Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES

Arrêtée ministériel : 25 mai 2016

Présentée par

Cyril BRESCH

Thèse dirigée par **Ioannis Parissis**, Université Grenoble Alpes
et codirigéee par **David Hély**, Maitre de Conférence, Université Grenoble Alpes
et **Stéphanie Chollet**, Maitre de Conférence, Université Grenoble Alpes

préparée au sein du **Laboratoire de Conception et d'Intégration des Systèmes (LCIS)**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

Approches, Stratégies, et Implémentations de Protections Mémoire dans les Systèmes Embarqués Critiques et Contraints.

Approaches, Strategies, and Implementations of Memory Safety Defenses in Critical and Constrained Embedded Systems.

Thèse soutenue publiquement le **16 noctobre 2020,**
devant le jury composé de:

Monsieur AURELIEN FRANCILLON
PROFESSEUR, EURECOM - SOPHIA-ANTIPOLIS, Rapporteur

Monsieur SEBASTIEN PILLEMENT
PROFESSEUR, UNIVERSITE DE NANTES, Rapporteur

Madame MARIE-LAURE POTET
PROFESSEUR, GRENOBLE-INP, Examinateur

Monsieur GIORGIO DI NATALE
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Président

Monsieur ROMAN LYSECKY
PROFESSEUR, UNIVERSITE D'ARIZONA, Invité

"To my family, and all my friends around the world, for all the love and support."

Acknowledgements

The existence of this work would probably require the writing of a second manuscript to thank all the people who supported it. It is thanks to them, over a time span of three amazing years, that I could complete this work.

First of all, I would like to thank my thesis' jury: Mr. Giogio Di Natale, Mrs. Marie-Laure Potet, Mr. Aurélien Francillon, and Mr. Sébastien Pillement for willing to serve on my thesis committee. They carefully attended my defense, listened to my vision regarding my research, evaluated it, recognized it, and finally, awarded me with the degree of doctor. Their insightful comments, suggestions, and challenging questions have helped me make significant improvements to this thesis and beyond. We had memorable discussions and interesting debates during the defense that will contribute to continuing my education as a young researcher.

This thesis would not have been possible without the dedication of my thesis directors, David Hély and Ioannis Parissis, as well as my co-supervisor Stéphanie Chollet. I am delighted to have worked with them, they encouraged me to pursue my interests and provided me the freedom and guidance to explore my topic. Besides their scientific support, they have always been there for support and helpful advice. I particularly appreciated Stéphanie's rigor and her approach regarding my research. She taught me the skills of a professional researcher such as exposing a scientific problem and approaching it from different perspectives. I obviously can't forget David's openness, good mood, and support in all my ideas. He approved my desire to go to the United States and collaborate with other researchers. Besides, I will never forget our sparkling meetings, full of ideas and ambitions during these moments. Working with him has been inspiring.

I would also like to thank Roman Lysecky, professor at the University of Arizona in Tucson. He hosted me in his research team during my international exchange in the United States. He treated me as one of his Ph.D. students, and I am immensely grateful for his guidance and assistance during my stay in the United States, and after, while collaborating on various research papers. We've accomplished a lot and I'm very happy about that.

I also want to take this opportunity to thank the SERENE-IoT project, the Laboratoire de Conception et d'Integration des Systèmes (LCIS) Laboratory, the Grenoble EEATS doctoral school, Grenoble Alpes Cybersecurity Institute, and IDEX for financially supporting this work, the conferences, and all travel.

Last but not least, I would like thanks to all my friends, my colleagues, and my labmates. I would like to give a special thanks to both Baptiste Pestourie and Luc Perard, my two friends who spent a lot of time with me in Valence. Thank you Soraya Zahouily and Louise Constant for supporting me, even in the most difficult moments. Finally, I would not forget my family for their continuous love and support throughout all these years.

Contents

Introduction

With the emerging technologies in several domains such as artificial intelligence, communication, sensors, and processing power, manufacturers are increasingly developing new ubiquitous connected devices identified as the “Internet of Things” –IoT–. According to Forbes [1], the IoT is a fast-growing market that may even double before 2021. As a result, manufacturers see in IoT a genuine business opportunity that encourages them to increasingly release new smart devices over the coming years.

The prime purpose of IoT devices is to operate in an environment by collecting, processing, and sharing data over the network with other computers without any human-to-human or human-to-computer interaction. From the consumers' perspective, smart devices are technological innovations that aim at improving their daily life. One of the most common applications example of the “Internet of Things” is the smart home. In a smart home, a user can monitor some equipment such as the lights, the temperature, and the appliances by only using a smartphone. However, the “Internet of Things” is not limited to smart homes. Several sectors [2] such as agriculture, transport, healthcare, and the military are heavily invested in the development of innovative IoT infrastructures to improve their quality of life and quality of service.

This thesis unfolds within the European project SERENE-(IoT Secured & EneRgy EfficieNt hEalth-care solutions for IoT market). SERENE-IoT project is labeled within the framework of PENTA, the EUREKA Cluster for Application and Technology Research in Europe on NanoElectronics. The project contributes to developing high quality connected care services and diagnostic tools based on advanced smart health-care IoT devices. SERENE-IoT leverages the emergence of the "Internet of Medical Things" -IoMT- to prototype new devices, fully manufactured in Europe, that increases the healthcare quality of service for patients remotely followed by caregivers at a much lower cost than the traditional care.

Unfortunately, connecting medical devices to the network is a concern. It raises relevant questions especially in the areas of privacy, security, and safety [3]. The main purpose of this thesis performed within the CTSYS team of the Laboratoire de Conception et d'Intération des Systèmes (LCIS) in Valence and collaboration with the University of Arizona in Tucson is to focus on the embedded system application security of the next generation of IoMT. However, before digging into the security concerns that impact medical devices, it is important to define what is a medical device. According to the Food and Drug Administration (FDA) [4], a medical device is “an instrument, apparatus, imple-

ment, machine, contrivance, implant, in vitro reagent, or other similar or related article, including a component part or accessory which is: recognized in the official National Formulary, or the United States Pharmacopoeia, or any supplement to them, intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man or other animals, or intended to affect the structure or any function of the body of man or other animals, and which does not achieve its primary intended purposes through chemical action within or on the body of man or other animals and which is not dependent upon being metabolized for the achievement of any of its primary intended purposes". For instance, two medical devices are shown in Figure 1. On the left, a pacemaker used in the treatment of heart diseases. On the right, an insulin pump commonly used in the treatment of diabetes.

Figure 1: Off-the-shelf medical devices

In consistence with healthcare manufacturers, the main purpose of connecting these two medical devices is to improve the quality of service. It may help healthcare professionals in the prevention, diagnosis, and treatment of patients' disease. From the patient perspective, wireless medical devices improve the quality of life. For instance, it is much less restrictive to update a pacemaker using the Bluetooth^{TM} [5] protocol rather than performing surgery. Thus, healthcare manufacturers see IoMT as a way to revolutionize and improve patient care. Thanks to connectivity many medical devices may help healthcare professionals to identify early disease at the lowest cost but also to monitor and adjust the progress of a patients' treatment over time [3].

A representative example of a wireless medical device is a connected insulin pump. Insulin pumps are commonly used in the treatment of diabetes. Today, according to the World Health Organization (WHO) [6] it is more than 8% of the worldwide population that suffers from diabetes. To stay alive, these people must regularly inject themselves manually with insulin to keep their blood sugar at a reasonable threshold. Although this practice is necessary, it remains very binding for patients. Thanks to new IoMT devices such as wireless insulin pump [7,8] the process of measuring and self-injecting the insulin is automated and even remotely controlled. As a result, the quality of life and comfort of patients significantly improved.

Figure 2 displays an overview of a closed-loop wireless insulin pump. Three major

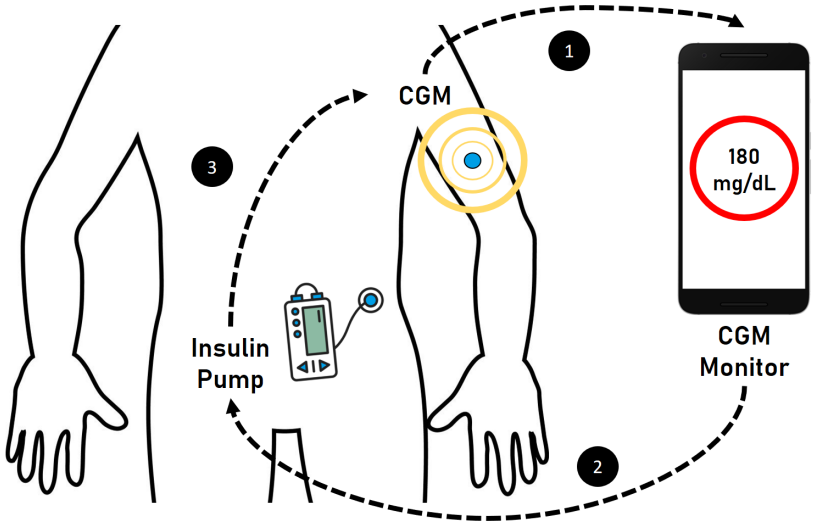

Figure 2: Insulin pump closed-loop

components are identified. The Continuous Glucose Monitor (CGM) is a tiny sensor implanted under the skin that measures glycemia at regular intervals. These measurements are sent to a remote smartphone application that computes and determines whether the blood glucose is too high (1 in Figure 2). If the glycemia is too high, a bolus injection command is sent from the smartphone to the insulin pump (2 on Figure 2). Usually, these commands are sent using a low-energy radio protocol such as Bluetooth^{TM} [5]. Once received, the command is processed and injects an insulin bolus to the patient. Finally, the injected insulin maintains the glycemia of the patient that is again measured by the CGM forming the closed-loop system (3 on Figure 2).

The extension of connected devices such as insulin pumps in the healthcare industry is a promising innovation and will undoubtedly bring many benefits to both caregivers and patients. However, as previously stated, connecting medical devices to the network is a concern. Indeed, a large majority of medical devices such as pacemakers, insulin pumps, defibrillators are considered life-critical systems. In other words, these devices are in direct interaction with the human body and may harm in the event of a hazard or even worse, a security breach. Of course, healthcare manufacturers are well trained about safety issues and various hazards in medical devices for decades. Many medical devices on the market have been validated by recognized institutions such as the FDA [4] and to date are considered very safe. However, it appears that manufacturers are less aware of cybersecurity issues [9]. This is because connecting a medical device to the network is a relatively recent trend. Consequently, these devices are more exposed to new threats that manufacturers were not confronted with in the past.

According to recent studies [9], it seems that basic security features such as integrity, authenticity, privacy, and defense-in-depth are missing in medical devices. One possible explanation for this is that manufacturers lack security experts [10]. Thus, security issues are left to non-security experts. Knowing that the integration of security in critical devices is a difficult task, without security expertise, the design and the integration of the latter

are usually poorly achieved. Furthermore, advanced vulnerability research should be part of the development life cycle of a medical device. However, this is rarely the case, advanced security tests are time and budget consuming making them regularly neglected.

For hackers, this lack of robust security trends is profitable. In recent years, many new attacks have appeared on various life-critical systems. For instance, security researchers were able to maliciously modify the firmware of a real infusion pump [11] and pacemakers [12] because there were not properly signed by the manufacturers. As a result, these researchers were able to implement malware directly in the device. Also, many wireless medical devices do not verify the authenticity and the identity of the host with whom they interact. Hence, a malicious hacker can easily craft a fake application [13] and send malicious commands to the critical device. Many medical devices use a proprietary protocol for communication. Unfortunately, these protocols do not use proper or standard ciphers leaving the data in plain text passing through the network [13]. As these devices are connected to the network, they can also be remotely accessed. Unfortunately, once again, the accesses are often poorly secured, and even sometimes, there is no security at all [14]. On top of that, medical devices are vulnerable to zero-day vulnerabilities such as memory safety issues that result from code and design mistakes [15]. These vulnerabilities can be exploited by hackers to remotely execute malicious code on the device. The seriousness of this type of attack is extreme, it can be achieved without requiring any physical access to the system. In an era of ransomware [16], such poor security devices are profitable to hackers that look for a quick return on investment. Since these critical-devices keep patients alive, it's even easier for hackers to pressure them to pay a ransom.

As it appears, there are various vulnerabilities in medical devices. Countermeasures against these vulnerabilities cover many areas of software and hardware security. This thesis focuses on embedded software security. More, specifically the thesis studies the memory safety issue in life-critical devices. When exploited, memory safety issues allow attackers to take control of a device and make it execute arbitrary malicious actions. While the criticality of these vulnerabilities is severe, and even more when it can be performed remotely, it seems that efficient protections are currently not deployed in medical devices [9, 17, 18].

