute malicious code.

2.2.1 Code Injection

The real danger of memory corruption and buffer overflows relates not just to the possibility to leak memory content or overwrite user data. By using vulnerable functions like `scanf` or `gets`, an attacker could perform a stack smashing attack to overwrite sensitive control data on the stack. If this happens, the return address of the function can be overwritten to redirect it towards a different code location. This attack paradigm takes the name of Arbitrary Code Execution (ACE), i.e., the ability to execute arbitrary code that is not part of the normal program code. One way to do this is to embed this code inside of the data passed to the buffer itself through buffer overflow, and then redirect the return address to the start of this code inside the buffer. To make this easier, sometimes a *NOP sled* is used, i.e., several NOP instructions are put at the start of the buffer before the desired injected code. In this way, the address can also be not the right one: as long as it falls inside of this series of NOPs, it will eventually reach the start of our malicious code, by basically "sliding down" the NOP instructions, like a sled down a mountain. This technique is called *Code Injection*, as the attacker injects the code in the form of a payload into the memory. The payload injection and the control-flow redirection do not necessarily need to happen in a single operation. As long as the attacker is able to do both, even in different stages, the attack will be successful.

These kind of attacks have been mostly defeated thanks to the widespread adoption of Write XOR Execute policies [69], which make sure (usually though hardware enforcement) that a given memory page can never be both executable and writeable. This is not a complete solution against buffer overflows and control-flow hijacking *per se*, since it is still possible to overwrite the return address and redirect the control-flow, but it gets to stop the execution of the injected code.

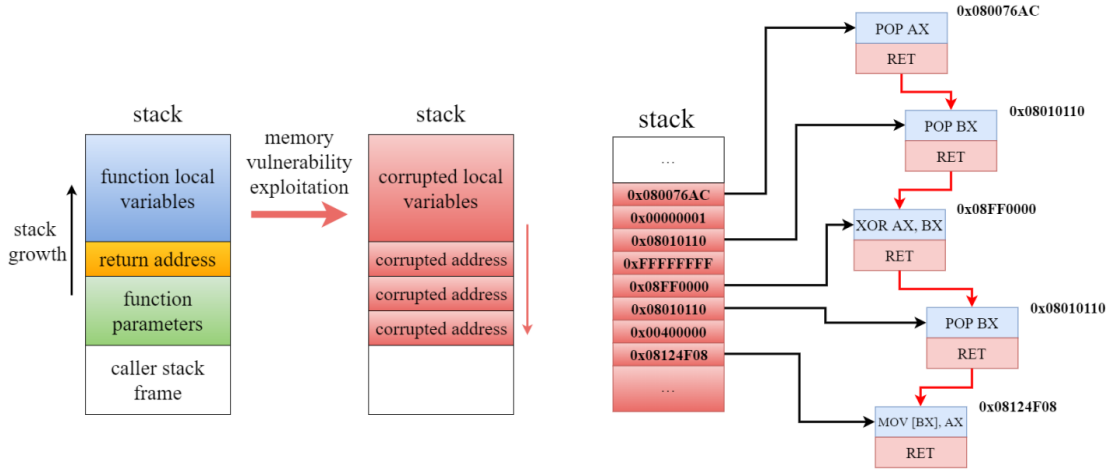
2.2.2 Code Reuse Attacks

As a way to counteract these defences, attackers came up with new classes of attacks that, instead of injecting malicious code into the memory, use the existing code to their advantage. These are generally called Code-Reuse Attacks (CRA).

This extends to the libraries used by the application, giving the attackers a huge code-base to work with. One common example is *return-to-libc* attacks [32], which exploit buffer overflows to redirect the control-flow towards sensitive functions part of the C standard library, possibly passing arguments by pushing them in the stack as well.

Return Oriented Programming

One of the most common examples of CRA is Return-Oriented Programming (ROP) [15] [39] [22] [21] [50]. In this exploit, the attacker finds inside the executable code a series of segments referred to as *gadgets*, made up of very few instructions and all ending with a *return* instruction. Then, he corrupts the stack through a buffer overflow and fills up the memory with the list of gadget addresses. Once the function returns, the gadget chains activates, and one after the other, all gadgets are executed. (Figure 2.1). It has been proven that, for programs with a big enough codebase, it is possible to achieve a Turing-complete execution in this way, i.e., the attacker gets to execute an arbitrary malware on the victim machine. Similar variants of this attack principle have been investigated, such as Jump-Oriented Programming (JOP) [11] [23] or Call-Oriented Programming (COP) [68], which make use of the same concept but using *jump* or *call* instructions instead of *return* ones.



(a) (b)

Figure 2.1: Execution example of a ROP attack. First, the attacker fills the stack with the gadgets' addresses in (a) and then in (b) the attack can be seen unfolding [61].

2.2.3 Mitigations

To counteract stack smashing, both for code injection and for code reuse, several solutions involving the use of a *stack canary* have been studied [28]. This implies inserting in the stack a *canary value* right above the return address. Before performing any return operation, the program checks the canary value against its expected value and will abort operation if there is a mismatch. In this way, when smashing, the attacker has to guess the canary value to have the attack succeed.

Another technique employed is *Address Space Layout Randomization* (ASLR) [9]. Whenever a process is created, the address space positions of several key data areas are rearranged

randomly. This includes randomising the base address of the main memory sections, such as the heap, the stack and the code libraries. The security of this approach depends on the size of the address space. Brute-force attacks can still permit a successful attack, so in the case of particularly small address spaces ASLR could be bypassed with relative ease.

To date, the most promising technique to protect against CRAs is *Control-Flow Integrity* (CFI), which will be described in detail in the next Section.

2.3 Control-Flow Integrity

Control-flow Integrity (CFI) [2] is one of the most effective defence schemes against control-flow hijacking by an attacker. The basic principle behind it is the idea that by checking at run-time every control-flow transfer against a predefined model, it is possible to block any hijacking attempt and prevent the execution of arbitrary code. There are several variants of CFI, which will be discussed along with details on the basic implementation of CFI, in the following.

2.3.1 Main Principles

The main principles behind the operation of CFI are the creation of a static model of our code and then the verification of every control-flow transfer at run-time. These are referred to as the *offline phase* and the *online phase* respectively. In the original implementation of CFI [1], this was done through the creation of a model in the form of a Control-Flow Graph (CFG) and the instrumentation of the code to protect, by inserting special instructions to check the branching operation against the CFG. To do this, *labels* are inserted to identify the code sections that are going to be the possible targets of control-flow transfers, and they are used to recognize the different nodes in the CFG. The operations that are going to be instrumented are the *indirect* control-flow transfers, i.e., the ones with a target address computed at run-time, since they are the only ones that could be tampered with. In particular, these are:

- Branch instructions with a register operand;
- Return instructions;
- Any other instruction that alters the value of the PC register.

The generation of the CFG is done through static analysis of the code or binary: the nodes of the graph are the basic blocks in the source program, and the edges represent the control-flow transfers that will be considered as valid. A good part of these transfers are not actually of interest from a security perspective, being either direct calls or conditional branches, which are not subject to hijacking risks.

Each relevant basic block is identified with a *label*, and every time an indirect branch is about to be taken, special code is inserted to see whether the intended destination is actually contained in the branch operand.

An issue appears when interrupts are considered. CFI works on the premise that all code is static and all control-flow transfers can be represented by a CFG, but interrupts are an exception to this premise. A sudden exception can be raised at almost any point in

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

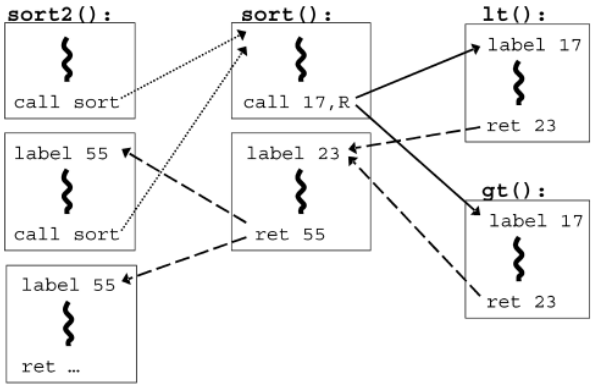


Figure 2.2: Basic example of CFG (on the right) built from the code on the left [2].

the program execution, and will return to that same point. From the CFG standpoint, this can be compared to the appearance of two new, temporary edges that connect the node for the current basic block and the one for the interrupt, disappearing after the interrupt is handled and the program resumes normal operation. *Interrupt Service Routines* (ISR) thus need a way to be made safe from a CFI perspective, for example through dedicated instrumentation at entry and exit points [61].

2.3.2 CFI Variants

Several variants of CFI exist. They mostly differ in their granularity and depending on whether memory of past control-flow transfers is considered or not.

Coarse-grained Control-flow Integrity

One of the main drawbacks of using a complete CFG is the possible overhead. If there are a lot of relevant labels, the CFG can become very large. Another big problem is the fact that obtaining a fully precise CFG is often extremely difficult, if not unfeasible. Complex pointer analysis is sometimes required. For that reason, some solutions instead implement a simplified CFG, that groups nodes with similar characteristics in a single node, which means assigning the same label to a group of basic blocks. How exactly these groupings are made depends on the specific implementation. Unfortunately, such *coarse-grained* CFI policies have been proven to be ineffective in providing adequate protection against attackers [44] [27] [17] [30].

Fine-grained Control-flow Integrity

Because of the security limitations of coarse-grained CFI, *fine-grained* CFI is the preferable choice to adequately protect against malicious actors. In this case, the objective is to provide a fully-precise CFG, complete with every possible valid target of every indirect

branch. A fully-precise CFG needs to be both complete and sound, meaning that it is required to eliminate all false positives and all false negatives, which cannot be guaranteed.

Shadow Stack

In [18], it has been shown how even solutions implementing fine-grained CFI can potentially be subject to ROP and other kinds of control-flow hijacking if the backward branches are treated the same way as the forward branches, meaning that they are only verified against the CFG. This is because there may be some functions, named by the researches *dispatcher functions*, that are called from several places in the code. Having a large number of valid return addresses means that an attacker could potentially use that as a way to string a series of gadgets together to achieve ACE, create loops inside the code and so on, all while respecting the CFG and thus remaining undetected. For this reason the concept of a *shadow stack* was introduced [74] [40] [25] [14] [13]. The idea behind a shadow stack is that when entering a dispatcher function, the correct return address gets saved inside of a secondary protected stack. When the function returns, the return value is compared to the one on top of the shadow stack to make sure it is still the same and no control-flow hijacking was attempted. There are both hardware and software implementations of shadow stacks. The downside of this approach is that there is an overhead imposed by writing and reading from the shadow stack.

Stateless vs. Stateful CFI

The basic implementation of CFI is *stateless*, meaning that previous operations are not tracked and the checks against the statically-generated CFG are the only way to verify the validity of a control-flow transfer. The addition of a shadow stack already introduces an element of dynamic verification, but is only applicable for return instructions. Calling functions can still be considered as dispatcher functions, if a high enough number of valid targets exist for the same indirect transfer. In that case, *stateful* solutions [73] which, for example, record a number of past transfers to analyse if a valid path inside of the CFG is being taken or not, could be applied. The problem here is that they still need enough information to be reliable (e.g., they are still vulnerable to attacks during the early stages of recording) and will most likely generate significant overhead. For these reasons, the most interesting application of CFI still remains fine-grained CFI with a shadow stack, offering good security with acceptable overhead, both for performance and complexity.

2.3.3 Hardware-assisted CFI with ARM Pointer Authentication

With ARMv8.3, *Pointer Authentication* (PA) [12] has been introduced. This new feature adds an hardware-assisted way to sign pointers with a unique code, that will be used for authentication before consuming them. This code (called Pointer Authentication Code or PAC) is generated using the QARMA algorithm [6], which takes (i) the pointer, (ii) a key, and (iii) a modifier to generate a value that will get truncated and inserted in the unused bits of the pointer. This is possible since 64-bit pointers are oversized for the address space they usually refer to, and thus have unused bits. There are 5 different keys that can be used, and the modifier depends on the developer's choice.

In 2017, a white paper by Qualcomm [46] introduced the idea of using PA for CFI, by possibly using context and/or location information as modifiers for signing function pointers at run-time. PA is resistant against memory disclosure attacks, since the PAC is generated through a cryptographically strong algorithm, meaning that obtaining the keys from the generated codes is infeasible. The paper introduced the possible issue of *pointer substitution*, which is the most likely kind of attack to be used against PA. This attack works by substituting an authenticated pointer with another authenticated pointer, which could be obtained for example through a memory disclosure vulnerability. To combat this, the use of appropriate modifiers can be of help, for example by using the local context, as a way to make sure that authenticated pointers coming from different contexts could not be exchangeable. The issue of possibly repeating contexts is still present, and the authors talk about this by describing how an application using the stack pointer as context for return addresses could be exploited to collect several authenticated pointers for different values of the stack pointer and reuse them in the program to string together a series of gadgets to execute a ROP attack. Therefore, accurate and smart choices need to be made when deciding the modifiers for the PAC generation, to reduce as much as possible the threat of pointer substitution.

Another possible attack could be to just try to guess the PAC through brute force. The feasibility of this depends on the system, since the size of the PAC depends on how many unused bits are leftover in the pointer, which depends on the addressing space size. It could go from as low as 3 bits (a 1 in 8 chance) to up to 31 bits, so some consideration needs to be done if one wants to use PA.

2.4 Embedded Systems Issues

As already mentioned, the above security needs have to deal with the peculiarities of embedded systems, which typically have a limited amount of resources, and are originally designed to meet low consumption requirements or dependability on the timing of execution. In this Section, some glimpses of the characteristics of the embedded systems to be protected are presented.

2.4.1 Real-Time Operating Systems

Quite often embedded systems are required to meet certain timing deadlines. This is especially true of system controlling complex and safety critical physical systems, like an industrial control system or an aeronautical one. In these cases, it is necessary to develop *real-time* platforms, which are supposed to perform certain operations in a predictable amount of time to respect certain deadlines. The various operations that can happen are divided into *tasks*, and every task has to produce a result inside of a set deadline. There are three main categories of real-time systems:

- **Hard real-time**: the system needs to respect the deadlines, a failure to do so means a failure for the whole system and can potentially have disastrous consequences. For example: automotive braking system not stopping the car in time.

- **Firm real-time**: if a deadline is missed the result is worthless, but the system keeps working and no grave consequences are expected to happen. This translates to just the single task failing and not the whole system.
- **Soft real-time**: if a deadline is missed the result can potentially still be useful, but generally its usefulness degrades over time after the deadline is missed.

Sometimes the distinction between firm and soft real-time is not made and real-time systems are just divided into hard and soft real-time.

In order to make the development of real-time systems easier, several specific operating systems have been created, introducing the concept of *Real-Time Operating Systems* (RTOS). These can vary depending on their complexity, hardware requirements, scheduler policies and so on.

Regarding the scheduler, the most important difference between them is if they are *preemptive* or *non-preemptive*. In the first case, the tasks can be switched out for a different one at any point in time, while non-preemptive systems need the tasks to do a *yield* operation to give control back to the scheduler. The non-preemptive case, while limiting, can be a good option for use cases that need a very predictable/deterministic behaviour. Scheduling tasks in a way that always follows predictable timings becomes significantly easier. There exist a lot of different scheduling policies: some assign a priority to the tasks and order their scheduling based on that (*priority-based scheduling*), some others instead opt for a simpler approach like just scheduling them depending on the time they request to be executed (e.g., First-In-First-Out policies), or they can be all just given a fixed amount of time and be switched out periodically (Round Robin policy). If it is possible to predict the duration of the tasks, more complex policies are possible as well.

Linux itself can be made into a real-time system with acceptable timings by using tools like Preempt RT [37] or LITMUS^{RT} [16] (respectively a real-time patch and an extension to the standard kernel). As one would expect, though, this is not advisable for severely resource-constrained systems, because of the large codebase and relatively large memory requirements, when compared to more minimal solutions. Lightweight alternatives exist, a few examples being RIOT [7], Micrium's µC/OS-III [51] and FreeRTOS. The last one is the one that is going to be considered for this proposal, so a closer look at it is provided.

FreeRTOS

Among the various existing lightweight RTOSes, one of the most popular is FreeRTOS [41]. FreeRTOS is an open-source RTOS made especially for embedded applications, managing to be both light on resources and feature-rich, with a very active community behind it. Depending on the configuration, the minimum requirements can go as low as 10MHz processing speed, 16kb of RAM and 128kb of program Flash.

The FreeRTOS scheduler is a preemptive scheduler implementing priority-based scheduling, meaning that whenever a task wants to execute, it checks if the currently active task has higher or lower priority, and preempts it to do a context switch immediately if it is a lower priority task. An example can be seen in Figure 2.3.

A Task Control Block (TCB) is assigned to each task. This contains the task's priority, stack pointer, stack size along with other information necessary for its usage. Task state (*Idle*, *Running*, *Ready*, *Suspended* or *Blocked*) is also saved in the TCB. When the scheduler

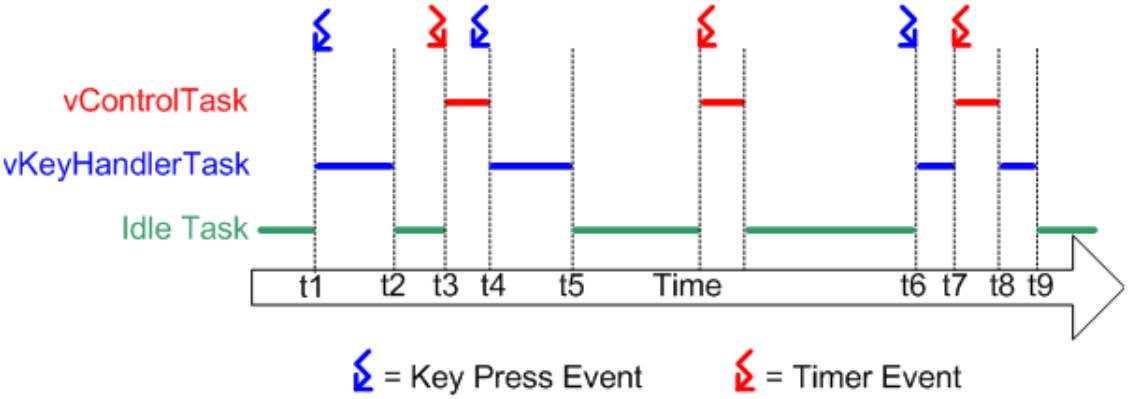


Figure 2.3: Example of task handling by the scheduler. vControlTask is the highest priority task, which is why when a key press event happens between t3 and t4 vKeyHandlerTask still has to wait for vControlTask to finish, and at t7 vControlTask preempts vKeyHandlerTask [42].

is started the *Idle* task is created, which is a lowest priority task put in place to make sure that at least one task is always running; this task will be preempted by any other incoming task. The only active function this task has is to free the memory previously allocated to now deleted tasks. Each task is assigned dedicated stack space, while the heap memory is shared among all tasks.

Whenever a context switch happens, the scheduler has to save the context of the running task before switching in the new task. This means saving all the registers and the Program Counter (PC), to be able to return to the right point in the code. The scheduler does this by pushing all of this onto the task's stack, saving a copy of the stack pointer into the kernel and using that to restore context when the task will be scheduled again, by simply retrieving the registers in order and putting the correct value into the PC register before resuming operation. A visual summary of these operations is represented in Figure 2.4.

2.4.2 Hypervisors in the Embedded Domain

Hypervisors are a powerful tool for OS isolation and managing multiple OSes on a single platform. Their use in the embedded domain is still fairly limited compared to general-purpose systems, often because of the overhead usually imposed by these tools. There exist two main categories of hypervisors:

- **Type 1 hypervisors**: known also as *native* or *bare-metal* hypervisor, they run directly on top of the hardware. One notable example could be Xen [36].
- **Type 2 hypervisors**: known also as *hosted* hyperviors, they run inside of an Operating System. The hypervisor will then run the wanted OSes as guests. QEMU [8] and VirtualBox are a couple notable examples. Some solutions that are kind of a hybrid between type 1 and type 2 also exists, like KVM [49] (which works as a kernel

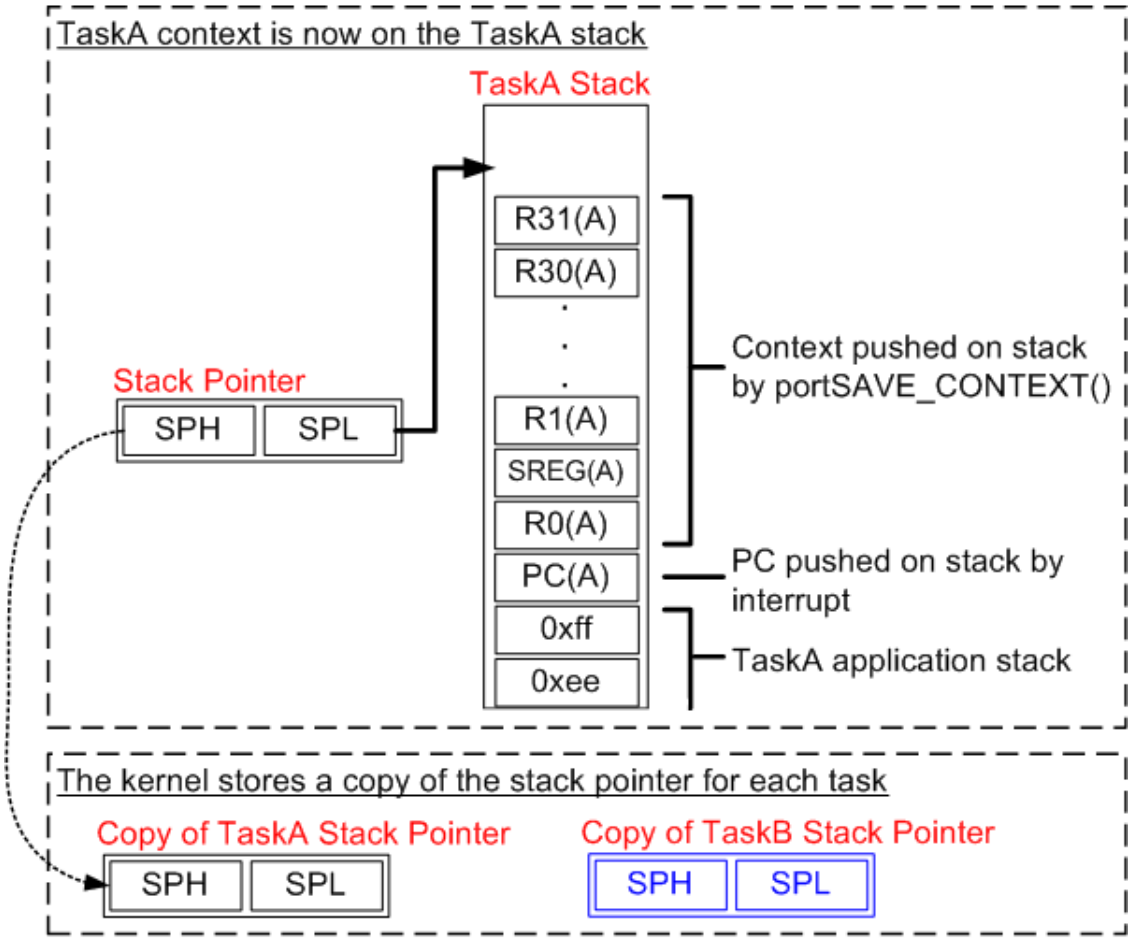


Figure 2.4: Scheme exemplifying how the context is saved in the task's stack whenever a context switch happens [43].

module "converting" the host kernel into a type 1 hypervisor), but they are usually categorised as type 2 hypervisors since they still need the support of an operating system to work.

Some of the hypervisors for general-purpose systems also support use on embedded systems, but there are some limitations in that sense. Both Xen and KVM have been expanding their support for ARM platforms in recent years, but in real-time applications while KVM provides acceptable overhead, Xen brings a much more significant performance penalty [4]. At the same time, KVM is limiting as well because it still requires the Linux kernel to run, with its impact on resources.

There are however some embedded-specific hypervisors out there. They usually have stricter requirements when it comes to performance and size overhead, both because of the limited resources and the support for real-time computing. Hypervisors also need some kind of memory protection functionality, which rules out a lot of very basic embedded

microcontrollers, but makes them still usable on platforms that, even if lacking a full Memory Management Unit (MMU), at least have a Memory Protection Unit (MPU) for enforcing memory protection and the possibility of definition of a privileged and user mode. Most modern ARM systems support these functionalities, which means that embedded hypervisors still have a wide array of possible applications. A notable example is PikeOS [45] [48], which is an RTOS providing the support of a type 1 hypervisor, making it possible to run several different systems each with a different level of security and reliability. It has been developed with high-safety, high-security applications in mind, trying to abide by the various certification needs for application in the fields of defence, automotive, medical, etc. The matter of certifications is very important in a lot of fields, for example automotive, where compliance with standards (e.g., MISRA) is often required. All software running on automotive vehicles has to adhere to these standard and this is usually done thanks to the use of static analysis tools. For example, Xen tries to achieve MISRA compliance by using Helix QAC [38].

Bao

In [59], Bao is presented. The aim of the researchers was to create a lightweight embedded hypervisor with real-time use in a mixed-criticality environment in mind. Bao is a type 1 static partitioning hypervisor, meaning that the hardware resources are allocated statically by the hypervisor to each guest once before they boot up (Figure 2.5). The interesting characteristic of this hypervisor is its minimal approach that results in low overhead while providing solid security guarantees. It also has ways to allow inter-VM communication between the different guest Virtual Machines (VM), through the use of interrupts (that can be used as a doorbell mechanism) and the possibility of defining a shared memory space. The guest VMs can contain bare-metal software or host a variety of OSes, both real-time and general purpose. The architecture support at the moment is limited to ARMv8 platforms and RISC-V. Every VM only sees what the hypervisor exposes it to, usually in the form of virtualised hardware, such as virtual CPU cores and interrupts, which are directly mapped to the physical ones.

The reason this tool has been chosen for the theoric implementation of our solution is the ease of use, availability and perfect fit for low resource mixed-criticality application. It is an open source project that is already available on GitHub [66], with a constantly expanding list of supported platforms.

Use in Mixed-criticality Systems

One of the most interesting usecases for embedded hypervisors is their application in mixed-criticality systems. This has become increasingly common thanks to the evolution of embedded platforms, now able to handle the functionality of a wide variety of OSes while at the same time providing stronger security and memory protection functionalities. This has made it possible to centralize the operation of something like the control and infotainment systems of a vehicle on the same platform, which means easier planning and a possible cost reduction for the manufacturers. The availability of powerful and inexpensive multicore SoCs has thus made the need for appropriate hypervisors all the more relevant. As said before, PikeOS is one of the commercial tools created with mixed-criticality in mind, and

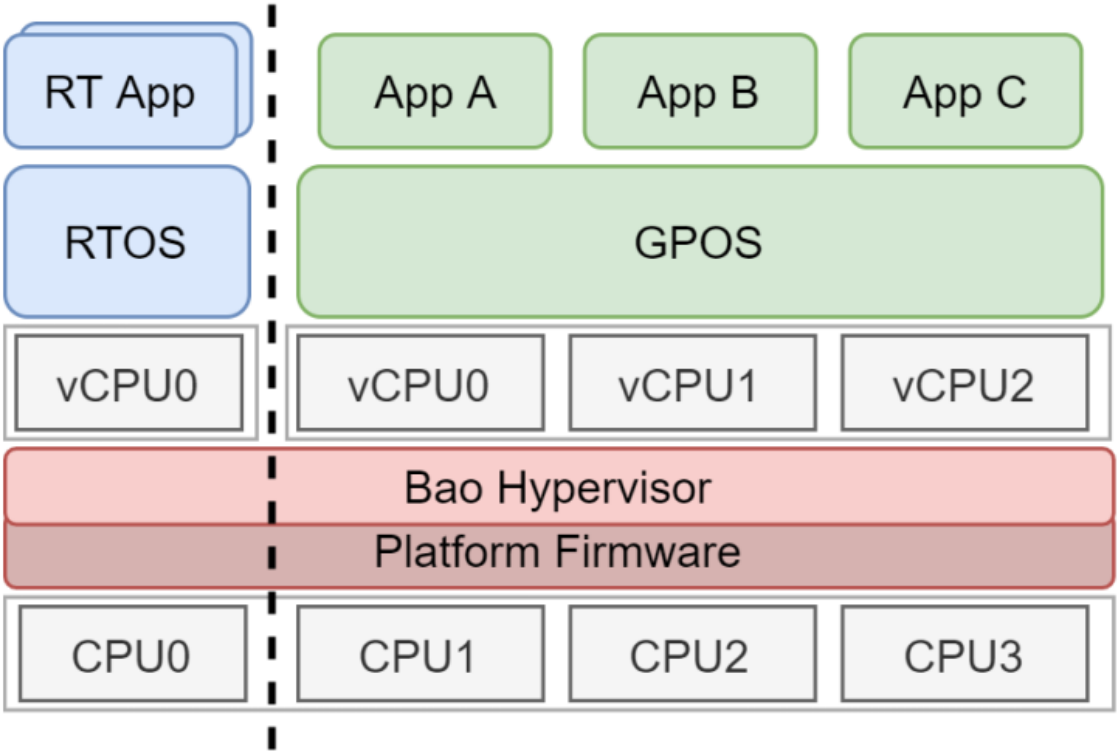


Figure 2.5: Hardware partitioning for a system with a dual-guest configuration, running an RTOS along a general-purpose OS [59].