To address these concerns, this manuscript proposes the following steps. The first chapter covers the state-of-the of various attacks and defenses regarding memory safety. It highlights the particularities of critical embedded systems such as medical devices, why they are exposed to memory safety issues, how memory safety vulnerabilities are exploited, and finally, what are the existing defenses. Chapter 2 confronts the state of the art with the requirements of life-critical medical devices. More specifically, this transitional chapter identifies why current medical devices do not benefit from existing defenses. Then, it exposes the problem: How to implement **practical** and **efficient** defense against memory safety defenses in safety-critical devices while respecting their constraints? The thesis proposes to deal with this problem using two approaches and a life-critical demonstrator to assess them. Chapter 3 exposes SecPump, a wireless insulin pump security system work-

bench used to model memory safety threats and tailored for further security assessments. Chapter 4 presents TrustFlow, the first approach of the thesis. This approach assumes that memory safety defenses can be implemented using both hardware and software. As a whole, TrustFlow is a framework that ensures low-level software protection specifically for memory-constrained embedded systems. Conversely, Chapter 5 presents BackGuard, the second approach of the thesis. Unlike the first approach, this approach assumes that most critical embedded applications are running on a fixed microprocessor that cannot be modified to implement security features. As a result of this assumption, this second approach aims to demonstrate and study the feasibility of memory safety defenses using only software primitives.

1 Background

Summary of the Chapter

This Chapter surveys the memory-safety issue in embedded systems. As this thesis unfolds in the medical area, a brief introduction to life-critical devices is given at the beginning of the Chapter. This introduction describes how these systems work, how they are designed, and why they are currently facing memory safety issues. The Chapter offers an overview of the various attack techniques that are leveraged by an attacker to execute malicious code on a system. Then, it exhibits the state-of-art regarding the existing countermeasures. This state-of-the-art regroups the countermeasures in four categories exposing their benefits, costs, and weaknesses. Finally, synthesis of the state of the art prepares the gap exposure and the thesis approach outlined in Chapter II.

Contents

1.1 The Memory Safety Issue in Life-Critical Systems

1.1.1 C a prominent programming language in critical systems

Life-critical systems such as wireless insulin pumps described are governed by both hardware and software components. One of the fundamental aspects of these components is that they should meet high performance and deterministic real-time constraints to deliver the patients' perfect dose of insulin at the right time. Also, for patient-worn systems, the components must be small in order not to be too invasive. The logical and deterministic actions performed by cyber-physical systems are driven by thousands of lines of code that are executed by microcontrollers. Microcontrollers are often considered as the brain of smart devices. They execute instructions that control and deliver input/output to external components. These external components include drug reservoirs, screens that establish physical contact between the user and the embedded system, sensors, and radio components. Finally, the set of lines of code that lead the logic of the system constitutes the firmware. Its main purpose is to monitor the behavior of the device throughout its lifetime.

According to a recent study made by the IEEE Spectrum 2017 [19], the C programming language is the dominant language in the context of embedded system programming. Most of the existing critical devices' firmware on the market are developed using the C programming language [18]. This popularity can be explained by several advantages that this C offers to critical embedded systems manufacturers over other programming languages:

- *Maturity*: The C programming language has been created more than 40 years ago to provide an alternative to writing code in assembly. Since then it has been widely used in companies and academia. Compared to much newer programming languages [20, 21], the C programming language has the advantage of being supported by a large and experienced community.

- *Readability*: C is considered a high-level programming language in comparison with assembly. It has been created to write data structures and algorithms without relying on complex assembly knowledge.

- *Toolchain support*: Many free C compilers [22], [23], and libraries are available for a wide range of microcontrollers. This means that the C programming language can be easily translated into efficient machine-independent code. On top of that, the current C compilers are very mature and offer a lot of features such as detecting programming errors during compilation and/or code coverage [22]. Also, many free open-source and well-maintained toolchains are available to perform either static [23, 24], and/or dynamic [25, 26] verification of C programs.

- *Modularity*: The C programming language is very modular, thanks to header files it allows developers to export functionality from a C source file and import it in another

file. Consequently, by using modularity it is very easy for a programmer to break a complex program into multiple sub-modules that achieve specific functionalities.

- *Portability*: Many free C compilers are available for a wide range of Instruction Set Architecture (ISA). This means that the C programming language enables developers to write convenient processor independent code that ensures low-level tasks while retaining portability.

- *Efficiency*: The C programming language can easily interface with other programming languages such as assembly. In embedded system programming, it is very common to see routines that are completely written in assembly to efficiently use the hardware. Secondly, one of the most powerful features offered by the C programming language is full control over memory management thanks to pointers. Indeed, by using pointers, embedded system developers can directly read/write from/to memory or allocate it in an efficient, machine-independent way. Consequently, embedded systems programmers can customize the memory usage of embedded systems applications making it inherently fast and optimized in size. Finally, as most of the embedded systems have input/output devices that are directly mapped into the memory, the use of pointers allows instant access to these memory locations.

- *Real-time operating system support*: On the market, the most popular real-time operating systems [27–30] are written in the C programming language. The C programming language is very fast and efficient; it can easily meet real-time constraints. Besides, these real-time operating systems provide C APIs enforcing the use of this language by developers.

- *Determinism*: C programs can be written to be deterministic. This means that it is formally possible to determine the behavior of a program knowing its current state, the previous events, and the environment. Determinism is a very strong feature for critical systems because it allows manufacturers to develop very accurate programs.

- *Standard*: The C programming language is compliant with safety-critical standards such as the DO-178C [31] and MISRA C [32].

It turns out that the C programming language is a perfect language for critical systems. The language is close to the hardware, allowing developers to tune specific routines and memory allocation. This programming language can be used to develop lightweight applications that are very **fast**, **deterministic**, **safe**, and **portable**.

Unfortunately, these benefits come at a certain price. The C programming language does not provide any specific security support against memory access mistakes. This can lead to several security breaches covered in the following section.

1.1.2 The C programming language weaknesses

While the C programming language provides many benefits to embedded system developers, it is often the source of numerous in-memory security flaws due to the misuse of

pointers. To understand the origin of in-memory security flaws, it is important to introduce the basic memory layout of a C program.

A simplified example of a C program is displayed in Figure 1.1.

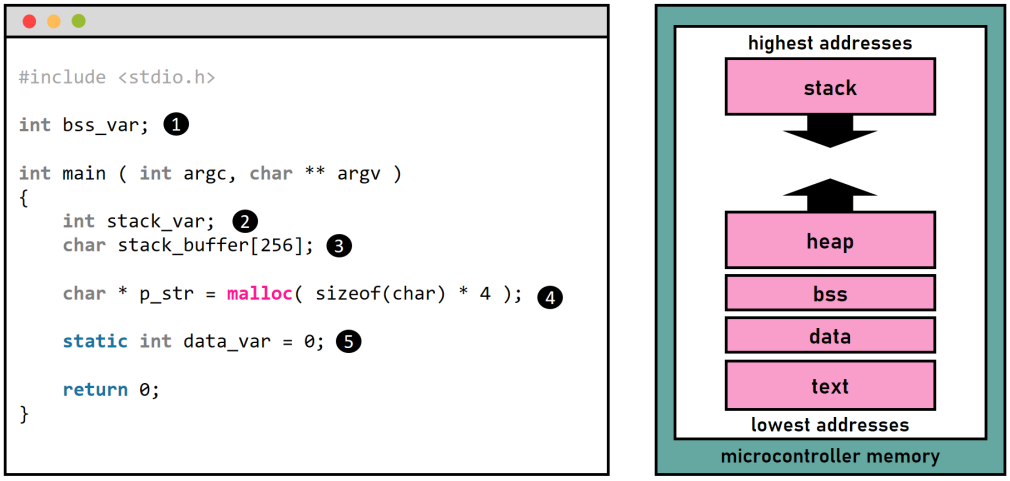

Figure 1.1: C program memory layout

Several variables in the displayed program are annotated with bullets. When compiled into instructions, the program is mainly stored on the "text" memory section of the microcontroller. When executed, this program performs computations using variables that are stored in several areas of the memory.

The variable (1) is an uninitialized global variable. It is stored in the Block Started by Symbol (BSS) area of the memory. This global variable can be accessed throughout the entire lifetime of the program by any function. If it turned out that the variable was initialized it would be placed in the data section.

Both variable (2) and (3) are used within a function, their scope is limited to the life span of the main function. Regarding the microcontroller memory, these temporary variables are placed in the stack area. The stack is a memory structure that follows the Last-In-First-Out (LIFO) concept. In Figure 1.1, the stack is ascending downwards. The stack memory structure is addressed by the stack pointer register of the microprocessor core. This register always points to the memory address of the top of the stack. Two main operations manage the stack: push and pop. When data is pushed, the stack pointer moves to the lowest adjacent address, and the corresponding data is stored. When data is popped, the data stored at the stack pointer address is returned and the pointer moves to the highest adjacent address. Considering our example, when the main function executes, (2) and (3) are pushed on the stack and destroyed when the function returns.

The variable (4) is slightly different from the variable (2) and (3). Indeed, the variable (4) is a pointer stored on the stack but points on a freshly allocated memory which is located in the heap area thanks to a memory allocation function. In Figure 1.1, the heap memory area is increasing upwards. The heap is governed by specific memory allocation algorithms that depend on the "malloc" implementation. Each memory chunk allocated in the heap is accessible by a program through the use of pointers. Besides, allocated

memory chunks can be freed thanks to the "free" function.

Finally, the variable (5), is a static initialized variable. This variable is placed into the data memory section and its value is preserved even when the main function returns. However, this variable is considered private and can only be accessed by the main function.

Knowing the structure of a C program in memory, it is now possible to point out its weaknesses. Unlike high level and/or interpreted programming languages [33, 34], the C programming language does not provide any extra features for spatial and temporal memory safety. The C programming language is thus considered unsafe. Spatial memory safety is defined as the property that all memory objects are always accessed within their bounds. In other words, the C programming language does not ensure that the data copied into a buffer will no longer be larger than the buffer can hold. It follows, that if a developer does not explicitly check for oversized input, an attacker can intentionally provide large enough data that will write past the end of a buffer. As a result, adjacent memory areas that do not belong to the buffer will be overwritten, leaving the program in an erratic state. The most prominent example of a spatial memory error is a stack buffer overflow/underflow. A classical stack buffer overflow is displayed in Figure 1.2. In such a situation, an attacker overflows a local buffer on the stack and overwrites useful buffers' adjacent data of the program. In Figure 1.2, the "stack_buffer" buffer is in the stack area and the "strcpy" function copies a buffer coming from "argv [1]" which is controlled by an attacker. Unfortunately, the "strcpy" function does not check the size of "argv [1]" before copying it to the "stack_buffer" buffer. Consequently, an attacker can exploit this feature to modify some data in the memory that does not belong to the buffer. Thus, if this data is reused later, the behavior of the program may be modified. It should be mentioned that spatial vulnerabilities are nowadays much more complex than the straightforward example displayed in Figure 1.2 [35].

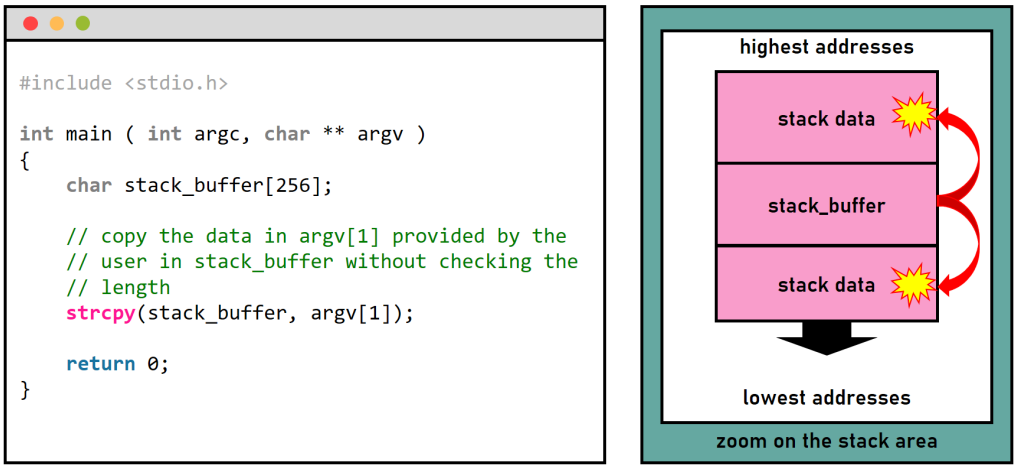

Figure 1.2: Spatial memory issue

Temporal memory safety is defined as the property that ensures that all memory objects are valid at the time they are dereferenced by a pointer. Like spatial memory safety, the C programming language does not prevent any attempts to read or write an object after it has been deallocated. It follows that, after deallocating an object in memory,

if the developer does not explicitly invalidate all pointers on this object, some will still point to the freed memory location. These pointers are called dangling pointers; they can dereference invalid memory location. By leveraging dangling pointers, an attacker can intentionally reallocate the previously freed memory, fill it with malicious data, and dereference a dangling pointer to this location. As a result, the previously deallocated object is reused thanks to the dangling pointer with invalid data causing unexpected memory corruption.