Bao is an interesting project with a similar mission that could gain traction in the future, thanks to its open nature.

Chapter 3

Current State of the Art and Open Issues

At the moment, the available solutions for CFI in the RTOS domain proposed in the scientific literature are quite limited. During the work for this thesis, a statement paper by me and my group has been accepted: *Control-Flow Integrity for Real-Time Operating Systems: Open Issues and Challenges*. This work was presented at the *IEEE East-West Design and Test Symposium 2021* [72] conference in September of 2021, and will be published and become publicly available soon. The paper analyses the proposed software-based CFI solutions for RTOSes or approaches that seems to be most promising, if they were appropriately modified. First, the basic characteristics of an ideal solution were defined, and then used to analyse several works. A few of them have been selected on the basis of the presence of interesting features, to have a clear picture of the current state of the art in this field. Also, the issues that are still unsolved and how to possibly improve on them are outlined.

3.1 Characteristics of an Ideal Solution

The requirements of a security solution for a RTOS are different from the ones of a general-purpose system, since the impact on schedulability has to be addressed as well. Thus, for our analysis, our *ideal solution* is defined as having these characteristics:

- **Complete CFI coverage**, which is comprised of:
  - **Full forward and backward branches protection**;
  - **Protection of the interrupt context**, which is not predictable from the CFG analysis only [60].
- **Uncompromised workload schedulability**: this means having ideally negligible overhead. This is evaluated depending on the specific system and workload. Most embedded platforms running real-time applications are usually resource-constrained, which makes this a significant issue. Overhead should be considered both for performance and code size.

All the examinated solutions incurred in the trade-off between security and overhead, although in different ways and to different degrees. One or the other is usually given priority, at the cost of the other one, making obtaining a complete, all-encompassing solution still an open issue. A deeper look at the various solutions examined is below.

3.2 Current State of the Art

From the available research, five solutions have been selected, as they were the most relevant and interesting at the time of the paper's writing.

3.2.1 RECFISH

In [76], a solution called RECFISH is proposed. This solution is a RTOS-specific solution that implements fine-grained CFI with a shadow stack by taking advantage of the MPU functionalities of the ARM platform. Support for use with FreeRTOS was also introduced by making appropriate modifications to its code.

CFI scheme

The CFI validation is implemented through binary instrumentation for forward branches and the use of a shadow stack for backward branches. A CFG is generated through static analysis of the binary. The solution chooses to use binary and not source code instrumentation in an effort to be able to retrofit this to existing devices for which the source code may not be available. Forward branch protection works in a traditional way, by using labels to identify each section of code and doing the appropriate comparisons with the CFG. Since this is done on the binary and not the source code, to make sure there is no problem regarding relative addressing because of instruction insertion, a separate memory section **.cfi** is dedicated to the CFI instrumentation, and the original instructions inside the **.text** section are replaced with trampolines (i.e., direct branches). For each existing function, the instrumentation adds trampolines to an instrumented prologue, branch operation (when present), and epilogue, all inside of their dedicated memory (Figure 3.1), which correspond to the original prologue, indirect branch and epilogue of the instrumented function.

- **Instrumented Prologue**: in this section, the registers normally pushed on the regular stack are still pushed there with the exception of the link register, which is instead pushed onto the shadow stack. The shadow stack is accessed through a supervisor call.
- **Instrumented Jump**: here the actual CFI check for forward branches happens, by comparing the expected label with the target of the indirect branch.
- **Instrumented Epilogue**: before returning, the previously pushed registers are popped from the regular stack, and the link register is retrieved from the shadow stack. Finally, the program branches to the address inside the link register.

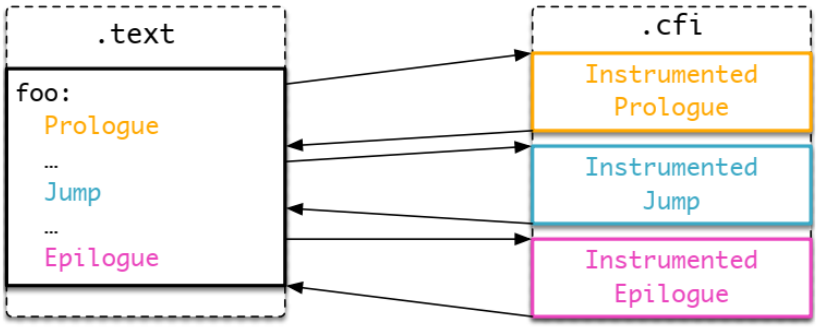


Figure 3.1: Instrumentation through the use of trampolines [76].

FreeRTOS Modification

In order to make use of these functionalities in conjunction with FreeRTOS, some modifications were applied to the RTOS. The presence of multitasking/multithreading and context switching makes some adjustments necessary. In RECFISH, the adopted solution is to save the context not in the regular stack (as described in the last paragraph of subsection 2.4.1) but in the shadow stack. Since the scheduler already runs in privileged mode, no overhead from the supervisor call execution occurs. To do this, three main modifications were implemented:

- **Task Control Block extension**: the TCB was modified to add an extra field for the shadow stack.
- **Task creation modifications**: the functions handling the creation of a task were modified, adding the assignment of a shadow stack region to each task along with the changes necessary to interact seamlessly with the shadow stack.
- **Scheduler modifications**: to be able to use the already existing instructions for context saving and restoration, the necessary changes to make those point to the shadow stack were put in place.

Issues

This solution differs from our ideal solution on two points:

- **Protection of the interrupt context**: there is no protection for interrupts, which run in privileged mode and are thus a possible vulnerability. The researchers consider this to be a non-issue since usually real-time interrupts are very short and could be designed to avoid having any memory writes. This imposes limitations on the design of interrupts, and is especially a problem when retrofitting already existing binaries.
- **Schedulability**: the researchers tested a wide array of workloads, with mixed results depending on their composition. Around 15% of workloads were not schedulable

anymore after the instrumentation, with an overhead of 30% for the worst cases. A proposed mitigation to this issue was the possibility to mark tasks as *safe* or *unsafe* and just apply CFI checks to the unsafe ones. This has proven to be a successful strategy in their tests, but introduces the new problem of needing to either mark by hand the tasks (introducing human error in the mix) or create a tool able to analyse and mark the tasks automatically.

3.2.2 TrackOs

TrackOS [64] is a RTOS based on FreeRTOS that implements a CFI scheme with schedulability and time determinism at its core. It generates a CFG from static analysis of the binaries and stores it in program memory. A monitoring task is created and scheduled along with all the other "normal" tasks. The CFI checks, instead of happening at every vulnerable control-flow transfer, happen only when scheduled, depending on the workload definition.

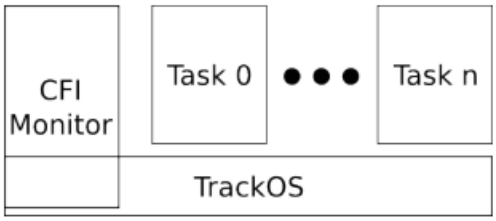


Figure 3.2: View of the tasks running on TrackOS. The monitoring task is represented as overlapping TrackOS to represent the privileged mode it runs in [64].

The way the monitoring task works is by analysing the stack of a task (or several tasks at once) and comparing it with the CFG to detect any violation. To be able to do this, the task is allowed to run in privileged mode, since it needs to access others tasks' stacks (Figure 3.2). This approach mostly solves the schedulability problem, because the monitoring task can be scheduled in such a way to avoid missing deadlines.

Issues

While TrackOS does provide security improvements with potentially no compromise on workload schedulability, it is still not a completely secure solution. As long as a task is executing, no control flow violation will be detected as long as no context switch to the monitoring task happens. If an attacker were able to also clean up before returning to the scheduler, no trace of the attack would be detected. A possible approach to solve this would be to instrument all the monitored tasks with yield instruction before an indirect branch instruction, but that would completely defeat the whole point of TrackOS's approach, making it just an even more inefficient standard CFI solution. Furthermore, with the monitoring task being, in the end, just a task like the others, the possibility of task starving

exists in the presence of higher priority tasks in the workload. It is not always possible to put the monitoring task as the highest priority one, as shown in the testing done by the researchers. They tested TrackOS on a drone using ArduPilot, an open source autopilot. In this case, there are several tasks handling communication and controls that need to have higher priority than the CFI task, making it possible to starve the task indefinitely.

3.2.3 FINE-CFI

Among the solutions for general purpose operating systems, FINE-CFI [53] has an interesting approach. The purpose of FINE-CFI is to secure an OS kernel with CFI. Full fine-grained CFI is implemented, along with protection of the interrupt context. For the last part, virtualisation is employed.

CFI scheme

As the name implies, this solutions implements fine-grained CFI, in contrast with similar past solutions like KCoFI [29]. Both forward and backward branches are handled using *indexed hooks* [54], which is a CFI scheme proposed by the same authors of FINE-CFI. With this approach, a jump table and a return table are generated for handling forward and backward branches, respectively. Then, through instrumentation, the program is modified in such a way to use the indexes of said tables instead of the actual addresses, making it impossible for an attacker to jump to a certain part in the program by writing an arbitrary address in the stack, because now only indexes are used, making it possible to only use allowed addresses.

For the CFG generation, an algorithm using a particular data structure called a *struct location vector* is implemented. The struct location vector of a function pointer is used to identify the position of a function pointer inside of a nested struct. This information is then used to implement the complex pointer analysis needed to generate the fine-grained CFG of the kernel, complete with all the valid point-to sets for each function pointer or return operation.

Interrupt context protection

In FINE-CFI, the interrupt context is protected thanks to the use of KVM. The kernel to be protected is running inside of a VM and all the incoming interrupts have to go through the hypervisor first. The hypervisor keeps an interrupt stack where it stores the Code Segment/Instruction Pointer (CS/IP) couples for each interrupt. This memory, being managed directly by the hypervisor, cannot be compromised by an attacker. The order of action whenever an external interrupt arrives is as follows:

1. The interrupt is raised
2. The hypervisor receives it and saves the CS/IP couple in its interrupt stack
3. The hypervisor sends the appropriate virtual interrupt to the VM
4. The VM executes the interrupt handler

5. Before returning with the **iret** instruction, it uses the **vmcall** instruction to exit the VM and execute an hypercall handler
6. The hypervisor compares the CS/IP pair in the kernel stack with the one in its interrupt stack
7. If they are the same, the topmost element in the interrupt stack gets popped, otherwise an error handler is executed

A visual representation of this operation can be seen in Figure 3.3.

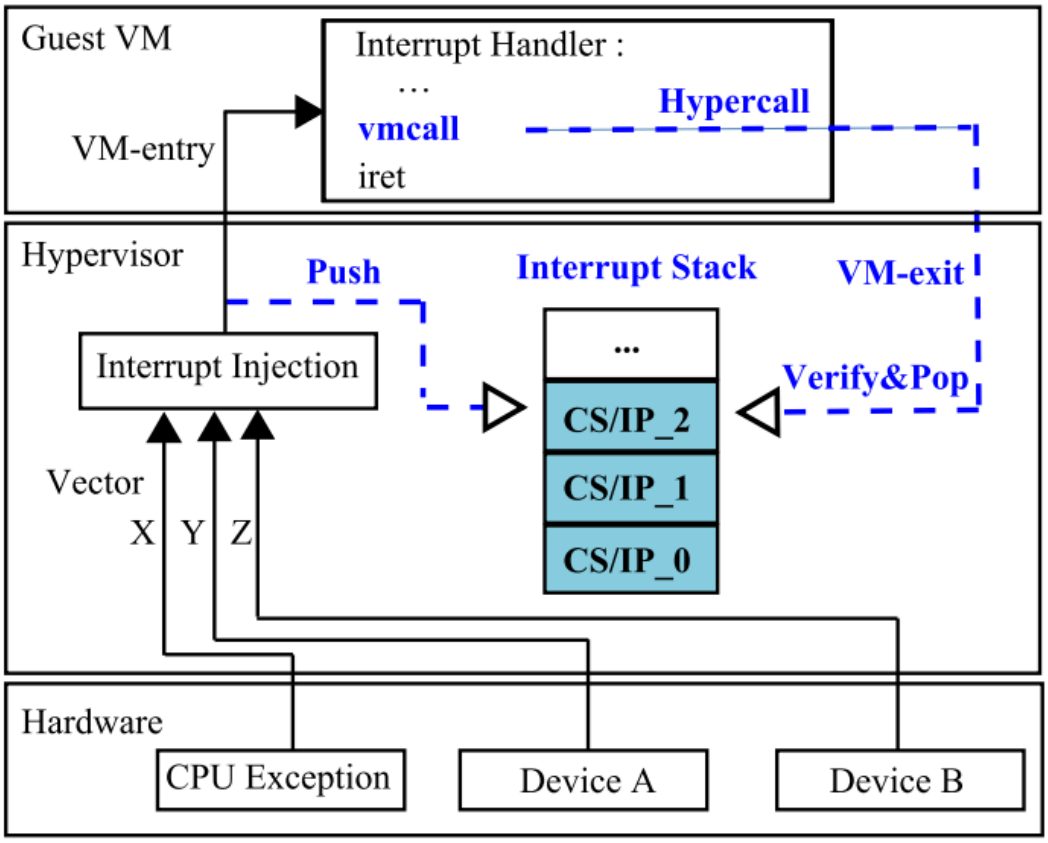


Figure 3.3: Model of the hypervisor intercepting incoming interrupts and saving the CS/IP couple in its interrupt stack before passing the interrupt to the VM [53].

Issues

The use of virtualisation adds a powerful ally to the mix, but since this is a solution for GPOSes there are some considerations to be done if one wants to adapt this to the

embedded world. While KVM has gained much more support for operation on ARM platforms in recent years, the problem of needing the Linux kernel to operate is still a potential problem. Not only is there a considerable size overhead, but the introduction of a very large Trusted Computing Base is also worrying. The decision not to use a shadow stack but instead depend on indexing hooks also introduces the problem of the exploitation of dispatcher functions as described in 2.3.2.

There is also the issue of compromised schedulability. The researches tested several benchmark suites, and found average overheads going from 7.46% to 9.89% depending on the suite, with several single benchmarks going over 15%. It is not clear how this would translate for real-time workloads, but it is still a non-negligible amount of overhead that could potentially have an impact on schedulability.

3.2.4 Camouflage

Among the analysed solutions, two are making use of ARM Pointer Authentication (described in 2.3.3). The first one is Camouflage [31]. This is also a solution for securing kernel code. PA is leveraged to sign function pointers and return addresses. In particular:

- **Forward branch protection**: forward branches can appear in two forms, either as regular function pointers or as function pointers inside of static *operations tables* in the form of structures, which are stored on read-only memory. While in the last case the function pointers themselves are not at risk of being modified, the pointers to the operations structures themselves could be, and thus need protection. The modifier used for the generation of the PACs for both cases (function pointers and pointers to operations tables) is comprised of 16 bits identifying the combination of the containing type and compound type member, together with the 48-bits address of the containing object. Appropriate setter and getter functions are used to implement signature and authentication of the pointers
- **Backward branch protection**: to avoid the vulnerability to replay attacks that the original implementation of PA had, thanks to possibly repeating Stack Pointer (SP) values, a more complex modifier is used here. To protect backward branches, functions are instrumented with a prologue and epilogue. In the prologue the LR is signed using as a modifier the low order 32 bits of the SP concatenated with the low order 32 bits of the function address.

Camouflage also uses an hypervisor to protect the keys used for PA, by only using them through functions inside of a XOM (eXecution Only Memory) at the kernel level, which interacts with the hypervisor in a way that makes sure the keys are never exposed. The keys are generated by the bootloader which then updates the kernel functions before booting. The setter in the XOM will do its work all before interrupts are enabled, to stop possible key leakage, and all the registers involved are zeroed out before starting normal operation.

Issues

Camouflage has some issues that need to be solved, especially if one wants to use its approach in a real-time system.

From the benchmarks ran, the overhead goes from 10% to 30%, which the researchers argue is not reflective of the actual impact on normal operation, since the instrumentation is on the kernel and the use of system calls is usually fairly limited. This means that, in the case of a RTOS, depending on system call usage in the workload, the overhead could go from moderate to very significant. This also poses a problem if someone were to try to apply the same approach not only to kernel code but to the tasks as well. There is also the issue of the heavy involvement of the hypervisor, which handles all the key operations, meaning extra overhead for the system.

There is also not protection of the interrupt context. The authentication and branching are not done as an atomic operation, but are instead executed by separate instructions, creating the possibility of time-of-check-to-time-of-use (TOCTOU) [3] attacks.

As a last point, Camouflage needs some modifications to the C library functions to function properly, thus not achieving ISO C compliance, which could be an issue for applications that need to adhere by certain standards and certifications, which is commonplace in several safety-critical systems.

3.2.5 PATTER

The second solution using PA for CFI that was analysed is PATTER [79], which stands for **P**ointer **A**u**T**hen**T**ication for k**ER**nels. As the name implies, this is another solution that aims at introducing CFI to the kernel of an operating system. PATTER works together with Clang/LLVM [52] to insert instrumentation right at the Intermediate Representation level and protect both forward and backward branches. There are two main phases (Figure 3.4), the one at compile-time and the one at run-time:

- **Compile stage**: the kernel is first compiled to LLVM IR (Intermediate Representation) with Clang. At this point static analysis is ran on the kernel IR and all function pointers are identified. After this, instrumentation for forward and backward branches is inserted:
  - **Forward branches**: for forward branches, the modifiers is chosen after making an observation. All the function pointers in memory are stored in the same address space, which means that their relative address can be used as the modifier for their signing and subsequent authentication, making a pointer substitution attack very difficult to pull off. Past solutions used the pointer type, but that was significantly more likely to leave openings for pointer substitution. All pointers are signed and then stored in memory, and when they need to be used they are loaded from memory and then authenticated. When branching, the **blraa** instruction is used to guarantee atomicity. That instruction authenticates the pointer before a branch and link, making TOCTOU attacks impossible to be executed.
  - **Backward branches**: for return addresses, the stack pointer value is used as the modifier for PAC generation. Prologues to sign the return value are put at the start of each function, and to return the atomic **retaa** is used, to authenticate and branch in a single time to stop the possibility of TOCTOU attacks.

- **Run-time stage**: at run-time, several operations happen in order. During kernel boot-up, PA is initialised, the keys and registers are set. Then the case of statically initialised function pointers is handled, they are signed and saved in memory. There are also some dynamically assigned function pointers that needed to be first initialised before the PA initialisation, so those are signed as well. After this, the booting operation is complete and PATTER can protect the kernel during its normal operation.

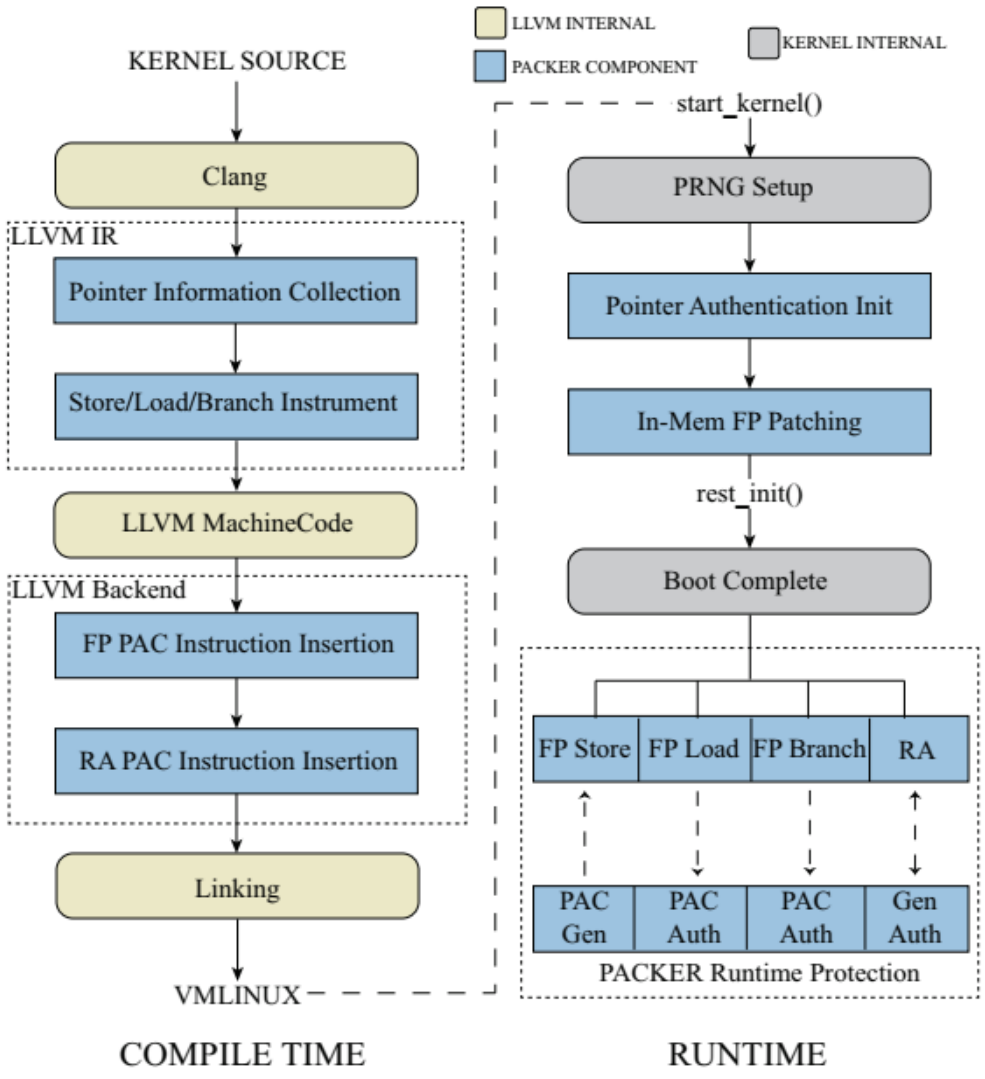


Figure 3.4: Schematic representation of the two stages of PATTER protection [79].

The authors created their own algorithm for function pointer detection and classification, which manages also the problem of pointer propagation with some changes to the

PAC generation.

Issues

The approach of PATTER cannot completely stop possible TOCTOU attacks, since at the moment there are no instructions to atomically sign and store or load and authenticate. This is not something that can be solved by the authors themselves, which argue that it is an hardware limitation, and also point out how correct authentication will still be needed before any branching operation, strongly limiting the possible impact of these attacks.

There is also the problem of the possible presence of several corner-cases regarding function pointer generation, like the use of pointer arithmetic, pointers holding physical addresses, and pointers inside of unions. All of these cases are not correctly handled by their algorithm, so they need specific workarounds. The researchers implemented the solution for the Linux kernel, and in that case these occurrences were fairly rare, making it possible to patch manually the first two cases and use a simple protocol for the last one. In particular, they used the 64-bit alignment property of pointers as a way to recognize the element type for the case of function pointers inside unions, which works fine for the Linux kernel. These workarounds are not a structured solution that can necessarily be applied to every kernel, so depending on the OS it would be ideal to have a more solid approach.

Regarding overhead, the researchers decided to use Unixbench to run several tests. The system call intensive ones noticed overhead from 10% to 20%, while some user-space tests denoted a much more acceptable overhead. This is a similar situation as Camouflage, where the kind of workload on a RTOS could significantly influence the overhead depending on the system call usage frequency, and possible adoption of this CFI scheme to the task code itself could result in some very significant overhead.

Solution	**F&B coverage**	**Int. awareness**	**Schedulability**
RECFISH [76]	Yes	No	Compromised
TrackOs [64]	No	No	Uncompromised with appropriate scheduling
FINE-CFI [53]	Yes	Yes	Compromised
Camouflage [31]	Yes	No	Compromised
PATTER [79]	Yes	Yes	Compromised

Table 3.1: Summary of analysed solutions and their coverage of the main issues.

3.2.6 Open Issues

To summarise the results from our analysis, there are still no real software-based solutions to the problem of CFI for RTOSes. A summary of the results of our analysis with respect to the metrics defined for our ideal solution is presented in Table 3.1. There is not enough work on RTOS-specific solutions, and even solutions that could possibly be adapted to the real-time domain still have some issues that could limit their usability. The trade-off between security and performance is always present, and while that may be acceptable for general purpose systems, the desired solution for RTOSes is something that could be

able to introduce minimal overhead (as to not impede the schedulability of workloads with very tight deadlines) but without sacrificing security. The only solution that solves the schedulability problem, TrackOS, has security flaws that in the author's opinion are not considered acceptable. Unfortunately, solving this problem is not an easy task, and some compromises on the possible applications may need to be made. In the next Chapter it will be introduced our proposal for a solution in the case of multicore, mixed-criticality systems, that aims to solve both the security and overhead issues of other solutions at least for this class of systems.

Chapter 4

Proposed Solution for Mixed-criticality Systems

Taking into account all the analysis of the existing state of the art done in the previous Chapter, our proposal can be advanced. A software solution that can respect all the requirements described in 3.1 is unfortunately difficult to obtain, if not impossible, depending on the system. For that reason, some research was done to find a subset of systems for which a compliant solution could be implemented. A possibility was found in *multicore* systems running *mixed-criticality* workloads. The idea came partly from the works using hypervisors for CFI protection analysed in the previous Chapter, and from the principle behind the work described in [61], which will be briefly described in the following Section. Several notions regarding this solution have been taken from the thesis of Valentina Forte, *Automatic Binary Analysis and Instrumentation of Embedded Firmware for a Control-Flow Integrity Solution* [35], who developed the static analysis tool for this solution together with Prof. Prinetto and his team. This work goes more in depth regarding the implementation details compared to the paper alone, which is why it was a very valuable support when writing this Section.

4.1 The FPGA Monitor Idea

In the paper *A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems* [61], authored by Prof. Paolo Prinetto and his PhD students Gianluca Roascio and Nicolò Maunero here at Politecnico di Torino, a clever solution to the CFI problem is presented. By using an FPGA present on board, all the CFI checks can be demanded to it instead of having the processor execute all the needed operations. This makes it possible to incur in a much smaller overhead while having solid security, guaranteed by the isolation of the FPGA. This solution is able to handle security for bare-metal application by protecting both forward and backward branches, while at the same time taking care of the interrupt context. In our proposed solution, some additions and changes needed to be made to adapt these principles to operation in the presence of an RTOS, but it is beneficial to take a look at the inner workings of the existing solution.

4.1.1 Basic Model

There are three 3 building blocks of this solution: (i) the binary instrumentation, (ii) the static analysis tool, and (iii) the architecture synthesized on the FPGA.