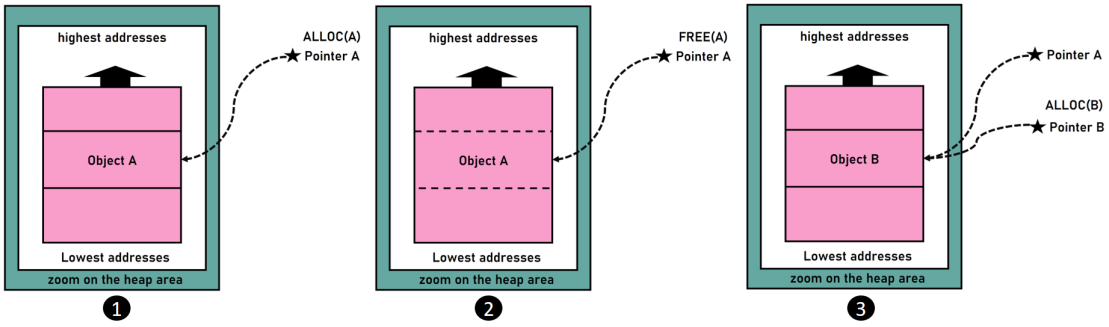

Figure 1.3: Temporal memory issue

A typical dangling pointer scenario is displayed in Figure 1.3. This example assumes that an attacker can use object A and free it thanks to the pointer A. Also, it assumes that object A has previously been allocated by the application (1) and cannot be modified by an attacker. However, the example assumes that an attacker can fully allocate an object B and modify it. By leveraging this dangling pointer issue, an attacker can launch a memory attack in three steps. First, the attacker makes the application free object A in memory (2). Due to the vulnerability, pointer A is still pointing to invalid memory object A. Then, the attacker allocates a new object B and fills it with custom data (3). Finally, the attacker makes the application reuse the pointer A which is pointing on an invalid controlled object B. As a result, the application executes code that belongs to object A but with object B values. As spatial attacks, dangling pointers can be maliciously manipulated by attackers to make an application execute invalid data.

The incorrect verification of oversized input and the usage of freed memory are common programming vulnerabilities related to the usage of the C programming language [36]. Nowadays, buffer overflows and dangling pointers, represent a significant part of the low-level security bugs [15, 37]. Their exploitation has many consequences from crashing a target application to executing user-supplied malicious code. This thesis is chiefly concerned about malicious code execution on critical devices. Indeed, the ability of an attacker to induce faulty behavior into a critical system is a significant threat that can lead to disastrous consequences.

To understand arbitrary code execution on embedded devices, Figure 1.4 exposes the concept of control-flow graphs.

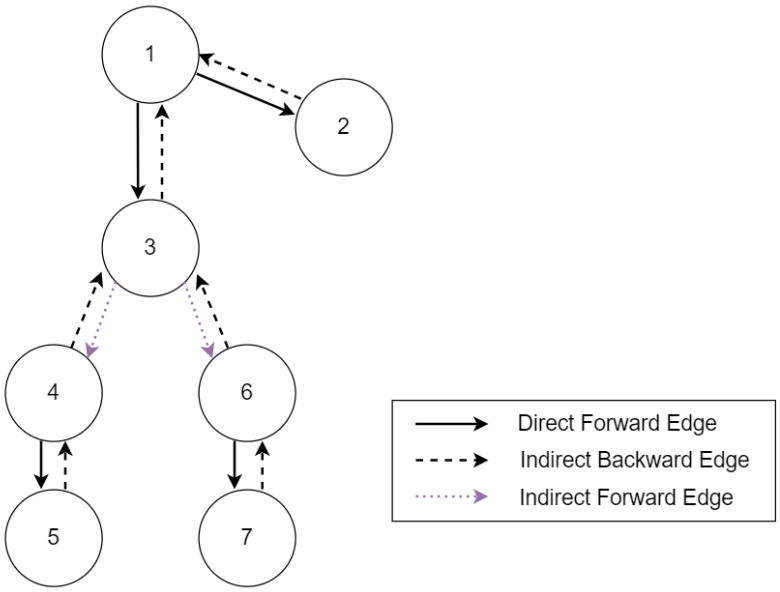

Figure 1.4: Control-flow graph

A typical embedded system application written in C can be represented by a control-flow graph such as Figure 1.4. All vertex in such a graph usually represents a basic block of code. However, to simplify the study the present work considers that each vertex in this graph represents a function. Of course, each function is composed of basic blocks and has an internal control-flow graph as well. Vertices are linked with each other using edges. These edges represent transitions between functions in the graph such as function calls and function returns. A function call is denoted as a forward-edge and is represented in green in Figure 1.4. A forward-edge can either be direct, or indirect through a function pointer. Conversely, a function return is denoted as a backward-edge and is represented in purple in Figure 1.4. Most of the time, a backward-edge is indirect and uses the return address of the caller function that is already pushed in the stack or a register according to the microcontroller architecture calling convention.

It turns out that every function transition in an application is governed by control-flow information. This control-flow information is either immutable in the source code (direct forward-edge) or calculated at run-time through a function pointer (indirect forward-edge) and/or function return address (indirect backward-edge). Unlike fixed control-flow information, indirect control-flow transitions are usually computed at run-time and stored in either registers or memory. Thus, from the moment control-flow information is stored in memory it can be tamper through the exploitation of one of the previously exposed vulnerabilities such as buffer overflows or dangling pointers.

A typical control-flow attack is performed at run-time when data is exchanged between a malicious user and the application. The main goal of a control-flow attack is to hijack the execution flow of an application and make it execute malicious actions. For instance, a spatial memory vulnerability can be exploited by an attacker to write past beyond the limit of a buffer. Thus, the attacker possesses an arbitrary write in memory that may

overwrite control-flow information located close to the buffer. By replacing in-memory control-flow information, an attacker can divert the execution flow on a destination of his choice.

Nowadays, every embedded system written in C can be vulnerable to memory corruption bugs. Even with extremely accurate and advanced tests, the time proved that some memory bugs remain undetected [18, 35]. However, these memory bugs can potentially threaten the security of a system after being discovered by attackers. Unfortunately, there are still a lot of reasons why the memory safety issue is far from being solved in critical IoT devices:

- *Device complexity*: Connected devices especially life-critical devices in healthcare are becoming increasingly complex. These devices have to deal with hard real-time constraints and deterministic behavior in any condition. For instance, an infusion pump in its intended operating mode had to deliver the perfect amount of insulin when needed without misbehaving due to physical external factors. Besides, some embedded software may control physical devices. These physical devices can be the source of high pressure or X-ray control and therefore extremely difficult to debug when controlled by software. Finally, the combination of several physical elements controlled by software increases the complexity of a system, its safety analysis, and the implementation of accurate tests that cover all functionalities without unwanted bugs.

- *The number of lines of code*: There are no existing metrics that give the average bugs per line of code. However, it is often considered that there are between 15 and 50 bugs per 1000 lines of code [38]. Applied to a pacemaker which is around 80,000 lines of code [39], an infusion pump which is around 170,000 lines of code [39] or a magnetic resonance imaging which is around 7 million lines of code [39], there is a high probability that some memory bugs remains undetected after tests and attackers are looking for them.

- *Third-party services*: When developing a critical application, it may require the use of libraries or third-party code. Although it is possible to test these third-party codes before integrating them, this operation may require non-trivial reverse engineering skills due to the unavailability of the source codes.

- *Increased attack surface*: From the three previous points it follows that the attack surface of a system is directly proportional to its complexity, its number of lines of code, and its number of integrated third-party code. Hence, the more complex the system proves to be, the more likely it is that attackers will find vulnerabilities. Also, with the Internet of Things trend, more and more manufacturers are connecting devices to the Internet and critical medical systems are no exception [9]. Unfortunately, the connectivity sharply increases the attack surface. Indeed, with a memory vulnerability, an attacker is now able to remotely control a critical system.

Finally, while the C programming language offers fine control over embedded systems,

this section demonstrated that it comes with a certain security cost. As extensively discussed, the C programming language does not provide any safeguard against memory safety issues. One can argue that the exploitation of memory corruption bugs is not new and the ways to overcome it have been studied over the past twenty years [40]. Unfortunately, it seems that critical devices are still vulnerable to it [15, 37].

1.1.3 Critical systems programming rules

The usage of the C programming language in critical embedded systems differs from traditional desktop applications. Life-critical medical devices can mean the difference between life and death. To release medical devices on the market, manufacturers closely follow important standards such as the IEC 62304 [41], a functional safety standard that specifies the software requirements for medical devices, the ISO 14971 [42] for the risks managements, the ISO 13485 [43], a set of procedures guaranteeing the quality monitoring, and finally, the FDA regulations [44] concerning medical records and device traceability. These certification standards assess the quality and the safety of medical devices software based on the development process [45]. Regarding the implementation, the IEC 62304 [41] strongly enforces the use of coding standards such as MISRA [32] to enforce safety. By respecting these guidelines, manufacturers reduce the likeliness of software hazards and they allow certification of their products. While these standards ensure quality, reliability, and safety, it should be mentioned that they do not remove security issues related to memory safety [15, 17, 37].

To avoid the use of long and complex (sometimes vague) coding standards, the Jet Propulsion Laboratory (JPL) of NASA proposes the "The Power of Ten – Rules for Developing Safety Critical Code" [46]. This paper summarizes some strict development rules to write safety-critical applications using the C programming language. The JPL argues that 10 rules are not all-inclusive but it is enough to achieve measurable effects on software reliability. These 10 rules mostly summarize the highest coding standards such as MISRA [32].

This thesis considers that most life-critical medical devices enforce these rules. Consequently, the security contributions detailed in the following thesis are designed in accordance with [46]. More specifically, the security contributions will at a minimum be able to protect systems that comply with the 10 rules enforced by the JPL. Not all the rules are mentioned in this manuscript. Some of them encourage the improvement of code quality in safety-critical applications. For in-depth details, the thesis refers the reader to "The Power of Ten – Rules for Developing Safety Critical Code" paper [46]. However, bellow is three important rules that have an impact on the outcome.

- *Use of simple control-flow structures*: No "**setjmp**", "**longjmp**", "goto" or direct/indirect recursion: By respecting simple control-flow structures developers improve the clarity of an application and make debugging easier. Moreover, the use of recursion introduces cycles into control-flow graphs that complicate the verification task of static analyzers. Besides, it is not an easy matter to cover all the tests to

determine the upper bound of a recursive function. As a consequence, recursion can induce prohibitive unexpected large usage of stack memory in life-critical systems. Finally, according to [38], a recursive function can always be translated into iterative function.

- *No dynamic memory allocation after initialization*: According to [38, 46] and most of the references in critical software development [31, 32], the use of dynamic memory allocation using “malloc” introduces memory pool fragmentation and potential memory leakage (dangling pointers).

  It follows that the usage of dynamic memory allocation is not deterministic and thus cannot be safe.

- *No more than one level of pointer dereferencing*: The use of pointers is one of the major sources of programming errors, safety, and security issues. Although their use may be practical, it must be minimized and restricted to the smallest scope as possible. Moreover, according to [46], the use of function pointers should be justified and even prohibited. Indeed, when checking an application, the use of a function pointer prevents static analysis tools from determining a call destination and the absence of recursion. However, after being reviewed 10 years later [38], it seems that the JPL considers that "constant function pointers, for instance, stored in lookup tables, pose no risk to safe execution or code analysis".

Considering the previous rules and with section 1.1.2, life-critical application mostly consumes static memory. Consequently, most memory vulnerabilities are located in static regions such as the stack, the data, and the BSS. In critical embedded systems, temporal memory safety issues are restricted by the non-use of dynamic memory allocation.

1.2 Exploitation Techniques

A successful memory attack results from the exploitation of a spatial and/or temporal memory issue. The main goal of a memory exploit is to divert the execution flow of an application to make it trigger malicious actions. This section highlights the various exploitation techniques used by both attackers and researchers to hijack the behavior of a program. The state-of-the-art is carried out in two parts: control-flow attacks and data-oriented attacks. Control-flow attacks aim at diverting the execution flow of a program by violating its control-flow graph whereas data-oriented attacks break the logical behavior of an application without necessarily violating the control-flow graph.

1.2.1 Control-flow attacks

This section discusses various control-flow attacks. More specifically, control-flow attacks mostly rely on malicious code injection and/or code-reuse techniques.

1.2.1.1 Code-injection attacks

Historically, the first control-flow attacks were based on code injection [47]. To illustrate the code-injection process this section refers to Figure 1.5.

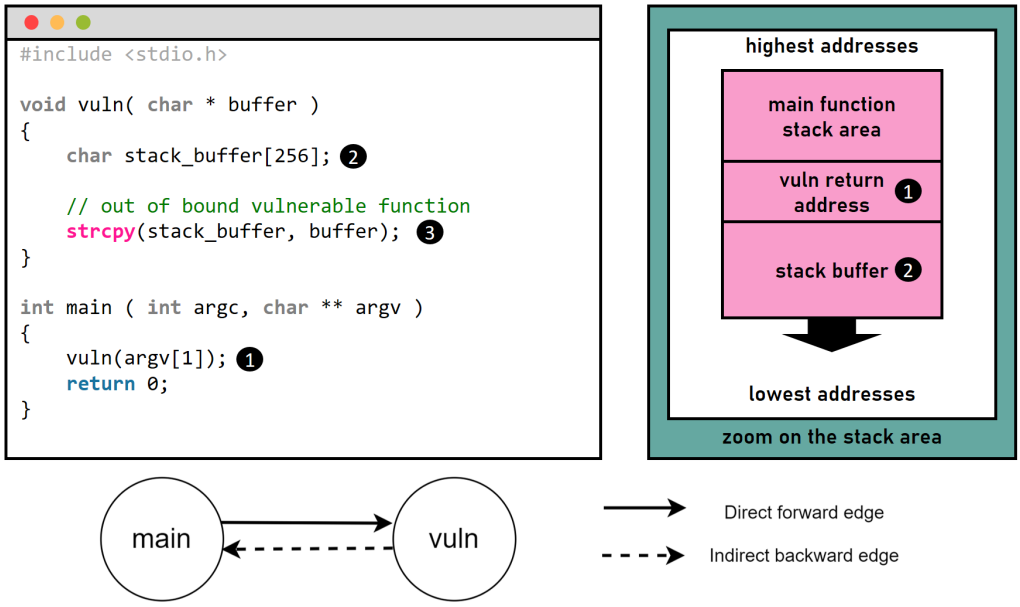

Figure 1.5: Stack-based buffer overflow issue

A trivial example of a buffer overflow vulnerability is displayed on the code part of the Figure. A primary function "main" is calling a "vuln" function with a buffer argument. Such code can be easily represented by a basic control-flow graph with two vertices (on the bottom in Figure 1.5). The zoom on the stack memory area is particularly interesting in this example because this is exactly where an attacker achieves his attack. First, the "vuln" function (1) is called by the "main" function. This has the effect of saving the return address (1) of the "vuln" function in the stack. Indeed, after its execution, the

"vuln" function must return to the "main" function to continue execution. To achieve this, the "vuln" function needs control-flow information stored in memory such as the saved return address. Then, a buffer (2) is also stored in the stack area (2) of the "vuln" function. According to Figure 1.5, the buffer is close to the return address stack. Finally, the "strcpy" function is used to copy data controlled by an attacker in the stack buffer (2). As the "strcpy" does not perform any bounds checking while copying the attacker data in the stack buffer, a spatial memory vulnerability can be exploited. Such a spatial attack is shown in Figure 1.6. First, an attacker injects a malicious code (shellcode) in the stack buffer. Such shellcode is a string of hexadecimal characters that correspond to instruction opcodes that can be executed by a processor. When executed, the main purpose of shellcode is to manipulate registers, functions, or syscalls of a program (originally, the term shellcode was derived from the action of spawning a root shell when executed on a target). Then, the attacker exploits the out-of-bound vulnerability (1) to overwrite the "vuln" return address in memory and replace it with the address of the buffer which contains the shellcode (2).

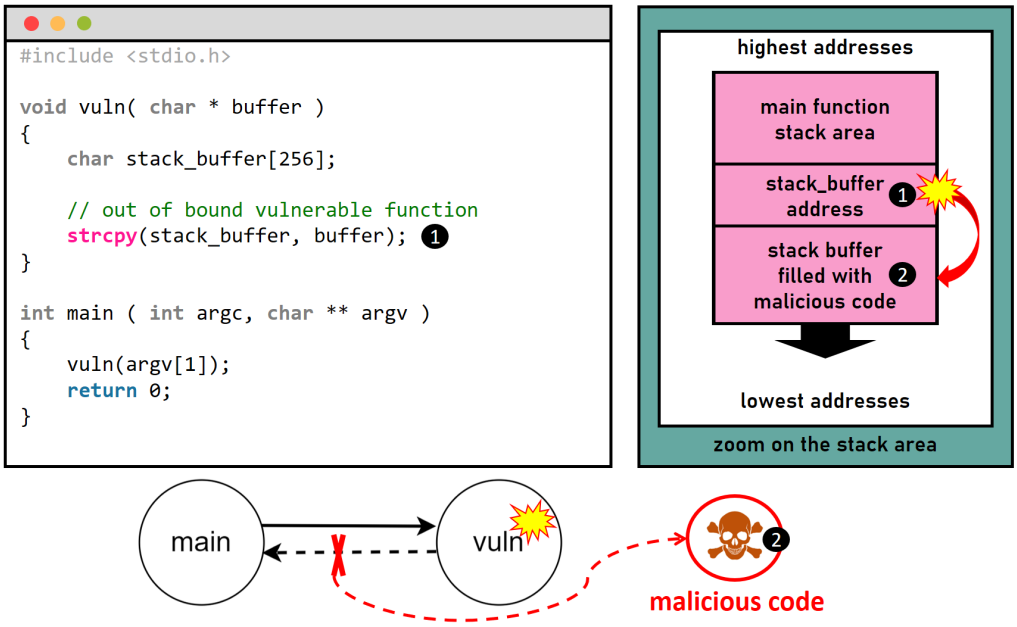

Figure 1.6: Stack-based buffer overflow exploit

As a result, when the "vuln" function returns, its return address is popped from the top of the stack to the program counter register that points on the next instruction to execute. Consequently, instead of returning to the "main" function, the "vuln" function returns to the shellcode previously injected by the attacker (2). Finally, the behavior of the target application is controlled by the injected shellcode.

Code injection attacks were very popular around the 2000s on desktops computers. For instance, the SQL Slammer worm [48] was exploiting a buffer overflow in Microsoft's SQL servers to infect computers. Likewise, the Morris worm [49] exploited a vulnerability in the Fingerd protocol to spread over the network and paralyze hundreds of thousands of computers.

Since then, several countermeasures have been proposed to protect applications against memory safety exploits. Three very famous buffer overrun protections such as Address Space Layout Randomization (ASLR) [50], Data Execution Prevention (DEP), and Stack Smashing Protector [51] are currently widely deployed in most personal computer application. These countermeasures are summarized in Figure 1.7.

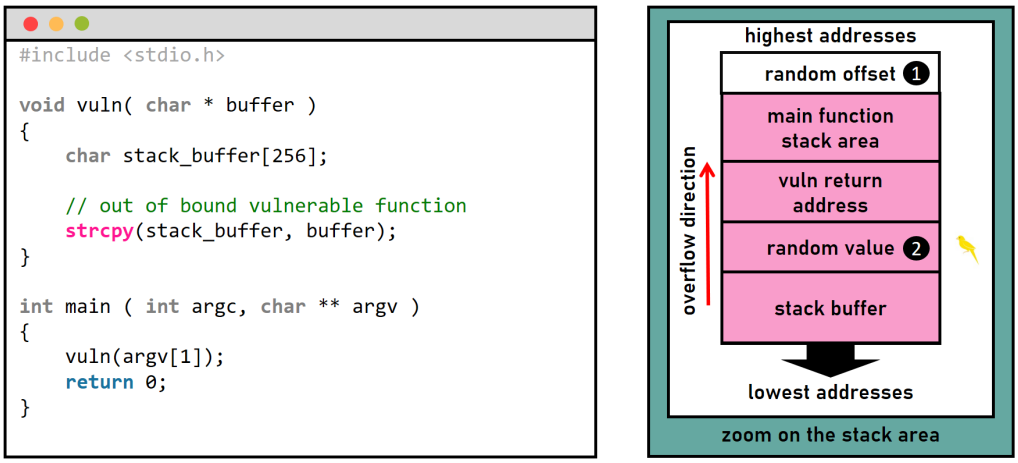

Figure 1.7: Common exploits mitigations

The Address Space Layout Randomization (ASLR) is managed by a classic operating system such as Linux or Windows. It has the effect of placing the stack at different memory offset (1) each time an application is executed. As a result, it is much more difficult for an attacker to predict the address of an injected malicious code in randomized memory areas. To push it further, various types of ASLR are currently available and deeply exposed hereafter.

The Data Execution Prevention (DEP) is a hardware feature that can be managed by operating systems with MMU and/or MPU support. The main purpose of DEP is to define non-code memory regions as not executable. Consequently, any malicious code injected into the stack, in particular, will no longer be treated as instructions.

The Stack Smashing Protector (SSP) is a compiler feature that requires a Random Number (RN). This feature enforces the return address protection of function that manages stack frame objects subject to overflow issues. Each time an application is executed, a random value is placed in a function's stack frame between the return address and the local variables (2). Then, before a function returns, the value of the Stack Protector is checked to ensure that no out-of-bounds write modified the return address.

With the ASLR, DEP, and the SSP, memory corruptions are much harder to exploit on desktop computers. However, there are imperfect. For instance, the ASLR applies to both 32-bit and 64-bit systems. Regarding 32 bits systems, the entropy of the ASLR is very low (up to 8 bits). This means that a maliciously injected code position can be brute-forced in a reasonable time. Also, there is a downside to using canaries. At first, the compiler does not instrument all functions (around 20,5% functions for the Linux kernel [52]), so some of them remain unprotected. Besides, checking a canary value for each protected function introduce extra code that often results in performance degradation.

Finally, DEP prevents attackers from executing malicious code in the data memory areas. Although DEP is effective against code injection attacks, new attacks such as code-reuse attacks [53] completely overcome it.

Regarding embedded systems, it seems that the ASLR, DEP, and the SSP are not well-suited. Unlike desktop computers, embedded systems do not always benefit from an operating system with full MMU support nor Truly-/Pseudo-Random Generators [54]. Hence, embedded applications cannot be effectively protected by the ASLR and the SSP. On top of that, many embedded systems are based on 32 bits' microcontroller systems. As previously discussed, the 32 bits ASLR is weak. Finally, some embedded systems with a Memory Protection Unit [55] (MPU) can define non-executable memory regions to thwart data-execution. However, the MPU does not enforce DEP by default on major embedded systems [54]. The configuration of the latter is left to developers, a task that is not always trivial. As a consequence, classical defenses against code injections are very limited when applied to embedded systems. It is therefore currently still easy to exploit software vulnerabilities on embedded systems using simple code injections [18, 37, 56].

1.2.1.2 Code-reuse attacks

With the emergence of Data Execution Prevention (DEP), code injection attacks became less powerful. Given this, attackers rapidly demonstrated new attacks based on code-reuse to circumvent DEP. These attacks force unexpected executable control-flow paths in an application avoiding any malicious code execution in data areas. A simple code-reuse attack is displayed in Figure 1.8. The stack memory area is snapshotted in vertex 3 which suffers from a memory vulnerability.

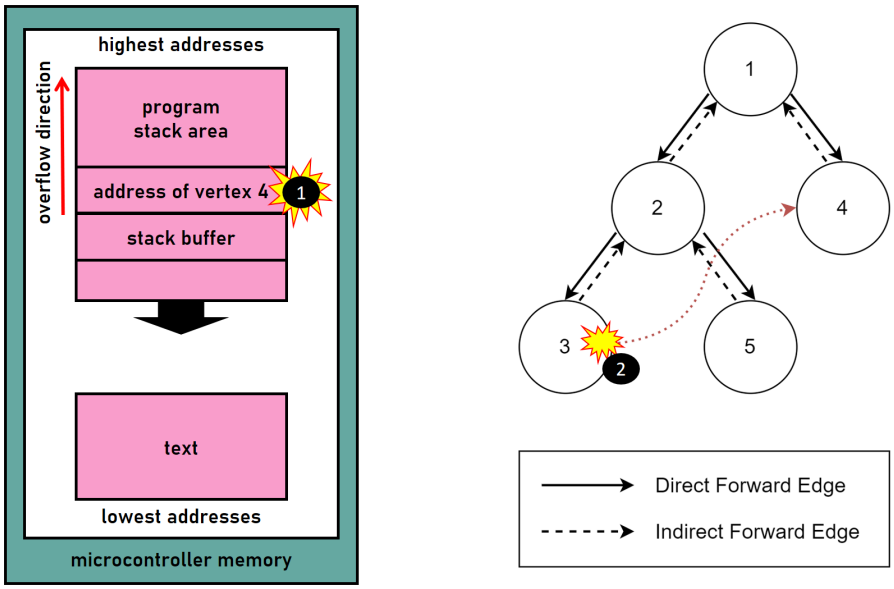

Figure 1.8: Control-flow diversion

First, the attacker replaces the return address (previously pointing to the return location in vertex 2) of the vertex 3 by the address of the vertex 4. Hence, when vertex 3 returns, the execution-flow is redirected in vertex 4 instead of vertex 2. Such a control-

flow violation attack overcome DEP. No malicious shellcode is executed in the stack. The attacker only reuses the application code already located in the executable text memory area.