Binary Instrumentation

The instrumentation is applied only to select branches which are considered vulnerable, and to interrupt routines. The instrumented forward branches send information to the FPGA depending on their classification.

The classification is made on a few factors, which are:

- whether the forward branch is *secure* or not. In particular:
  - a branch is *secure* when it is a direct branch or an indirect branch whose target(s) depends on a value that never enters data memory;
  - a branch is *insecure* if it is an indirect branch whose targets(s) depends on a value coming as combination from vulnerable memory which could be modified through memory vulnerabilities [61];.
- whether the forward branch has a single or multiple possible targets.
- whether the forward branch's target has multiple or a single return target.

Depending on the various combinations of these factors, the appropriate functionalities have been implemented in the FPGA, and the binary is instrumented both at the source and the destination(s). Backward branch instrumentation also has appropriate handling by the CFI monitor and are instrumented appropriately.

To provide protection of the interrupt context, the register saved automatically by ARM are also saved inside of a secure data structure in the FPGA every time an exception is triggered and then retrieved before returning to the normal operation, as way to preserve the program context. The registers interested by this are register `R0` to `R3`, `R12`, `LR`, `PC`, `xPSR` [56].

Static Analysis Tool

To conduct the static analysis on the binary file, a Python script has been developed. It goes through several stages, by first disassembling the binary file, parsing it to recreate an assembly file, then examining it creating the CFG. It then analyses all the paths traversed by the values of indirect branch targets and based on that classifies the edges on the CFG. Finally, based on these classifications, the code instrumentation is applied.

FPGA Monitor Architecture

The design of the FPGA is comprised of a central Control Unit (CU), which handles appropriately all incoming requests, and three data structures, used to support the CFI operations:

- **Secure Edge Table**: a table containing the information about the CFG needed to enforce CFI;
- **Secure ID Stack**: used to implement shadow stack functionalities for function calls;
- **Secure Register Stack**: a stack used to store all the registers needed to protect program context in case of an ISR execution.

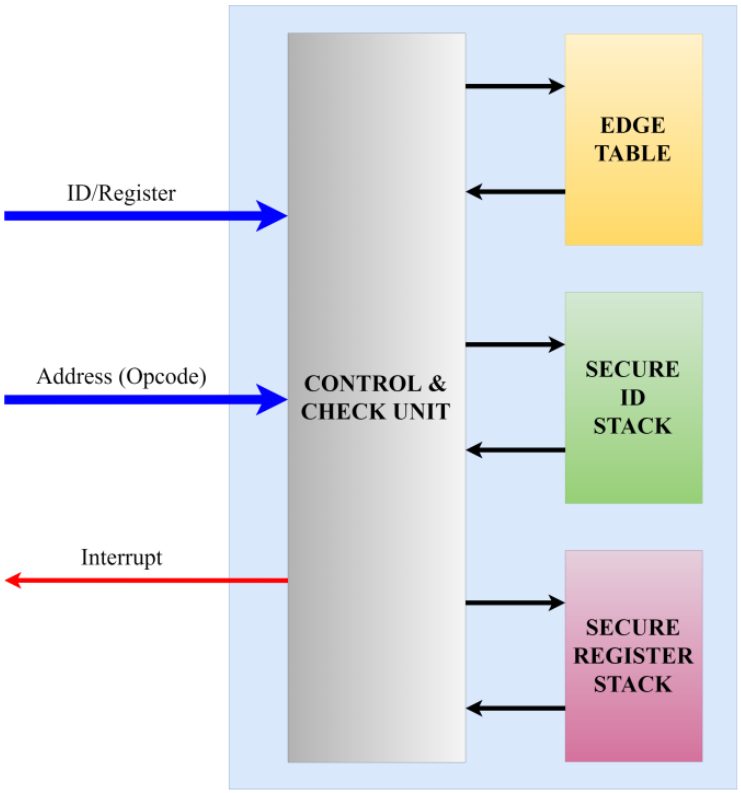


Figure 4.1: CFI monitor block diagram [61].

The CU receives all the requests from the CPU with a specific `opcode`, and depending on that performs some action interacting with the data structures previously described and using the value passed to ID/Register appropriately. Depending on the outcome of the operation, an interrupt can be sent to the CPU to signal a CFI violation, so that the program can take whatever measures are set in place to handle the situation, that being completely stopping operation, restarting, or whatever else the security policy decides. A visual representation of the FPGA design and its interaction with the system can be seen in Figure 4.1.

4.1.2 Protection Mechanism

When put together, the previously described parts of the solution define the complete workflow and operation for the offline operations (summarised workflow is in Figure 4.2) and the run-time ones (Figure 4.3).

To prepare the system, the binary is first analysed with the static analysis tool; after that, the code is instrumented and the CFG information is used to generate the design for the FPGA. When all of that is done, the system is initialised, by programming both the program memory and the FPGA. At this point, the system is ready to run.

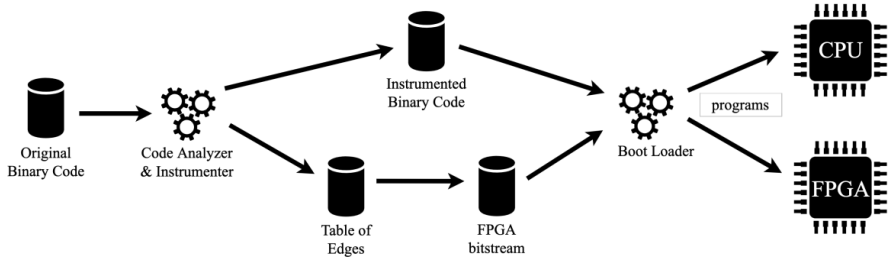


Figure 4.2: Workflow of the analysis, instrumentation and programming process [61].

When the system is running, the instrumented code will send requests to the FPGA. This is done by doing `STORE` operations on certain addresses, which serve as the `opcode`, in which the value that is to be checked will be passed as the `STORE` argument. This is the ID/Register argument for the FPGA. The FPGA can write on an interrupt line to the CPU, which it will raise if a violation is detected.

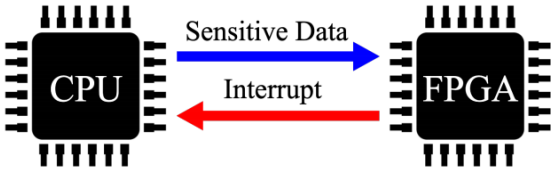


Figure 4.3: CPU-FPGA interaction for monitoring [61].

4.2 Design Goals

The aim of this thesis is to create a solution that uses some of the principles from the previously presented solutions to make an all-encompassing solution tailored for the needs of mixed-criticality systems.

For our proposal, a system that uses a hypervisor to assign different cores of a multicore platform to the high and low criticality parts of our system is used. Our priority is securing

the high-criticality portion of it, which will run a hard real-time RTOS, while keeping performance overhead to a minimum. The FPGA-based solution demonstrates how delegating the bulk of CFI monitoring to an external monitor is an effective strategy, but this means having the extra hardware needed for that. At the time of writing, a lot of commercially available solutions do not have a FPGA on board, but multicore embedded SoCs have become increasingly cheap and common. From this, the idea of using the resources of the lower criticality portion of our system to protect the high-criticality RTOS came to be.

While the high-criticality system is a hard real-time system, the low-criticality one could be a soft or firm real-time system, or just a general-purpose one. This means that the added overhead from the CFI monitoring is tolerable, since it does not have to strictly adhere to set deadlines.

4.3 Threat Model

It is assumed the presence of a powerful attacker that can exploit several vulnerabilities in the code to read and write on the memory. Separate considerations have to be made for the high and low criticality subsystems, assuming the attacker can exploit vulnerabilities in both of them.

For the high-criticality core, the objective is to secure the insecure edges. This means creating the CFG of the tasks and then instrumenting the code. Our system takes interrupts and context switches into consideration as well, so appropriate instrumentation for them is also necessary. All the actual CFI verification will happen in the low-priority core, which holds the CFI monitor and all the data necessary for its operation.

For the system containing the monitor, only protecting the memory by limiting its access to privileged code is not enough. For this purpose, the monitor will make use of the MPU on the platform as well, to lock the memory and unlock it only when necessary. The sensitive data of the monitor will always be inaccessible from userspace, and outside of a set of secure routines it will be read-only for privileged code. The attacker can still potentially read the sensitive data by exploiting for example vulnerable ISRs, but, especially in embedded systems, most exploitable vulnerabilities lie in the userspace code, with interrupt handlers being pretty short and secure. But, even if that were not the case, our solution still makes it so that the attacker has no way to influence the protected regions anyway, as just explained.

4.4 Approach

For our solution, it is needed to use an embedded hypervisor which is tailored for real-time support. It was decided to base our solution on Bao (previously described in 2.4.2), because it is an hypervisor created for application in mixed-criticality environments and it is available as an open-source project.

On its official project page [66], there is a repository containing a few demos, including a demo that shows the operation of FreeRTOS together with Linux on the same quad-core platform, by assigning one core to FreeRTOS and 3 to the Linux system. In this demo, the two VMs communicate through the interrupt system guaranteed by the hypvervisor and send data to one another by writing on a shared memory section. Bao uses a configuration

file to assign the physical resources to each image, defining the optional shared memory section and the interrupt(s) to create for inter-VM communication inside of it. This basic structure is the basis of our solution (two VMs interacting through interrupts and a shared memory region). A model can be seen in Figure 4.4, using a dual-core architecture for the sake of simplicity. This hypothetical dual-core system will be the one considered for all the model definitions from here on.

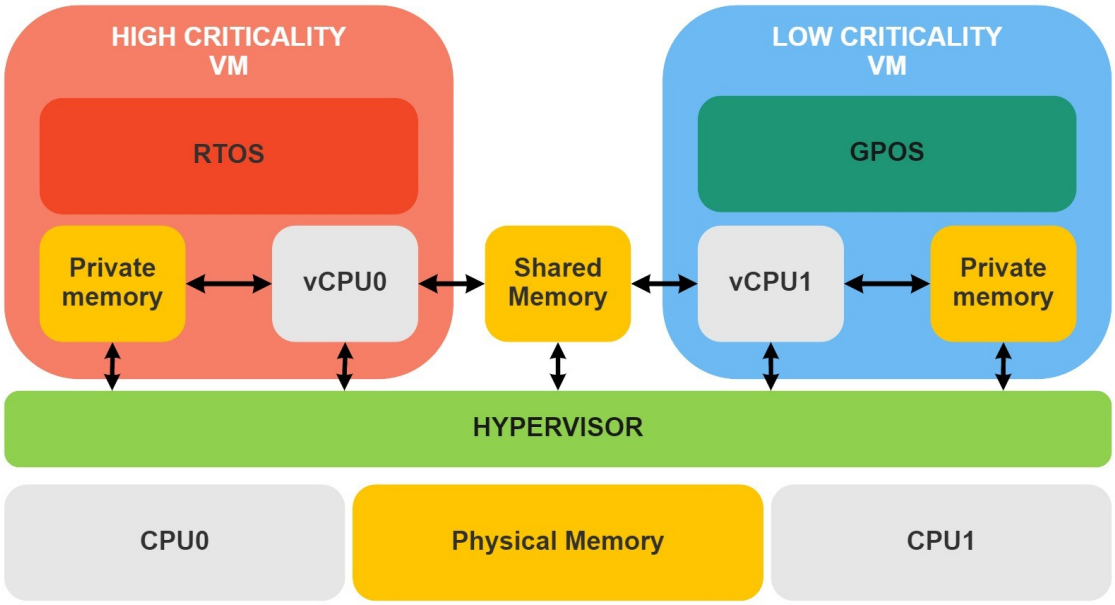


Figure 4.4: Model of the general system definition.

What needs to be defined now is exactly how this structure will be used to implement our design.

4.4.1 Branch Protection

Branch protection has to be applied only to the *insecure* branches, as per definitions in 4.1.1. This means that all the insecure edges can be classified in different groups, each one needing their own dedicated handling:

- Type 1:
  - Insecure forward branches (with a single or multiple targets) to a routine ending with a secure backward branch;
  - Insecure forward branches (with a single or multiple targets) to a routine ending with an insecure backward branch with a single target;
  - Insecure backward branches with a single target;
- Type 2:

  - Insecure forward branches (with a single or multiple targets) to a routine ending with an insecure backward branch with multiple targets;

- Type 3:
  - Secure forward branches to a routine ending with an insecure backward branch having multiple targets;

- Type 4:
  - Insecure backward branches with multiple targets;

Each group will be handled in a different manner, both regarding their instrumentation and the function performed by the CFI monitor. A different interrupt is defined for each one of them to request the right behaviour to the monitoring core.

- **Type 1**: in this case, both the source and the destination(s) will be marked with a unique label identifying them, and instrumentation is inserted at both ends. At the source instructions are inserted right before the branch instruction to write the source label on the shared memory before raising the right interrupt and branching. At the destination (or destinations in the case of multiple targets) instructions are added at the very start to write the destination label on the shared memory and then raise an interrupt to the monitor. When the monitor has received both values, it will use them to check if this is a valid edge of the CFG. If it is not, it will raise an interrupt to signal a violation.

  After the first interrupt from the source instrumentation is received by the monitor, the monitor starts a timer, if it does not receive the second interrupt in due time (which will be the expected time for the branch and instrumented instructions to be executed) it will send an interrupt to signal a violation. This is because in this case the program has jumped to a destination which is not instrumented and thus not a valid target.

- **Type 2**: in this case, both the source and the destination will be marked with a unique label identifying them, and instrumentation is inserted at both ends. The instrumentation is the same as Type 1, with a different interrupt line for the first interrupt.

  The important difference with the previous group is that in this case the monitor will also save the source label for later use when returning from the destination routine. Since this routine can have multiple targets for its backward branch, it is necessary to make sure that the routine returns to its caller and not to another valid target. This is made possible by saving the caller's label and comparing it with the destination of the backward branch when necessary. The same timer functionality as Type 1 is present here as well.

- **Type 3**: in this case there is not any instrumentation at the destination, since there is no need to actually check the branch itself. The only instrumentation is at the source, where the source label is written to the shared memory and a specific interrupt is raised. The monitor will just save this label for later use to make sure that the

backward branch inside the routine will return to the right caller, like in the case of Type 2 instructions.