An exploit technique called Return-to-Libc [57] uses the presented scheme to bypass DEP. The main purpose of the Return-to-Libc attack is to gain execution-flow and chain multiple returns on security-critical functions located in the application libraries. Of course, the purpose of these attacks is to force an application to execute malicious actions. Many "return-into" attacks such as [58] have been inspired by the Return-to-Libc scheme. However, "return-into" attacks suffer from minor limitations. Indeed, such an attack reuse functions that are part of the program or its libraries. Thus, an attacker may not find the functions he wants to execute his payload. Besides, for embedded systems that rely on a complex operating system such as Linux, the loaded shared libraries are often located in an executable area of the stack. Consequently, due to the Address Space Randomization (ASLR), the position of function libraries may change over time limiting the capabilities of exploits. Finally, bare-metal systems use statically linked libraries. It is frequent that compilers embed whole libraries without removing dead code, giving leeway to the attacker to build a working payload.

To circumvent the limitation of "ret-into" attacks, S. Krahmer introduced a new exploit technique [59] which consists of redirecting the execution flow of a program on short sequences of instructions instead of functions. Later, this innovative exploit technique is generalized by Shacham [53] into the so-called Return-Oriented Programming (ROP) attack. The main purpose of ROP attacks is to divert the execution-flow of a target application on a short sequence of instructions that end with an indirect branch. These short sequences are called "gadgets" and are located in the code of the application itself. The general idea is to make an application executing a gadget and use the branch at the end of the instruction sequence to transfer the execution-flow on another gadget. Consequently, given a large application with many instructions, an attacker may find enough gadgets to mount a Turing-complete language. By using this language, the attacker may be able to make the application successfully executes any malicious action he wants. It worth mentioning that the term Return-Oriented Programming comes from the fact that the first attacks used gadgets ending with a "ret" instruction. This instruction allows popping the top of the stack in the program counter register and transfers the control-flow to another gadget. An overview of a ROP attack is represented in Figure 1.9.

First, an attacker exploits a vulnerable buffer (stack-based in Figure 1.9). Then, the attacker injects a sequence of pointers (1), (2), (3). In Figure 1.9, this sequence first overwrites the return address of the vulnerable function (1) on other data higher up in the stack (2)(3). These pointers dereference gadgets located in the text section of the program. When the vulnerable function returns, the execution-flow of the application is redirected to the address pointed by the first gadget pointer(1). Then, when the first gadget returns, it transfers the execution to the next gadget (2) higher up in the stack using a "branch" [60, 61] or "ret" instruction. Finally, the process is repeated for (3), as well as all gadget addresses potentially injected by an attacker. In conclusion, by chaining

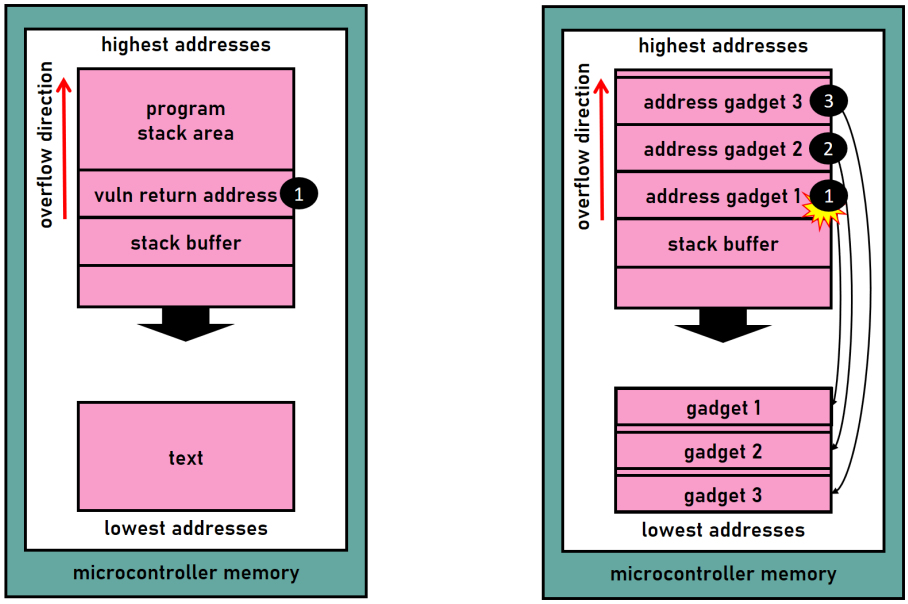

Figure 1.9: Return-Oriented Programming attack

gadgets into a so-called "ROPchain" an attacker can trigger any arbitrary code execution.

Although the return-oriented programming attack was initially introduced on the x86 architecture [53], the technique has been quickly extended by researchers to target many other architectures such as ARM [53], SPARC [61], Atmel [56], PowerPC [62] that are widely used in embedded systems. As a result, small Internet of Things devices based on exotic architectures are no exception to code-reuse attacks.

The major issue with code-reuse attacks and ROP, in particular, is that it overcomes the common protections against code injection attacks (ASLR, DEP, SSP). First of all, ROP is based on gadgets found in the executable text section of a program. Thus, no malicious code is executed on the stack bypassing the DEP protection. Secondly, many systems only benefit from a partial ASLR. This means that the text sections of programs are fixed from one application run to the next. As a direct consequence, the position of the gadgets in memory is fixed, allowing adversaries to deploy large scale exploits. Concerning systems that benefit from the full ASLR (randomization of the entire memory space of an application), it is often found that vulnerability is associated with a memory disclosure [42]. From this memory leak, an attacker can derandomize the vulnerable application memory layout and therefore, compute on-the-fly-all the memory addresses of the gadgets to achieve a successful attack [42]. It is worth reminding here that all embedded systems do not benefit from ASLR. Therefore, these systems are even weaker at mitigating ROP attacks. Finally, under certain conditions, A. Bittau et al. [57] demonstrated through the blind Return-Oriented Programming technique that the stack canary can be brute-forced. The blind return-oriented programming attack works particularly well against an application that handles several processes. According to [57], the crash of a process does not always cause the crash of the whole application as well as a reset of the canary value. As a result, if the vulnerable process restarts, it is, therefore, possible to replay the exploit several times and brute force the canary value byte by byte. While this technique is noisy,

it demonstrates the non-infallibility of stack canaries. Of course, the protection offered by a stack canary requires the support of a TRNG/PRNG [63] which is not always acquired for embedded systems [54].

That concludes the discussion on code-reuse attacks. Of course, this section does not tackle all the advanced exploitation techniques in the literature. They would be exposed when evaluating the state-of-the-art of existing memory safety defenses.

1.2.2 Data-oriented attacks

This section introduces the data-oriented programming attacks, a powerful class of attacks that exploits memory vulnerabilities to modify the behavior of an application without violating its control-flow graph. Indeed, diverting the control-flow graph is not the only way to achieve a successful attack. Usually, an attacker aims at modifying the behavior of an application to unlock security restrictions or gain more control. Consider, for instance, the code snippet with its the control-flow graph in Figure 1.10.

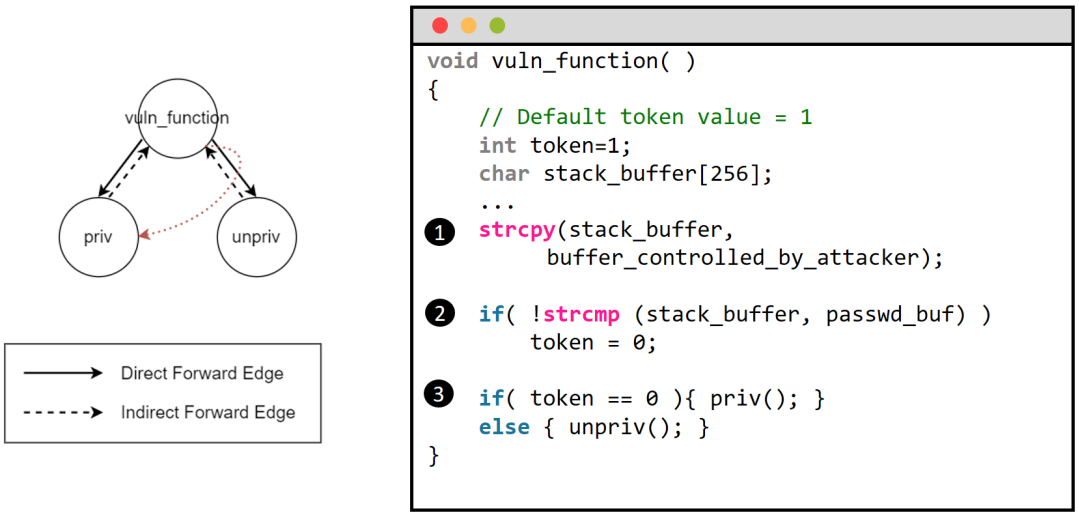

Figure 1.10: Data-oriented attack

There is an obvious out-of-bound vulnerability in (1). Also, the "vuln_function" is calling both the "priv" or the "unpriv" according to the providing password. The token variable is placed in the stack memory, very closed to the vulnerable buffer. As a result, an attacker can maliciously set the token variable to zero by exploiting the spatial vulnerability in (1). Of course, in this case, the comparison in (2) may fail. However, as the token has been maliciously set to one in (1), the comparison in (3) succeeds allowing the attacker to reach the privileged function. The important point is that a successful attack can be achieved without necessarily violating the control graph. Besides, data-oriented attacks such as the one presented in Figure 1.10 are very silent. They don't induce unexpected control-flow path and they can be very effective just by modifying a couple of bytes in memory.

In their paper, S. Chen et al. [64] demonstrated that data flow attacks targeting security-critical non-control data are realistic. Through their work, they identified many

security-critical data in real-world software that can be tampered resulting in compromises equivalent to control-flow attacks. According to [64] this critical data includes configuration data, user input, user identity data, and decision making data such as in Figure 1.10. In the case of an insulin pump, the amount of medication to be delivered to a patient is critical data.

According to the previous description, it seems that the expressiveness of data-oriented attacks is more limited than control-flow attacks. However, H. Hu and al. [65] demonstrated that data flow attacks are Turing-complete. They introduced a generic exploitation method called data-oriented programming. This technique requires two elements: data-oriented gadgets and a gadget dispatcher. Data-oriented gadgets are sequences of instructions that manipulate and perform operations between registers and either load or store their results in/from the memory. These gadgets should respect the control-flow graph of the program and operate on data location controlled by an attacker thanks to a memory error. The purpose of the gadget dispatcher is to connect the data-oriented gadgets. To achieve this, the gadget dispatcher is composed of both a loop and a selector. At each loop turn, the selector is used by the attacker to select and activate gadgets that perform operations on the data. Thus by chaining several loops and activating different gadgets, [65] demonstrated that they could build a Turing complete language allowing them to perform any operation on any data in memory. According to their results, 8 out of 9 real-world programs contained data-oriented gadgets, and 2 were fully exploitable using data-oriented programming. These results demonstrate that attacks on data flow becomes increasingly practicable and empower more and more attackers.

So far, data-oriented attacks [64,66,67] are powerful enough to bypass DEP, ASLR, and SSP. Protecting systems against data flow attacks is thus a complex problem. First, data-oriented attacks do not violate the control-flow graph of an application. Consequently, it is difficult to distinguish legitimate access to resources from illegitimate ones. Then, as applications are becoming increasingly complex, the amount of critical data increases proportionally. For performance reasons, it is impractical to protect every critical data within an application. Finally, during the software development phase of a program, it is extremely difficult to determine which data is critical or not. Some data may appear non-critical, however, in the hands of an attacker, it can become devastating. According to [65] it seems that a simple loop iteration data can be reused for malicious purposes. Nowadays, most of the countermeasures against data-oriented attacks are still at the research stage and are discussed later in this thesis.

1.2.3 Real-world exploits

The preceding sections highlight the various techniques used by attackers to hijack a vulnerable program. While several exploit examples were given, it is not discussed whether control-flow or data-oriented attacks have already been applied to critical medical devices. Also, do these exploits exist in the wild on real applications? Which exploit technique is the most commonly used? The purpose of this section is to demonstrate that medical devices are no exception to memory vulnerabilities. Unfortunately, threats such as attacks

on control-flow or security-critical data are awfully real on critical medical devices. Recent research on Smiths Medical Medfusion 4000 pumps [18] has uncovered several stack buffer overflows in the real-time operating system of the device. Such critical vulnerabilities allow an attacker to remotely take control of the pump. This research uncovered an embedded system that does not benefit from the ASLR, DEP, and SSP. As a result, a successful complete exploit is achieved with a simple code injection attack. This vulnerability also underlines what was previously exposed in section 1.1. Indeed, the root cause of the vulnerability is the use of a third-party real-time system, itself having a certain complexity and a consequent number of lines of code. It must be stressed that this example of memory corruption is only one among others. By digging on different Common Vulnerability Exposure (CVE) databases it is straightforward to find many critical vulnerabilities on medical systems [15, 17, 18]. Another important point is that the exploitation techniques explained in the previous sections are very categorical. In reality, the exploitation techniques of the attackers are hybrid. Real-world exploits combine code-reuse, code injection, and data-oriented attacks to bypass the defenses of a device. For instance, in the presence of an extremely protected system, an attacker tends to combine techniques such as code-reuse, and/or data-oriented programming to first overcome the system protections such as DEP. Then, in a second time, the attacker performs a code injection attack to execute its malicious shellcode.

Studying the life cycle of an exploit makes it possible to realize the importance of memory-based defenses. According to [35], after being discovered, an exploit has a lifespan of 6.9 years. This means that even if a patch is released by manufacturers, many systems are never updated and remain exposed to security threats. Furthermore, after a critical vulnerability discovery, it takes an average of 22 days for hackers to develop a working exploit [35]. This means that attackers are moving fast. Consequently, the current memory-based defenses can be significantly improved to decrease the criticality of vulnerabilities and increase the development time of working attacks. Finally, according to [35], "given stockpile of zero-day vulnerabilities, after a year approximately 5.7 percent have been discovered and disclosed". This underlines that many vulnerabilities are currently not public and work in the shadow threatening all types of systems. It is, therefore, necessary to innovate in defense mechanisms to break functional exploits techniques and harden embedded systems software as much as possible to face the next Internet of Things era.

1.3 Existing Defenses

The previous section outlined an overview of memory-based attacks. Awareness of attacks is worthwhile to design and understand efficient defenses. In this section, the state-of-the-art regarding advanced defenses against software attacks is exposed. Through this study, new advanced attacks are also presented and complement the ones discussed in the previous section. Indeed, with the advent of new countermeasures, innovative and more robust attacks are constantly developed.

Figure 1.11 summarizes the various attacks previously discussed. To remind, an attack often occurs in two stages. The first stage is the corruption of in-memory data. This data, may or may not be a control-flow data. The second stage is the execution of the payload that diverts the normal behavior of the target system. In Figure 1.11, a code injection attack (control-flow violation) diverts the vertex 3 to A, then B (A and B represents maliciously injected code). Equally, a code-reuse attack redirects the execution flow from vertex 2 to gadgets located in vertex 5, then 7. Finally, the Figure displays a data-oriented attack that forces the application to take the path to vertex 4 instead of vertex 6.

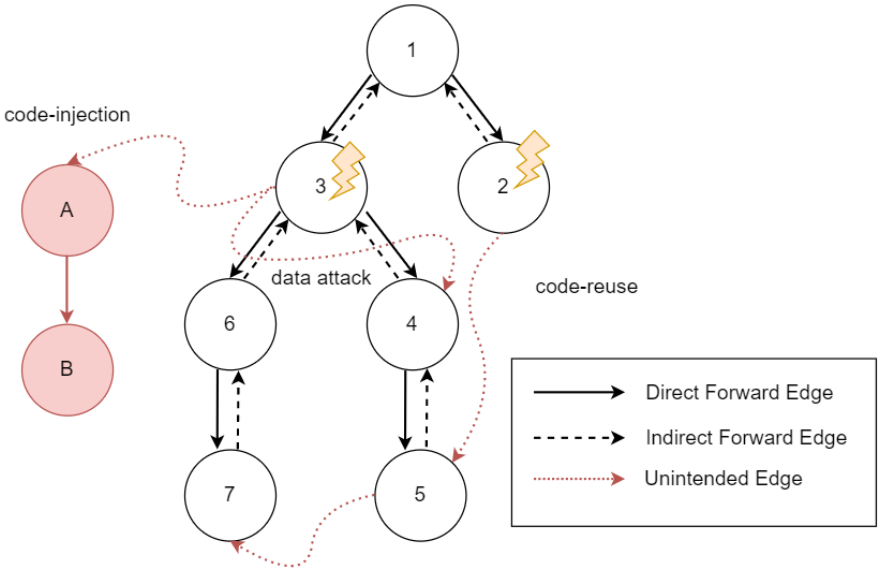

Figure 1.11: Memory safety attacks

From the observation of the control-flow graph, primary defense mechanisms can be easily deducted. One type of defense could be to force the program to follow its control-flow graph. Another would be to rely on heuristics to monitor the execution flow of the program in real-time. If the latter deviates too much from its normal functioning, malicious behavior can be suspected. Randomization may also be an efficient method to prevent code-reuse attacks. Following this idea, each application has the same high-level code but different low-level instructions. Consequently, a working exploit on an application may be difficult to deploy at scale. Finally, it seems that defenses based on data integrity are much harder to design when thinking in terms of control-flow graphs. A concept that would probably reduce data-oriented attacks would be to monitor critical data integrity

over time. The following sections explore these various protection mechanisms regrouped into 4 sections such as control-flow integrity, heuristic defenses, software diversity, and data-flow integrity.

As this work targets critical embedded systems, the following state-of-the-art gives more importance to hardware/software co-design countermeasures. However, this work also gives credit to the defenses implemented on desktop computers. In the end, all the concepts are compared with each other to expose the gap regarding life-critical systems defenses and our approach.

1.3.1 Control-flow integrity

Control-flow integrity is a practical defense introduced by M. Abadi [68] to mitigate control-flow attacks. Control-flow integrity ensures that an executed path in a program always complies with the original static control-flow graph. Figure 1.12 displays a control-flow graph where each indirect transition in the graph is checked to ensure that a legitimate path is taken. There are several ways to implement control-flow integrity providing both a different **policy** and a different **accuracy**. Some existing implementations are backward-edge oriented. This means that the control-flow integrity **policy** guarantees that all backward-edges are always reaching a valid destination. Conversely, there are control-flow integrity **policies** that ensure that all forward-edges always reach a valid vertex (forward-edge control-flow integrity). Of course, the combination of the two equally exists and provides a much more complete defense. The **accuracy** of a control-flow integrity solution is defined by the number of valid destinations a branch can reach. For instance, in Figure 1.12, it appears that vertex 6 can either return in vertex 4 or vertex 5.

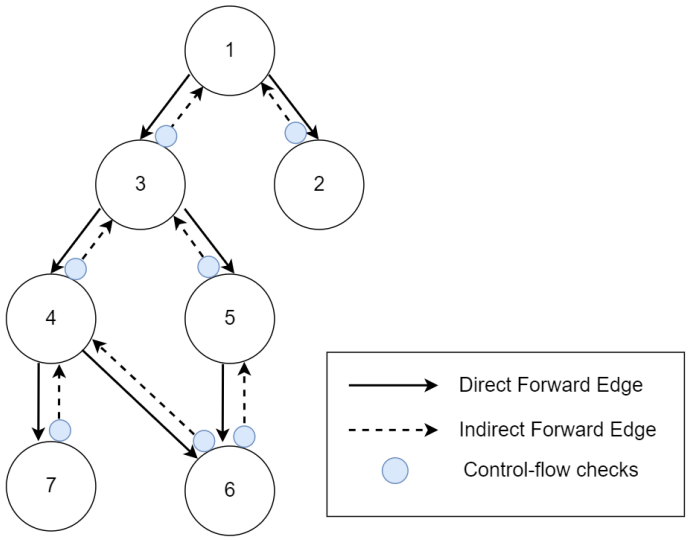

Figure 1.12: Control-flow integrity

A fine-grained control-flow integrity solution ensures that the vertex 6 returns to its more recent calling vertex. On the other hand, a coarse-grained control-flow integrity solution only ensures that the vertex 6 returns to one of the two vertices which are 5

or 4. Although coarse-grained control-flow integrity is less accurate than fine-grained control-flow integrity, it still degrades a wide range of control-flow attacks. However, it does not prevent an adversary from slightly modifying the program execution flow without breaking the control-flow graph. There is a metric titled Average Indirect target Reduction (AIR), introduced by bin-CFI [69]. This metric is commonly used to evaluate control-flow integrity accuracy. The AIR is given by the formula 1.1:

$$AIR = \frac{1}{n}\sum_{j=0}^{n}(1 - \frac{T_j}{S}) \tag{1.1}$$

Where n is the number of indirect control-flow transfer in a program, S the total number of target addresses that can be reached by a branch (it is the size of the binary). T_j represents the number of addresses that can be reached by a transfer j restricted by control-flow integrity protection. It seems that if the percentage given by this formula is close to 99% the control-flow solution can be considered as fine-grained [70]. Conversely, a very coarse-grained control-flow protection has an AIR close to 80% [70].

1.3.1.1 Label-based control-flow integrity

One solution to achieve control-flow integrity is to use immutable code labels [68, 71]. A label is a unique value that can be used to identify a vertex in a control-flow graph. During the execution, it is possible to verify the destination label value of each transition to ensure its validity. Any deviation from the original control-flow graph results in an invalid label destination location that terminates the program. To assign a unique label to each vertex, one solution is to inline it in the assembly code of its corresponding function or basic block of code. Due to the immutability of the code (text section) in a program, the label cannot be tampered by any memory-based attacks. Figure 1.13 displays a pure software label-based backward-edge control-flow integrity protection.

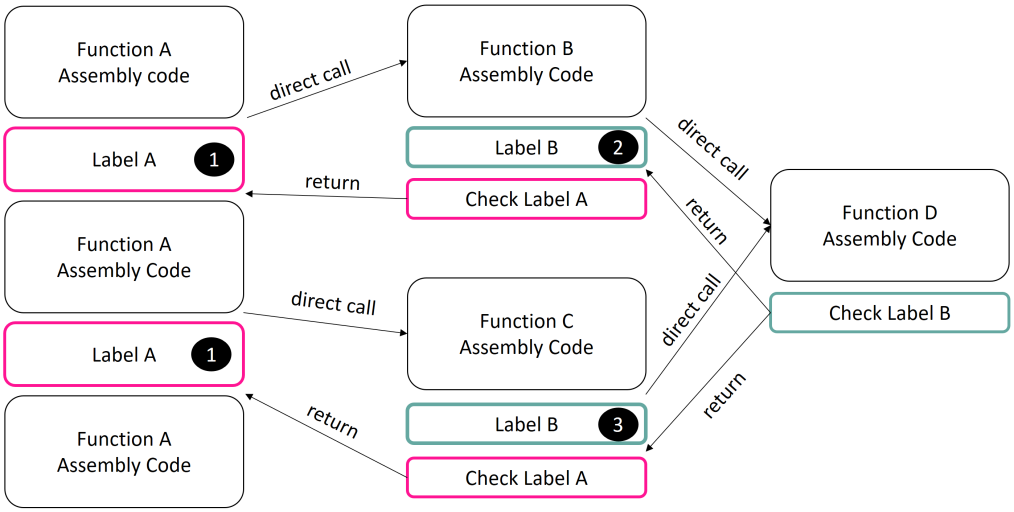

Figure 1.13: Label-based control-flow integrity

Four functions (A, B, C, and D) are displayed in Figure 1.13, they form a simple call

graph. Function A is calling function B, then function C. Functions B and C are both calling the function D. The labels referenced by numbers (1), (2) and, (3) are inserted within the assembly code of each function. More precisely, these labels are call-preceded. This means that the labels are inserted after a call instruction on the top of a basic block of code. It worth mentioning that labels maybe just regular 32 bits memory words. The control-integrity solution works as follows, when a function returns to its caller, the destination label of the return site is checked to ensure that the application is following a legitimate path. For instance, when function B returns, it checks that the preceding location of the return site contains the correct label.