- **Type 4**: in this case the instrumentation is the same as Type 1, with the monitor receiving both the source and destination labels and activating a timer inbetween them to make sure a valid branch is taken. In addition to that, the monitor also checks the received destination label against the previously saved caller label (the label of the routine that called the one the program is returning from) to make sure that not only a valid branch was taken, but the single right one.

To summarise the workflow needed for each type, some simplified diagrams can be seen in Figure 4.5. It is important to notice that, to keep the models simpler and more immediate, the timer feature is omitted in that representation, but it is of course still an integral part of our real design.

4.4.2 Protection of the interrupt context

Our proposal is modelled for ARM platforms, which means that the way exceptions are handled automatically for ARM was taken into consideration. To ensure maximum security in the event of an interrupt, the same idea presented as presented in 4.1.1 has been adopted, which means saving all the 8 registers automatically saved by ARM in the stack into the monitor as well, and then retrieving them before the ISR returns control to the program. In our model, this is accomplished by 8 `STORE` operations to the shared memory followed by an interrupt to signal the transfer to the monitor. To retrieve them an interrupt is sent to request them to the monitor, which will write them on the shared memory to allow the high-criticality core to receive them.

The problem with this is the possibly significant overhead introduced by this. For applications that have frequent interrupts with very short ISRs, the introduced latency could be a problem and severely impact real schedulability of a workload. A possibility to mitigate this could be to only secure a few select registers. Right now, the registers that are saved are, in order:

- `PC` (Program counter)
- `xPSR` (Processor Status Register)
- `R0` to `R3`
- `R12`
- `LR` (Link Register)

To potentially reduce the overhead, a choice could be made to only secure two registers, `LR` and `PC`, because these are the ones that, if modified, could be used to perform control-flow hijacking attacks, while the other registers are significantly less dangerous in that sense. Depending on the situation, reducing the saved registers to just these 2 could be an acceptable compromise.

4.4.3 Scheduler Awareness

As explained previously in 2.4.1, when working on a CFI implementation for a RTOS, compared to a bare-metal application there is the added issue of context-switching and how to protect the task context when other tasks are running. In FreeRTOS the scheduler saves a total of 34 registers, namely:

- Register `R0` to `R31`
- `SREG` (Status Register, `xPSR` in ARM platforms)
- `PC`

They are used to save the task context and are pushed onto the task's stack. The scheduler only saves the final stack pointer for every task, to be able to restore context later. The problem with this is that vulnerable privileged code (e.g., ISRs) could potentially now access the stack memory of a suspended task and modify it, making control-flow hijacking possible by changing the PC value inside the stack.

To combat this, the strategy from 3.2.1 is adopted. All the context register are sent to the low-criticality core, which saves them inside of an appropriate data structure, making this information not reachable by the RTOS. To do this, some modifications have to be made to FreeRTOS, which are very similar to the ones employed in RECFISH.

- **Task Control Block**: the TCB is extended by adding two fields, one to save the position on the low-priority core memory used to save the context between context switches, and the other one to keep their shadow call stack pointer (this data structure is introduced in 4.4.4). For this to be effective, the functions that handle the initialization of a task and its stack need to be modified as well.
- **Scheduler**: the scheduler runs its own routines to save and restore context. These routines are modified to make the scheduler send and receive data to the monitor instead of using the task's stack. The task's stack pointer, which is used to restore context and is kept by the scheduler for each suspended task, will also be saved in the monitoring core, and a verification will happen when restoring context. If the saved stack pointer and the one retrieved from the monitor differ in any way, a violation will be detected.

4.4.4 Monitor Data Structures

To implement all the functionalities presented up to here, the monitor needs some specific data structures, which will be hosted on the private memory of the low-criticality core.

- **Edge Table**: this is a table that contains all the edges part of the computed CFG. To access the table, the monitor makes a XOR between the source and destination labels, the result of which is used as the index to access the table. Each label is 13 bits long, for a total number of 8192 possible edges. Every entry on the table is a half-word (16 bits), the first 3 bits (which are equal to 0 in the half-words used for the labels) are used to signal if the edge exists in the CFG or not, while the last 13 are the expected source label. This is so that the monitor can unequivocally verify the edge. The first 3 bits can assume the values:

  - **000:** No Edge
  - **100:** Existing Edge

  For the size of the labels 13 bits is just a proposed value to have a reasonable amount of available labels and an Edge Table that is moderate in size, but it could go up to 15 bits to still keep treating them as half-words while maintaining the edge validity check, which needs only one dedicated bit.

  The edge table will be set as a Read-Only memory accessible only to privileged code (which includes interrupts) before starting the system. At boot time, the bootloader will use the MPU functionalities of the ARM platform to do that after the table is populated.

- **Shadow Call Stack**: this is the memory used to store the caller label during the ISR for type 2 and type 3 routines, that will be used to do the caller verification for type 4 edges. The size can be decided depending on the expected maximum number of stacked calls. Each task will have its own shadow stack sector, to avoid possible collisions caused by context switching. To do this, the stack pointer used by the Shadow Call Stack will be updated at every context switch, to operate on the correct subregion. The relative field on the TCB of the task that is being switched out will be updated by writing its current value in the monitor to it. After that, the monitor will take the value from the task's TCB that is being switched in to update its own. All of this is managed by the scheduler.

- **ISR Registers Stack**: in this memory sector, the registers saved for protecting the interrupt context are pushed upon entering the ISR and popped before exiting it. Since ISRs can be nested, the size needs to be able to handle multiple sets of registers. The expected number of nested interrupts can vary depending on the application, but if a maximum value of 5 is supposed, this would mean needing the space to save a total of 40 registers, meaning just 160bytes of memory for the whole stack.

- **Task Context Memory**: in this memory sector, the information needed to restore a task's context is kept. Each task gets its own subsector, dimensioned to hold exactly the needed registers. When a task is running, all the other suspended tasks will have their context stored inside this data structure, meaning that, supposing all tasks have been started already, at every given moment only a subsector will be unused, while the other ones will hold the data necessary to restore their context whenever the scheduler will set them as running.

Data Protection

While the hypervisor ensures that the RTOS has no way to access the memory of the monitor, this is not true for the low-criticality OS. To be able to operate the monitor, the system needs to be able to write and read from the monitor data structures (except for the Edge Table, which is always read-only). Even if they were set as accessible only to privileged code, this would still make possible attacks that use ISRs vulnerabilities to write data on that address range.

While the ideal solution would be to just have minimal ISRs with no external interaction at all, basically eliminating any vulnerability, this is unfortunately not something that can be guaranteed. Another solution would be to just have the low-criticality system run with interrupts disabled, but this is clearly unacceptable for the vast majority of usecases.

Instead, it was decided to make use of the MPU on the ARM platform. The MPU allows the developer to set a number of memory regions to be protected by regulating their accessibility. They can be made so that only privileged code can access the protected memory, with different attributes defining the read-write permissions. In our model three MPU regions are defined, one for each of the data structures that the monitor needs write access to. Whenever the monitor is not active, they are all set to Read-Only (RO) for privileged code exclusively. When a request arrives, the monitor sets the needed memory region(s) as writable for privileged code, writes to it and then sets it back to RO. During all of this interrupts are disabled, to avoid attacks that could exploit a well timed interrupt to access the now unprotected memory regions.

To use the MPU regions, one needs to set the correct values inside the dedicated registers. A region can have its properties updated only when it is disabled, so since they are all enabled the first time during boot, this means that each subsequent update needs to first disable the region, then modify the permissions and finally enable it again. To do this, one needs to modify the values into the `MPU_RNR` (Region Number Register), `MPU_RBAR` (Region Base Address Register) and `MPU_RASR` (Region Attribute and Size Register) registers. An example taken from the ARM documentation [57] is present in Listing 4.1.

```
; R1 = region number
; R2 = size/enable
; R3 = attributes
; R4 = address
LDR R0,=MPU_RNR           ; 0xE000ED98, MPU region number
   register
STR R1, [R0, #0x0]        ; Region Number
BIC R2, R2, #1            ; Disable
STRH R2, [R0, #0x8]       ; Region Size and Enable
STR R4, [R0, #0x4]        ; Region Base Address
STRH R3, [R0, #0xA]       ; Region Attribute
ORR R2, #1                ; Enable
STRH R2, [R0, #0x8]       ; Region Size and Enable
```

Listing 4.1: Basic example of MPU region update. Note that this is the procedure for a region that was previously enabled hence the disable at the start.

For all uses of the MPU functionalities after the initial setup it is only needed to modify the `MPU_RASR` and `MPU_RNR` registers. The first one is the one that enables/disables the region, as well as controlling the permissions, while the latter is used to select to which region the `MPU_RASR` and `MPU_RBAR` registers are referring to. This one is important since three different regions were defined, and the monitor has to write on the right one depending on the operation.

Special routines are made to update each MPU region defined. There are routines for pushing and popping from the shadow call stack(s), the ISR register stack and the task context memory. The routines handling the shadow call stack(s) do not add any extra

latency to the violation detection for type 2, type 3 and type 4 edges, since they are always done at the end of the ISR that handles them. The data on them can still be read to do the CFI evaluation, and only after that will it be modified. Examples of possible code to handle these push and pop from these stacks are presented in 4.2 and 4.3 respectively.

```
push_shadow_stack:
    PUSH {r0, r1, r2, r3}
    LDR r1, SS_POINTER  ; load address of the shadow stack
       pointer
    SUB r1, r1, #0x2    ; decrement stack pointer value

    LDR R0, =MPU_RNR  ; load MPU_RNR address
    MOV R3, #0x0  ; prepare the region number to access
    STRH R3, [R0, #0x0] ; set the region number

    LDR R0, =MPU_RASR ; load MPU_RASR address

    LDRH R3, [R0, #0x0] ; load Region Size and Enable
    BIC R3, R3, #1      ; unset enable bit
    STRH R3, [R0, #0x0] ; disable region

    LDRH R2, [R0, #0x2] ; load attributes
    BIC R2, #0x200      ; change permissions value in R2
    STRH R2, [R0, #0x2] ; set permissions as RW for privileged
        mode

    STRH r0, [r1]       ; store new value on top of stack
    STR r1, SS_POINTER  ; push a value on top of the shadow
       stack by
                ; changing the stack pointer

    ORR R2, #0x200  ; change permissions value in R2
    STRH R2, [R0, #0x2] ; set permissions as R0 for privileged
        mode

    ORR R3, R3, #1      ; set enable bit
    STRH R3, [R0, #0x0] ; enable region
    POP {r0, r1, r2,r3}
    RET
```

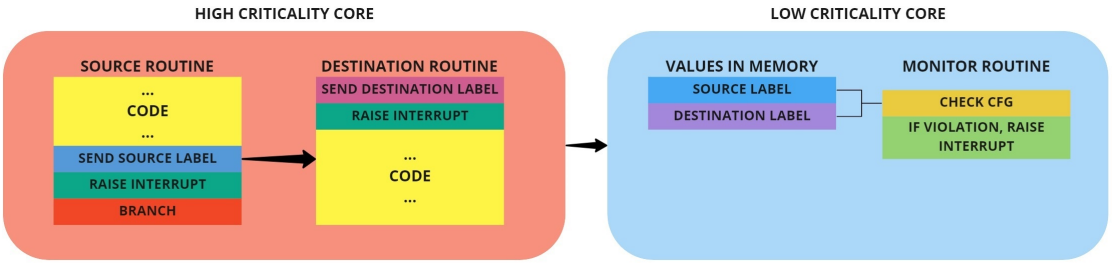
Listing 4.2: Code example of routine for pushing new data on a task's shadow stack.

```
pop_shadow_stack:
    PUSH {r0, r1, r2, r3}
    LDR r1, SS_POINTER
    ADD r1, r1, #0x2
```

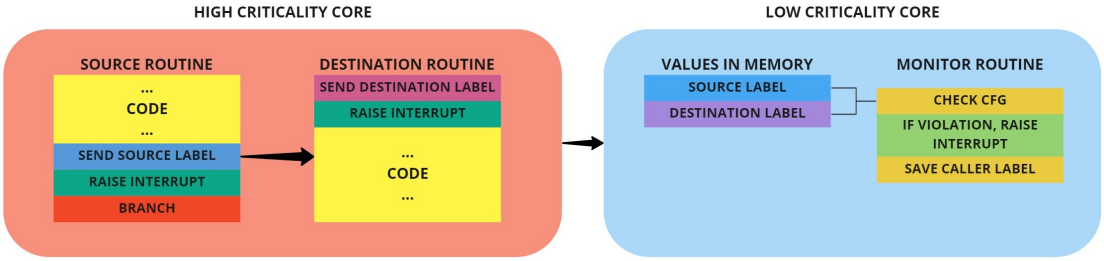
```
    LDR R0, =MPU_RNR  ; load MPU_RNR address
    MOV R3, #0x0  ; prepare the region number to access
    STRH R3, [R0, #0x0] ; set the region number

    LDR R0, =MPU_RASR ; load MPU_RASR address

    LDRH R3, [R0, #0x0] ; load Region Size and Enable
    BIC R3, R3, #1      ; unset enable bit
    STRH R3, [R0, #0x0] ; disable region

    LDRH R2, [R0, #0x2] ; load attributes
    BIC R2, #0x200      ; change permissions value in R2
    STRH R2, [R0, #0x2] ; set permissions as RW for privileged
        mode

    STR r1, SS_POINTER  ; remove a value from top of shadow
       stack by
                ; changing the stack pointer

    ORR R2, #0x200  ; change permissions value in R2
    STRH R2, [R0, #0x2] ; set permissions as R0 for privileged
        mode

    ORR R3, R3, #1      ; set enable bit
    STRH R3, [R0, #0x0] ; enable region
    POP {r0, r1, r2, r3}
    RET
```

Listing 4.3: Code example of routine to remove the topmost element of a task's shadow stack. This is not an actual pop operation since it only changes the stack pointer value the value is read directly from the ISR that later calls this function.