At first glance, such a principle defeats a wide range of control-flow based attacks exposed in the previous sections. However, Figure 1.13 displays inaccurate protection. Indeed, functions B and C are both calling the function D. Thus, functions B and C are sharing the same label B as an identifier giving function D two different valid targets. This makes the presented label-based solution coarse-grained. More precisely, label-based control-flow integrity suffers from several constraints. First, the protection is based on the accuracy of the control-flow graph. Although it is easy to state that a called function will always return to its caller, it is much more difficult to obtain an accurate control-flow graph in the presence of indirect forward-edges. Indirect forward-edges are the result of function pointers. It is statically impossible to determine all the point-to set of this type of pointers and therefore to correctly inline labels. On top of that, embedded systems frequently use interrupts to handle specific routines. An interrupt can be triggered at any time and any location during the execution. Therefore, it is impossible to determine its return point by only using static analysis. The usage of interrupts represents a strong security limitation for label-based control-flow integrity.

As previously discussed, some functions share the same label ID. Unfortunately, it relaxes the accuracy of the control-flow integrity policy as some function can return to multiple callers. One of the positive points of label protections is that the labels are fixed and not modifiable by the attacker thanks to the immutability of the code. Thus, an attacker can neither modify a label nor inject code with a new label in the application data space. On the other hand, label-based control-flow integrity protection does not prevent an attacker from achieving an attack that remains within the control-flow graph enforced by the labels. This well-known attack, titled call-preceded [72] return-oriented programming leverage the label-based control-flow graph to achieve malicious code execution.

In their paper, M. Abadi and al. [68] implemented an inline label-based control-flow integrity protection for Windows applications. They analyzed all indirect control-flow transfers in various applications. Then, they instrumented them with unique labels using Vulcan [73] a binary rewriter framework. They demonstrated that label-based control-flow integrity is easily portable and can be compliant with formal analysis methods. However, their control-low integrity implementation induces a run-time overhead up to 45% on the SPEC2000 [74] benchmark suite. Applied to embedded systems such a pure software label-based control-flow integrity is excessively costly regarding real-time performances. Finally, control-flow integrity introduced by M. Abadi and al. [68] is coarse-grained and

vulnerable to [72].

Label-based control-flow integrity takeaways

- Label-based control-flow integrity identifies each vertex in a control-flow graph with a unique label.
- Labels are protected by the immutability of the code. They restrict the control-flow of a program to its static control-flow graph.
- Pure software label-based control-flow integrity is coarse-grained and may induce execution-time overhead up to 46% [68].

1.3.1.2 Shadow call stack-based control-flow integrity

To overcome the label limitation, M.Abadi et al. [68] suggested the use of a shadow call stack. A shadow call stack handles copies of return addresses at run-time. Usually, a shadow call stack is composed of two elements, a hardware or software buffer and a controller. The hardware/software buffer is an isolated memory area that cannot be tampered by a memory-based attack. This buffer is driven by a logic component called a controller. The main purpose of the controller is to manage the state of the buffer (if it is full or not) and the requests made by the secured software. During execution, each function call and return make a query to the controller of the shadow call stack. More specifically, Figure 1.14 displays the functioning of a shadow call stack.

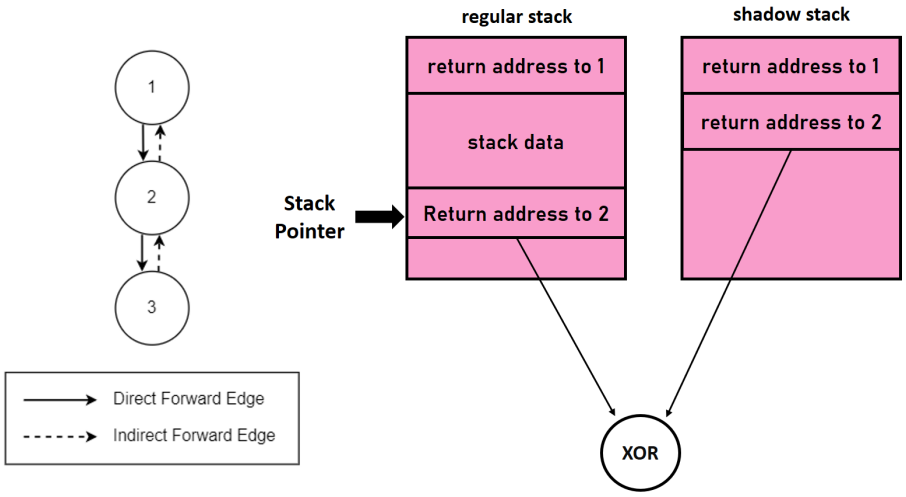

Figure 1.14: Shadow call stack

When a function call occurs, for instance, from vertex 2 to vertex 3, the vertex 3 return address is pushed both in the regular stack and in the memory buffer of the shadow call stack. When the vertex 3 returns, the return address on the top of the stack is used to return to the caller routine. Synchronously, the top return address of the stack is checked using the top address of the shadow stack (displayed using a xor in Figure 1.14). In case of a mismatch, the program ends with an error. Otherwise, the return address is popped

from both stacks. In Figure 1.14, the shadow call stack provides a fine-grained backward-edge control-flow integrity protection. As mentioned by [68] the shadow call stack ensures that a vertex is always returning to its most recent caller.

Many hardware designs that implement various shadow call stacks have been proposed in the literature to protect embedded systems [75–80]. One of the most recurring issues faced by shadow call stack designers is defining the size of the memory buffer. While hardware shadow stacks induce low execution-time overhead, the cost of an additional on-chip hardware memory can be increasingly high. Also, a shadow call stack can only contain a limited number of entries. Thus, if an application call-depth is higher than the size of the shadow stack, an overflow may happen. One solution to tackle this issue is to design a shadow call stack with a capacity that supports the maximum call depth of a wide range of applications. However, this proposition faces several limitations. First of all, a recursive call sequence with unfixed boundaries may tend to overflow the shadow call stack. Secondly, it is hard to set a shadow call stack size that may be suitable for every existing application. It follows that to be properly used, shadow call stacks must be accompanied by programming rules as well as tools to formally verify that software may not produce a shadow-stack overflow.

To overcome the size limitation, [81] proposed an interruption routine that regularly checks the stack capacity and moves the oldest return addresses to a secure memory space. Following the same idea, the NSA [77] proposed a flexible shadow call stack based on the Memory Management Unit. Unfortunately, shadow call stacks have other limitations such as multi-tasks programs. To manage a stack checking coherency in a multi-tasks application, a shadow stack should be managed for each independent task. This type of management is difficult to achieve for pure-hardware shadow call stacks. The proper way to implement it is to assist the hardware shadow call stack with a kernel that restores the appropriate shadow call stacks during context switching.

Another issue is that shadow stacks are not compliant with all possible constructions offered by the C language. One of the well-known issues is the use of *"setjmp/longjmp"* functions. These functions provide a way to perform inter-procedural jumps in an application breaking the standard call/return sequence. Although their use is prohibited in critical applications, it is not uncommon for them to be used to handle exceptions. Unfortunately, after a *"longjmp"* the address on the top of the shadow stack no longer matches the return address of the regular call stack. As a result, this mismatch may induce false control-flow violations.

To support the *"setjmp/longjmp"* issue, some advanced shadow call stack leverage open-hardware architectures with additional instructions and registers. For instance, HCFI [78] modifies the SPARC architecture to enforce both backward and forward-edge control-flow integrity. HCFI introduces 6 customs instructions to the SPARC architecture. Among these 6 instructions, 2 instructions (SJCFI and LJCFI) are specifically dedicated to the *"setjmp/longjmp"* issue. The SJCFI instruction is placed just after a "**setjmp**" function call and carries a unique label that is part of its 8 least significant bits. In addition to holding a label, the SJCFI instruction serves as the landing point for the *"longjmp"*

function. When the *"setjmp"* function returns, the SJCFI instruction is executed. This has the effect of saving the index of the top of the shadow stack in an isolated memory indexed by the label of the instruction. Likewise, the LJCFI instruction is placed after a *"longjmp"* function. Once executed, the execution flow is redirected back on the SJCFI instruction (which is the restoration point of the *"setjmp/longjmp"* procedure). Then, the second feature of the SJCFI instruction is activated. This instruction allows the hardware logic to read the index of the previously saved top shadow stack and restore it.

HCFI [78] is dedicated to bare-metal applications. To benefit from the hardware security feature, an application has to be compiled in an assembly file first. This assembly file is then instrumented using python script [78] to be finally compiled into a binary. While this process is scalable to test small applications, it is difficult to deploy in a production environment. The compilation process is heavy. It requires several steps to generate a final binary. In terms of security, [82] defeats most of the return-oriented programming attacks ensuring a fine-grained backward-edge control-flow integrity. [82] also ensures forward-edge control-flow integrity through labels. The implementation relies on the point-to analysis performed at compilation time. Finally, leveraging the hardware for software security support is a clever strategy to reduce the performance overhead. For instance, HCFI induces less than of 1% execution-time penalty on average.

To deal with the recursion issue, Davi et al. proposed HAFIX [82] an architecture that ensures a fine-grained backward-edge control-flow integrity based on hardware labels. HAFIX extends the Intel Siskiyou Peak architecture with four instructions and an isolated memory area to achieve control-flow integrity. Each application running on HAFIX is instrumented such that every function begins with a CFIBR instruction, each call instruction in a program is followed by a CFIRET instruction, and each return instruction is preceded by a CFIDEL instruction. As a result, each time a function is called, the CFIBR is executed by the processor. This instruction active a unique function label in an isolated memory area. When the function terminates, the CFIDEL instruction is executed before the return instruction. This has the effect of deactivating the label from the trusted memory. Then, the function returns to the CFIRET instruction placed after the call instruction of the caller. This instruction contains the label of the caller function and makes the hardware checking that this function is enabled.

HAFIX tackles recursion by using additional CFIREC instruction. As the CFIBR instruction, the CFIREC is placed at the beginning of a recursive function. Once executed by the processor, CFIREC enables the label of the recursive function. This instruction is linked with a specific CFIREC_CNTR register. Every time the recursive function is called, the CFIREC_CNTR register is incremented. Once the recursive function returns, the processor checks if the function's label is enabled and if the CFIREC_CNTR register is greater than 1. If both conditions are verified, the CFIREC_CNTR register is decremented. Otherwise, the recursion is over and the label is deactivated. In comparison with HCFI [78], HAFIX does not ensure forward-edge control-flow integrity. Also, a return instruction can always return to an active site that is not the most recent caller of a function. Indeed, HAFIX does not prevent an attacker from returning in a caller's active function of

its caller. Regarding the performance, HAFIX induces less than 2% execution-time overhead. Moreover, HAFIX comes with a compiler extension that automatically generates secure code with secure instructions. This approach is more practical to be deployed in software production environments than [78]. It allows error-free security integration for non-security experts.

The key point of this section is that shadow stacks are efficient at providing fine-grained backward-edge control-flow integrity. In other words, a shadow stack can be assimilated into a simple redundancy mechanism for functions' return address. However, this section also reveals that shadow stacks come with several limitations such as the size, the recursion, the way C structures such as **"setjmp/longjmp"** are managed and multitasks applications.

Shadow stacks takeaways

- Shadow stacks provide fine-grained backward-edge control-flow integrity.
- Shadow stacks require dedicated support for multi-tasks systems, recursion, and **"setjmp/longjmp"**.
- The execution-time overhead induced by hardware shadow stacks is low. Unfortunately, hardware shadow stacks are always limited by their size.

1.3.1.3 Branch policy-based control-flow integrity

The majority of control-flow attacks break the static control-flow graph of a program. Code-injection attacks redirect the execution flow of an application on a malicious code located out of the control-flow graph. In contrast, code-reuse attacks divert the execution-flow on several gadgets scattered in the code of a vulnerable program. Thus, monitoring any deviation from the original control-flow graph may be an effective process to catch control-flow based attacks. One way to detect control-flow violations is to monitor each indirect branch at run-time. Such a method, called "branch limitation" or "branch restriction" often requires the help of an external component called a branch monitor. The main purpose of a branch monitor is to analyze each indirect control-flow transfer based on a certain security policy. If an indirect branch breaks the security policy enforced by the branch monitor, an exception is raised. Figure 1.15 displays a control-flow graph instrumented with a branch monitoring protection.