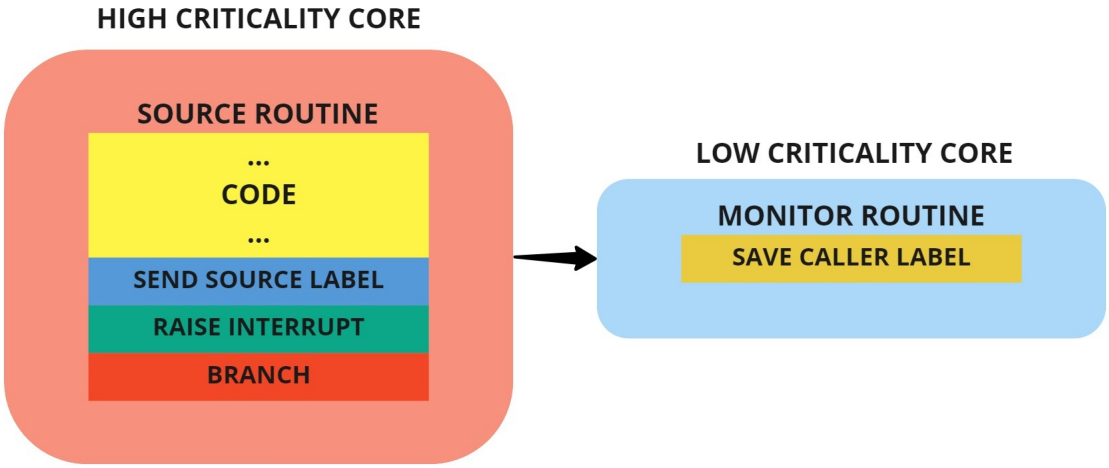
Similar routines are present for the handling the contents of the ISR Registers Stack and the Task Context Memory. The only real differences will be the amount of data saved and which MPU region is accessed.



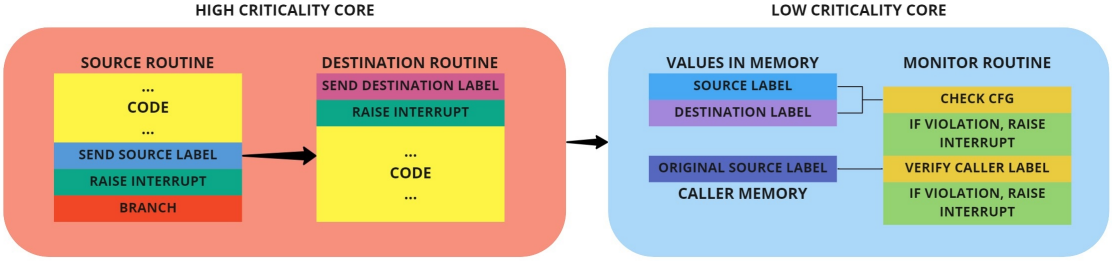
(a) Type 1.



(b) Type 2.



(c) Type 3.



(d) Type 4.

Figure 4.5: Simplified model of the workflow for each edge type. The timer operation has been omitted for brevity.

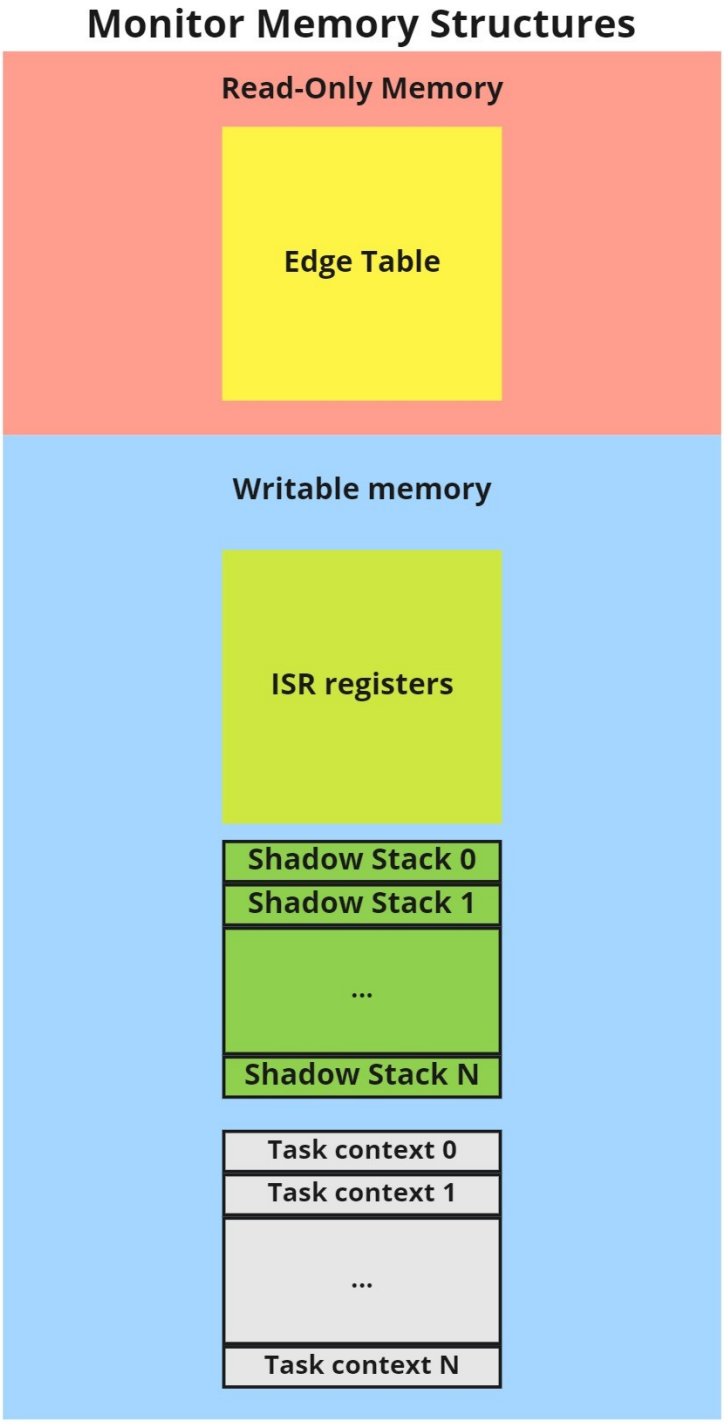


Figure 4.6: Internal memory subdivision of the CFI monitor.

Chapter 5

Theoretical Evaluation

The proposed solution still has no real implementation, so there are not actual benchmarks to analyse. Instead, a theoretical evaluation of the monitor timings is provided, based on the available information (e.g., the clock cycle cost of the ARM instructions). There are two categories that have to be considered:

1. The overhead introduced by the instrumentation on the RTOS
2. The timings needed for the monitor operations, which correspond to the timing in-between a possible a violation happening and its detection

Every type of edge is instrumented following the same two patterns for the source and destination routines, which are presented in Listings 5.1 and 5.2. Type 3 edges differ from the other ones in the sense that instrumentation happens only at the source, since it is a secure edge (at the destination only interrupts are re-enabled). The basic instrumentation stays the same, though, since the RTOS is still sending the label to the monitor to save it in the shadow call stack. Each type will of course raise a different interrupt, to let the monitor know which actions to take to handle the incoming data.

```
source_routine:
    ...
    [CODE]
    ...
    CPSID if        ;disable interrupts
    PUSH {R0,R1}
    MOV R0, SH_MEM_SOURCE_ADDRESS
    MOV R1, SOURCE_LABEL
    STRH R1, [R0]
    POP {R0,R1}
    [FIRST INTERRUPT]
    [BRANCH INSTRUCTION]
```

Listing 5.1: Pseudoinstructions for the source instrumentation.

```
destination_routine:
    PUSH {R0,R1}
    MOV R0, SH_MEM_DESTINATION_ADDRESS
    MOV R1, DESTINATION_LABEL
    STRH R1, [R0]
    POP {R0,R1}
    [SECOND INTERRUPT]
    CPSIE if          ;enable interrupts
    ...
    [CODE]
    ...
```

Listing 5.2: Pseudoinstructions for the destination instrumentation.

The cost of the various instructions (in CPU clock cycles) can be taken from the ARM instruction set documentation [55]. The instructions that interest us and their cost is reported below.

- `CPSID`, `CPSIE`: 1 cycle
- `STRH`: 2 cycles
- `PUSH`, `POP`: 1+N cycles, with N being the number of registers in the push/pop list
- `MOV`: 1 cycle

From that it can be calculated that:

- For type 1, 2, and 4 edges, that need full instrumentation at both ends, the overhead is:
$$Overhead = (1 + 3 + 1 + 1 + 2 + 3 + INT) * 2 = 22 + 2 * INT \tag{5.1}$$
- For type 3 edges, which have full instrumentation only at the source and just add the re-enabling of interrupts at the destination, the overhead is:
$$Overhead = 1 + 3 + 1 + 1 + 2 + 3 + 1 + INT = 12 + INT \tag{5.2}$$

The value of INT is the time it takes for the hypervisor to generate and transfer the virtual interrupt used to make the high-criticality and low-criticality cores notify eachother. In Bao, this is made through a *hypercall*. The specific operation is defined as a *Virtual Inter Processor Interrupt* (vIPI). Unfortunately, the overhead introduced by the hypercall varies greatly depending on the platform and the hypervisor, with some solutions differing a few orders of magnitudes from one another. More recent architectures have hardware support for virtualisation which can help significantly decrease the overhead of these operation, but at the moment there is no information available for Bao specifically, so benchmarks on a real implementation are needed to accurately evaluate the impact of the hypervisor on this solution. Depending on the results, some changes may be advisable. For example, instead of using a second interrupt at the destination, it could potentially be less costly to zero out the shared memory after each operation and use a polling mechanism instead of

the interrupt to wait for the second label. Future work and tests need to be done in this regard.

Another area of the RTOS that is impacted by our changes is the scheduler. The overhead introduced by the instrumentation itself is minimal, since the operations to save the context are basically the same, with the difference that the scheduler is storing and loading the values to/from the shared memory, to interact with the context memory in the monitor. Only two extra `STORE` instructions are needed, one to save the exiting task's stack pointer together with the context and one to verify the entering task's one against the saved one, for verification. This introduces an overall very limited overhead of just 4 cycles. The other source of possible overhead are the modifications done to the code handling the tasks, because of the modified TCB. This, like the hypercall overhead, cannot be quantified without running benchmarks on an implemented version of our solution, which once again reiterates the importance of future work and testing.

As for the CFI monitor, the overhead itself does not concern us as much. What is more important is understanding how many clock cycles have elapsed since the arrival of the second label when a possible violation is noticed and notified. Since Type 3 edges cannot trigger any violation (their routine just pushes the source label on the shadow stack) they are not subject to this analysis.

The instructions used by the monitor and their cost are:

- `CPSID`, `CPSIE`: 1 cycle
- `LDR`, `LDRH`: 2 cycles
- `EOR`, `AND`, `TST`: 1 cycle
- `CMP`, 1 cycle
- `BEQ`, `BNE`: 1 cycle if the branch is not taken, 3 if it is taken. For our evaluation, the cost when the branch is not taken is what is important, so it will be considered to be equal to 1

Here is the pseudocode for the interested edges:

```
type1_handler:
    CPSID if    ;disable interrupts
    LDRH r0 source  ;load source label from shared memory
    LDRH r1 dest    ;load destination label
    EOR r1 r0 r1  ;xor labels to get edge table index
    LDRH r1 table[r1] ;access edge table
    TST r1, #0x8000     ;test the validity bit
    BEQ error           ;if branch not valid
    BIC r1, #0x8000 ;delete validity bit
    CMP r0, r1    ;check if source label same as value in edge
        table
    BEQ success   ;if equal, jump over interrupt
    error:
    [SEND INTERRUPT]
```

```
    success:
    CPSIE if    ;enable interrupts
```

Listing 5.3: Pseudoinstructions for the Type 1 monitor routine.

```
type2_handler:
    CPSID if    ;disable interrupts
    LDRH r0 source  ;load source label from shared memory
    LDRH r1 dest    ;load destination label
    EOR r1 r0 r1  ;xor labels to get edge table index
    LDRH r1 table[r1] ;access edge table
    TST r1, #0x8000     ;test the validity bit
    BEQ error           ;if branch not valid
    BIC r1, #0x8000 ;delete validity bit
    CMP r0, r1    ;check if source label same as value in edge
        table
    BEQ success   ;if equal, jump over interrupt
    error:
    [SEND INTERRUPT]
    B out   ;do not push to shadow stack if it was a violation
    success:
    BL push_shadow_stack
    out:
    CPSIE if    ;enable interrupts
```

Listing 5.4: Pseudoinstructions for the Type 2 monitor routine.

```
type4_handler:
    CPSID if    ;disable interrupts
    LDRH r0 source  ;load source label from shared memory
    LDRH r1 dest    ;load destination label
    EOR r1 r0 r1  ;xor labels to get edge table index
    LDRH r1 table[r1] ;access edge table
    TST r1, #0x8000     ;test the validity bit
    BEQ error           ;if branch not valid
    BIC r1, #0x8000 ;delete validity bit
    CMP r0, r1    ;check if source label same as value in edge
        table
    BNE error   ;if true, jump over interrupt
    LDR r0, SS_POINTER  ;load shadow stack pointer
    LDRH r0, [r0] ;load label on top of shadow stack
    CMP r0, r1    ;compare with destination
    BEQ success   ;if equal, jump over interrupt
    error:
    [SEND INTERRUPT]
    B out   ;do not push to shadow stack if it was a violation
    success:
    BL pop_shadow_stack
```

```
    out:
    CPSIE if     ;enable interrupts
```

Listing 5.5: Pseudoinstructions for the Type 4 monitor routine.

It can be noticed that the routines for Type 1 and Type 2 edges behave the same way, except for pushing the value of the source onto the shadow stack. This means that, for the purpose of this evaluation, they incur in the same overhead. From that it can be calculated that:

- For type 1 and 2 edges, the time needed to detect a violation and notify it is:

$$Overhead = 1 + 2 + 2 + 1 + 2 + 1 + 1 + 1 + 1 + 1 = 13cc \tag{5.3}$$

- For type 4 edges, which have to not only check the Edge Table, but also check the source label against the value on top of the shadow stack, the cost is:

$$Overhead = 1 + 2 + 2 + 1 + 2 + 1 + 1 + 1 + 1 + 1 + 2 + 2 + 1 + 1 = 19cc \tag{5.4}$$

These numbers can be considered to be a positive result, since it is extremely unlikely to be able to pull off any kind of meaningful attack in this time frame. The longer time needed to evaluate Type 4 edges originates from the fact that it is going through a double verification, using both the Edge Table and the Shadow Call Stack. In order to reduce the delay, it could potentially be decided to just check the destination label against the value on top of the call stack, without going through the Edge Table. This should hold the same security guarantees, since there is trust in the memory holding the Shadow Call Stack (which is protected thanks to the MPU) and it can be supposed that the control-flow until this point has been kept safe, meaning that the backward edge is starting from the right place. To implement these changes, the instrumentation would be slightly different and this could end up saving up to 9 clock cycles. The different instrumentation could also reduce the overhead imposed on the RTOS: if the system can be trusted to have been kept safe up to this point, this means that the backward branch can start from only one point, making sending the source label unnecessary. Right now the decision made is to keep both verifications to have a double layer of security as a fail-safe in case of unknown vulnerabilities or new attacks. If this were to be proved as a problem during real applications and benchmarking, the described changes could be implemented.

Chapter 6

Conclusions and Future Work

This thesis aims to provide an analysis of the current state of Control-flow Integrity solutions for embedded Real-Time Operating Systems, defining their strength and weaknesses and laying down the basis for an “ideal" solution. It then proposes a solution for a subset of embedded RTOSes, in particular mixed-criticality multicore systems.

The analysis of the current state of the art has looked at software-based solutions in the available, most recent research. Since specific research in this field is still scarce, the search has extended to include solutions that, while not strictly implemented for RTOSes, could be a starting point for a specific solution. From this research a paper titled *Control-flow integrity for real-time operating systems: open issues and challenges* has been presented in the IEEE East-West Design and Test Symposium 2021 [72] conference in September of 2021, and will be published and become publicly available in November of 2021.

The proposed solution takes inspiration from both the analysed solutions and the solution presented in the work *A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems* [61] by Prof. Paolo Prinetto and his research team here at Politecnico di Torino. It applies virtualisation as a way to isolate the sensitive memory areas needed to implement CFI and decrease the performance impact on the RTOS, adapting the hardware-assisted idea of the FPGA-based solution to easily available multicore platforms.

To create this isolation, the low-criticality part and the high-criticality one are split onto different cores through the use of an hypervisor. For this work the hypervisor of choice was Bao, an open-source embedded hypervisor tailored for mixed-criticality systems with real-time needs. The CFI monitoring is done by a custom monitor built into the OS running on the low-criticality core, which will bear most of the overhead from this implementation. The high-criticality core will just be impacted by the instructions needed to send the data to the monitor. The way the two cores are able to communicate is through a shared memory and a virtual interrupt system, all guaranteed by the hypervisor. After sending the data, the RTOS keeps operating, while the monitor verifies the validity of the control flow transfers. If a violation is detected, the monitor will quickly signal the violation to the high-criticality core through an interrupt, to possibly stop the system or handle the violation otherwise. This operation has to be as quick as possible to reduce the feasibility of attacks during the verification phase.

The solution aims to provide complete CFI coverage with minimal overhead to the

high-criticality subsystem. At the moment only a theoretical evaluation of the overhead and monitor timings is provided. Future work should concentrate on implementing this solution and running a series of benchmarks to extract real performance metrics on actual hardware. This is necessary especially because part of the involved overhead is dependant on both the hardware and its interaction with the hypervisor, a factor which could not be accurately be accounted for in the evaluation in this work. Some of the changes introduced to FreeRTOS also need real benchmarks to be adequately estimated. Depending on the results, possibly further optimizations could be needed, for example by reducing the number of virtual interrupts needed (if they were to prove too performance-heavy) or researching of ways to optimize the FreeRTOS modifications. Future expansion of this solution could also work on adding CFI protection to the low-criticality core as well, by for example making use of *supervisor calls* instead of *hypercalls* to interact with the CFI monitor, similarly to 3.2.1.

Bibliography

[1] M. Abadi, M. Budiu, Ú. 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] Robert P Abbott, Janet S Chin, James E Donnelley, William L Konigsford, S Tokubo, and Douglas A Webb. *Security analysis and enhancements of computer operating systems*. Tech. rep. NATIONAL BUREAU OF STANDARDS WASHINGTONDC INST FOR COMPUTER SCIENCES AND ..., 1976.

[4] Luca Abeni and Dario Faggioli. «Using Xen and KVM as real-time hypervisors». In: *Journal of Systems Architecture* 106 (2020), p. 101709.

[5] J. Afek and A. Sharabani. *Dangling pointer: Smashing the pointer for fun and profit*. 2007.

[6] Roberto Avanzi. «The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes». In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 4–44.

[7] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. «RIOT: An open source operating system for low-end embedded devices in the IoT». In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4428–4440.

[8] Fabrice Bellard. «QEMU, a fast and portable dynamic translator.» In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[9] S. Bhatkar, D. DuVarney C, and R. Sekar. «Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.» In: *USENIX Security Symposium*. Vol. 12. 2. 2003, pp. 291–301.

[10] 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.

[11] 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.

[12] ARM Connected Community Blog. *ARMv8-A architecture – 2016 additions*. https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions. 2016.

[13] 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.

[14] 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.

[15] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. «When good instructions go bad: Generalizing return-oriented programming to RISC». In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM. 2008, pp. 27–38.

[16] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. «Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers». In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE. 2006, pp. 111–126.

[17] N. Carlini and D. Wagner. «ROP is Still Dangerous: Breaking Modern Defenses». In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 385–399.

[18] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. «Control-flow bending: On the effectiveness of control-flow integrity». In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 161–176.

[19] Defense Use Case. «Analysis of the cyber attack on the Ukrainian power grid». In: *Electricity Information Sharing and Analysis Center (E-ISAC)* 388 (2016).

[20] *CCWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*. https://cwe.mitre.org/data/definitions/120.html. [Online; accessed 21-July-2021]. 2021.

[21] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. «Return-oriented programming without returns». In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 559–572.

[22] S. Checkoway, A. J. Feldman, B. Kantor, J.A. Halderman, E. W. Felten, and H. Shacham. «Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage.» In: *EVT/WOTE* 2009 (2009).

[23] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. «Automatic construction of jump-oriented programming shellcode (on the x86)». In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, pp. 20–29.

[24] Thomas M Chen and Saeed Abu-Nimeh. «Lessons from stuxnet». In: *Computer* 44.4 (2011), pp. 91–93.

[25] 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.

[26] P Clarke. «ARM RISC leads the 32-bit embedded market.» In: *Electronic Engineering Times* 1100 (2000), pp. 49–49.

[27] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. «Losing control: On the effectiveness of control-flow integrity under stack attacks». In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 952–963.

[28] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. «StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks». In: 98 (Jan. 1998), pp. 5–5.

[29] John Criswell, Nathan Dautenhahn, and Vikram Adve. «KCoFI: Complete control-flow integrity for commodity operating system kernels». In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 292–307.

[30] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. «Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection». In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 401–416.

[31] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. «Camouflage: Hardware-assisted CFI for the ARM Linux kernel». In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[32] Solar Designer. «Getting around non-executable stack (and fix)». In: *http://ouah. bsdjeunz. org/solarretlibc. html* (1997).

[33] Common Weakness Enumeration. *CWE entry for stack-based buffer overflow*. https://cwe.mitre.org/data/definitions/121.html. 2021.

[34] Nicolas Falliere, Liam O Murchu, and Eric Chien. «W32. stuxnet dossier». In: *White paper, Symantec Corp., Security Response* 5.6 (2011), p. 29.

[35] Valentina Forte. *Automatic Binary Analysis and Instrumentation of Embedded Firmware for a Control-Flow Integrity Solution*. https://webthesis.biblio.polito.it/18099/. 2021.

[36] The Linux Foundation. *Official website of the Xen project*. https://xenproject.org/. 2021.

[37] The Linux Foundation. *PREEMPT_RT: The Linux Kernel real-time patch*. https://wiki.linuxfoundation.org/realtime/start. 2021.

[38] The Linux Foundation. *Xen: Embedded and Automotive*. https://xenproject.org/developers/teams/embedded-and-automotive/. 2021.

[39] A. Francillon and C. Castelluccia. «Code injection attacks on harvard-architecture devices». In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM. 2008, pp. 15–26.

[40] Aurélien Francillon, Daniele Perito, and Claude 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.

[41] FreeRTOS. *FreeRTOS - Market Leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. https://www.freertos.org/. [Online; accessed 27-July-2021]. 2021.

[42] FreeRTOS. *Real Time Scheduling in FreeRTOS*. https://www.freertos.org/implementation/a00008.html. 2021.

[43] FreeRTOS. *RTOS Context Switch*. https://www.freertos.org/implementation/a00022.html. 2021.

[44] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. «Out of Control: Overcoming Control-Flow Integrity». In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 575–589. DOI: 10.1109/SP.2014.43.

[45] SYSGO GmbH. *PikeOS RTOS & Hypervisor*. https://www.sysgo.com/pikeos. 2021.

[46] Qualcomm Technologies Inc. *Pointer authentication on ARMv8.3*. Tech. rep. 2017.

[47] *Interactive: The Top Programming Languages 2021 - IEEE Spectrum*. https://spectrum.ieee.org/top-programming-languages/. [Online; accessed 07-June-2021]. 2020.

[48] Robert Kaiser and Stephan Wagner. «Evolution of the PikeOS microkernel». In: *First International Workshop on Microkernels for Embedded Systems*. Vol. 50. 2007.

[49] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. «kvm: the Linux virtual machine monitor». In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[50] T. Kornau et al. «Return oriented programming for the ARM architecture». PhD thesis. Master's thesis, Ruhr-Universität Bochum, 2010.

[51] Silicon Laboratories. *Micrium Software and Documentation*. https://www.silabs.com/developers/micrium. 2021.

[52] Chris Lattner and Vikram Adve. «LLVM: A compilation framework for lifelong program analysis & transformation». In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[53] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. «Fine-CFI: fine-grained control-flow integrity for operating system kernels». In: *IEEE Transactions on Information Forensics and Security* 13.6 (2018), pp. 1535–1550.

[54] Jinku Li, Zhi Wang, Tyler Bletsch, Deepa Srinivasan, Michael Grace, and Xuxian Jiang. «Comprehensive and efficient protection of kernel control data». In: *IEEE Transactions on Information Forensics and Security* 6.4 (2011), pp. 1404–1417.

[55] ARM Limited. *Instruction set summary*. https://developer.arm.com/documentation/ddi0432/c/CHDCICDF. 2021.

[56] ARM Limited. *Stacking in ISR*. https://developer.arm.com/documentation/ddi0337/e/Exceptions/Pre-emption/Stacking?lang=en. 2021.

[57] ARM Limited. *Updating an MPU Region*. https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/optional-memory-protection-unit/updating-an-mpu-region. 2021.

[58] Bingbing Luo, Yimin Yang, Changhe Zhang, Yi Wang, and Baoying Zhang. «A survey of code reuse attack and defense». In: *International Conference on Intelligent and Interactive Systems and Applications*. Springer. 2018, pp. 782–788.

[59] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. «Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems». In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

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

[61] 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)*. IEEE. 2020, pp. 1–10.

[62] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. «Opaque Control-Flow Integrity.» In: *NDSS*. Vol. 26. 2015, pp. 27–30.

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

[64] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. «Trackos: A security-aware real-time operating system». In: *International Conference on Runtime Verification*. Springer. 2016, pp. 302–317.

[65] J. Pincus and B. Baker. «Beyond stack smashing: recent advances in exploiting buffer overruns». In: *IEEE Security Privacy* 2.4 (2004), pp. 20–27. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.36.

[66] Bao Project. *Offial Bao Project GitHub*. https://github.com/bao-project. 2021.

[67] Open Web Application Security Project. *OWASP Top 10 IoT vulnerabilites 2018*. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project. 2019.

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

[69] Microsoft Support. *A detailed description of the Data Execution Prevention (DEP)*. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. [Online; accessed 18-June-2019].

[70] L. Szekeres, M. Payer, T. Wei, and D. Song. «SoK: Eternal War in Memory». In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[71] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. «Sok: Eternal war in memory». In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[72] East-West Design & Test. *Official website of the IEEE East-West Design and Test Symposium*. https://conf.ewdtest.com/. 2021.

[73] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. «Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}». In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 941–955.

[74] Tzi-Cker Chiueh and Fu-Hau Hsu. «RAD: a compile-time solution to buffer overflow attacks». In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 409–417. DOI: 10.1109/ICDSC.2001.918971.

[75] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. «An empirical study of c++ vulnerabilities in crowd-sourced code examples». In: *IEEE Transactions on Software Engineering* (2020).

[76] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. «Control-flow integrity for real-time embedded systems». In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[77] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. «Securing untrusted code via compiler-agnostic binary rewriting». In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, pp. 299–308.

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

[79] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. «ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels». In: *arXiv preprint arXiv:1912.10666* (2019).

[80] Mingwei Zhang and R Sekar. «Control Flow Integrity for COTS Binaries». In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 337–352.

Acknowledgements

This thesis is the termination of my university career, which the people close to me know how stressful has been at times. I am thankful for all the wonderful people that stayed by my side during the most difficult periods.

I would like to thank Prof. Paolo Prinetto for the opportunity to work on this thesis and introducing me to the field of cybersecurity, which has proven to be very stimulating for me. Huge thanks go to Gianluca Roascio, who gave me a lot of support during the writing of my thesis, and who has been of great help during the formulation of the solution itself.

Special thanks, then, have to go to all the people that have supported me through the years, from my Bachelor's up to here. I want to thank my family and my parents, for having been extremely patient with me and always being there when I needed them. I want to thank Giulia, for having been an incredible support through the years and helping me be able to get here in one piece. I want to thank all of my friends: the childhood friends that have been there for me since we were little, the ones I've met here in Torino and the ones that live not here nor there, but even when distant have managed to have an important role in my life.

Thank you to all of you.