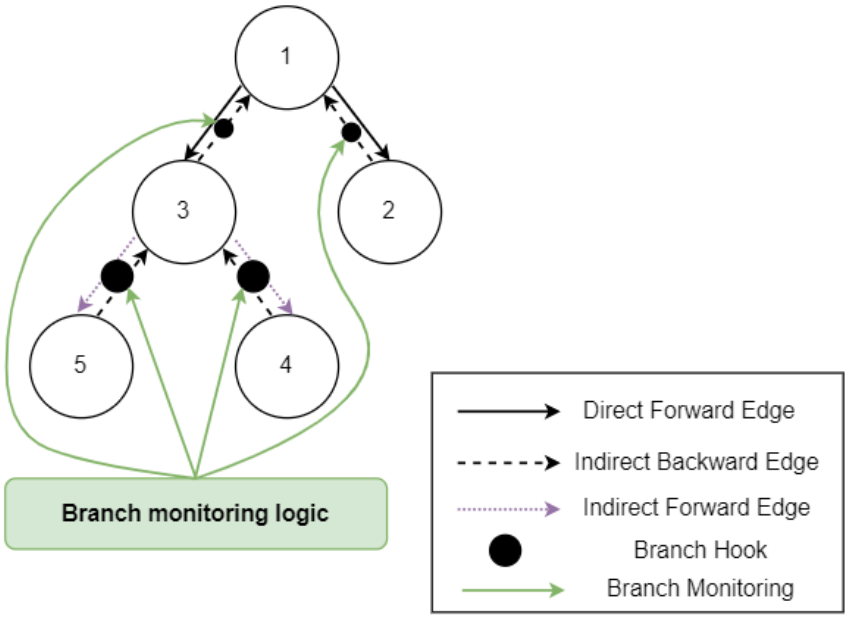

Figure 1.15: Branch monitoring

For instance, a branch restriction security policy can enforce each function return to target an instruction preceded by a call and, each indirect forward-edge to target the beginning of a function. Thus, any indirect-forward-edge targeting the middle of a basic block such as ROP would violate the security policy enforced and be detected.

BB-CFI [83] is an x86-based architecture with a branch restriction policy. BB-CFI uses a hardware control-flow checker to monitor the behavior of an application over time. The control-flow check enforces that every jump instruction is restricted to the beginning of a basic block. Besides, the control-flow checker restricts every function call to the first basic block of a function. To be aware of each valid branch destination, the BB-CFI must be configured with a metadata file that contains the valid destination target of every branch. Of course, this configuration file is stored in secure memory so that it may not

be tampered by any memory-based attacks. According to [84], BB-CFI can thwart every attack from the RIPE benchmark suite, providing fine-grained control-flow integrity.

To dispense with the use of a metadata file such as [83] , W. He [84] proposes BBB-CFI a similar approach than [83] . This proposal checks that the destination target of an indirect branch is preceded by a branch instruction. Indeed, since a branch instruction marks the end of a basic block and the beginning of a new one, checking the instruction preceding the jump target ensures whether the destination is the beginning of a basic block or not. Unfortunately, in the case of fall-through or switch, this policy doesn't work. To overcome this exception, W. He [84] proposes a second rule in which a list of allowed branches not covered by the first rule is maintained and does not raise a control-flow exception. Like BB-CFI [83], BBB-CFI [84] is implemented in hardware on an FPGA. It consists of several components such as a control-flow checker that interacts with the pipeline of a processor. This control-flow checker itself contains two elements; shadow call stacks for backward-edges, and a checker module that enforces rules 1 and 2. More deeply, BBB-CFI uses a 32 entry shadow call stack, and the average amount of memory required to store the metadata file containing the second rule represents 13% of an application. BBB-CFI also induces an overhead of 0.13% at runtime which is negligible. Finally, in terms of security BBB-CFI is confronted with the Benchmark-RIPE [85] suite. All attacks of the suite are countered by BBB-CFI except the return-to-libc attacks.

Intel Control-Flow Enforcement Technology [75] is similar to BBB-CFI. Indeed, it combines a hardware/software branch restriction policy for the forward-edges with a shadow call stack for the backward-edges. Indeed, Intel CET introduces a new ENDBANCH instruction in the x86 instruction set. The ENDBRANCH instruction is used to mark the beginning of a valid destination block for an indirect forward-edge. According to the specifications [70], the Intel x86 processor implements an additional finite state machine in the pipeline that tracks the indirect branches of an application. When a branch instruction passes through the pipeline, the state machine moves into a "WAIT_FOR_ ENDBRANCH" standby state. This state waits until the next instruction passing in the pipeline is an ENDBRANCH to confirm a valid branch destination. If an ENDBRANCH instruction is not seen by the processor, a control-flow exception is raised. Otherwise, the finished state machine returns to its initial "IDLE" state. Currently, no specific security assessment has been performed on Intel CET, according to [71], Intel-CET being similar to [84] provides approximately the same security level.

While [75, 83, 84] prevents control-flow attacks by enforcing branch monitoring, C-FLAT [86] proposes to measure the path taken by an application at runtime. To achieve path measurement, C-FLAT monitors every indirect branch operated at runtime and computes a cumulative hash using the source and the destination address of each taken branch. The whole mechanism works with a prover and a verifier. The prover runs on the system to be monitored and computes the cumulative hash of the paths taken by an application. Then, it sends the cumulative hash result to a remote verifier which checks that the hash is consistent with a list of pre-computed hashes. In case a cumulative hash does not find a match in the verifier hash database a control-flow violation is lifted. C-

FLAT is purely implemented in software and runs on the ARM architecture. Initially, the application which must be protected is instrumented thanks to a script. Each branch raises an interruption that transfers the execution to the prover. The latter computes the cumulative hash using the source and destination address of the jump. Finally, the prover returns the execution-flow to the main application and sends the cumulative hash value to the verifier. In C-FLAT, the prover is placed in an isolated memory (TrustZone [87]), unattainable from the monitored main application memory. It follows that the prover is protected from the monitored application. According to [86], C-FLAT is extremely accurate. Indeed, C-FLAT can also be used to monitor direct branches. Also, C-FLAT can detect non-control data attacks that target loops [65]. However, the accuracy of C-FLAT comes at a certain price. The performance of C-FLAT is measured on a real-time system such as the open-syringe [88]. It turns out that when an application is not monitored by C-FLAT the execution overhead decreases from 72 to 80%. Also, according to [86], the majority of the overhead induced by C-FLAT comes from the hash algorithm, from the switches between the TrustZone and the untrusted memory and, finally, from the interrupt of the measurement engine.

To improve C-FLAT [86], Davi et al. proposed LO-FAT [89] an implementation of C-FLAT but at the hardware level. LO-FAT leverages the open-source RISC-V architecture [90] to implement the monitoring mechanism of C-FLAT. To monitor branches, LO-FAT is incorporated directly into the instruction pipeline of the RISC-V Pulpino processor [91]. This strategy offers several advantages over C-FLAT. First, LO-FAT tracks branches directly at the hardware level without interrupting the software execution. Also, the hardware monitor of LO-FAT computes the cumulative hash in parallel with the application execution. Finally, the monitored application is not instrumented thanks to the branch monitor that recognizes branches in the processor pipeline.

In closing, LO-FAT takes advantage of hardware to fill the lack of C-FLAT. Regarding the security, LO-FAT [89] is as accurate as C-FLAT [86]. To remind, both solutions are fine-grained and able to detect some advanced data-oriented attacks [65]. Unlike C-FLAT, the run-time overhead induced by LO-FAT is very low, less than 5%. Besides, LO-FAT increases the size of a RISC-V Pulpino processor by around 20%. Finally, one of the negative aspects of both solutions is that they do not prevent attacks, they can only observe them. Moreover, these solutions require maintaining a hash database which can be extremely heavy depending on the application to be monitored.

Branch policy takeaways

- Branch policies restrict or monitor the execution-flow according to the control-flow graph.
- Hardaware-based monitors usually induce a low execution-time overhead.
- Branch policies are useless against data-oriented attacks and attacks that remain within the control-flow graph.

1.3.1.4 Trampoline enforced control-flow integrity

As control-flow attacks leverage indirect control-flow transfers, one mitigation is to check their destination using an intermediate trampoline. The protection consists of introducing an intermediate routine (trampoline) between an indirect branch and its destination address. Every indirect branch is first redirected to a trampoline that verifies the destination address, validates it and, redirects the execution flow to the expected function. Figure 1.16 displays the principle on indirect forward-edges.

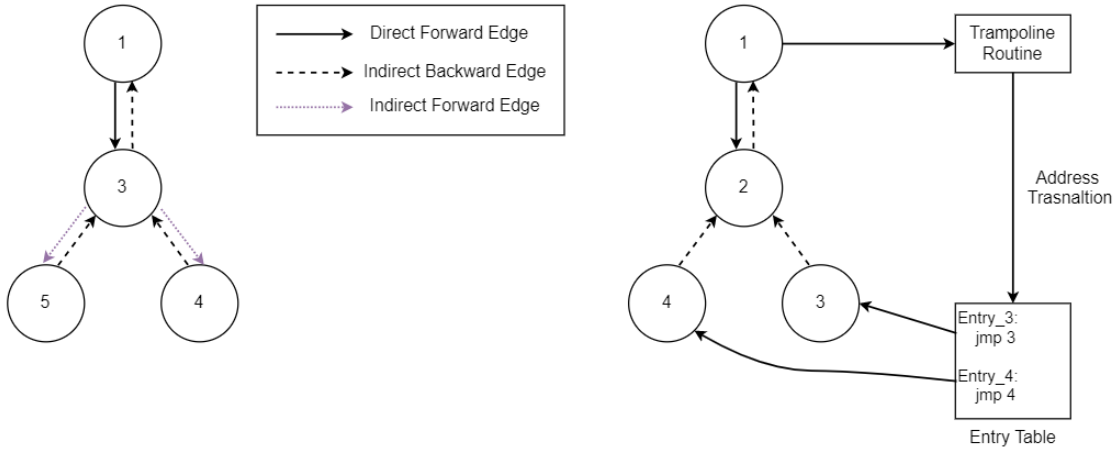

Figure 1.16: Indirect branch verification

The example assumes that forward-edges are determined at run-time by the value of a function pointer. Of course, by corrupting the function pointer's value, an attacker can hijack the application's execution flow. To counter this type of attack, a trampoline is introduced (on the right in Figure 1.16). This time, the indirect calls are first directly redirected on a trampoline routine. This trampoline performs an Address Translation of function pointers to compute an index in a table that corresponds to a valid destination target. Finally, the execution flow is redirected from the table entry which contains a jump to a valid final destination.

Many existing implementations use a trampoline approach to enforce a control-flow integrity [69, 77, 92, 93]. However, among these implementations two categories of trampoline approaches may be discerned: those that protect a program at compile-time [92] and those that act after compilation [69, 77, 93], directly on a binary.

For instance, MoCFI [77], CCFIR [93], and bin-CFI [69] use a trampoline based approach applied after compilation to enforce both forward and backward-edge control-flow integrity. Both [93] and [69] use various static binary analysis and rewriting techniques to redirect the indirect control-flow on injected checking routines. First, these approaches use a custom disassembler to collect all the indirect control-flow transfers. Then, a static analyzer determines all the possible targets of indirect control-flow transfers and constructs a table of every valid target destination. After that, each indirect control-flow transfer is replaced (patched) by a direct branch to a trampoline. This trampoline implements an address translation that transforms the destination address of the indirect control-flow

transfer into a valid function table entry address. Finally, a jump to the table is performed and the control-flow is transferred to the original indirect target. bin-CFI [69] maintains a different table and a different trampoline routine for indirect backward-edges and direct forward-edges. Consequently, an indirect return (backward-edge) cannot target a valid destination from the forward-edge table. CCFIR [93] proposes the same approach but adds a table for security-sensitive function. Both bin-CFI and CCFIR have a reasonable execution-time overhead, around 3.6% for CCFIR on SPEC2000 [68] and around 4% for bin-CFI on SPEC2006 [91]. In terms of security, CCFIR is confronted against 10 real working exploits taken from Metasploit. The results demonstrated that CCFIR can thwart any one of them. Conversely, bin-CFI exposes an AIR of 98,86% which is close to fine-grained protection.

MoCFI [77] follows the principle of both bin-CFI [69] and CCFIR [93] to enforce control-flow integrity. Unlike bin-CFI and CCFIR, MoCFI enforces a fine-grained backward-edge control-flow integrity thanks to a software shadow stack. Also, to compute forward-edge targets with accuracy, MoCFI emulates some parts of the assembly during the static analysis phase. From this static analysis phase, a set of (src, target) couple addresses is extracted. These couples are communicated to a protected Runtime Module. During execution, each indirect forward-edge is redirected to a trampoline which calls the Runtime Module. The Runtime Module checks that the indirect call destination is valid and restores the control-flow. MoCFI has an average overhead of 1.22% for indirect jumps and 7.45% for calls and returns.

Unfortunately, the trampoline approach enforced by binary rewriting has many limitations. Rewriting every indirect control-flow branch, adding trampolines and tables has the effect of increasing the size of the final binary. For instance, bin-CFI [69] increases the average size of an executable by 139%. As a result, this type of solution may be acceptable for a desktop application with unlimited memory but unreasonable for a small embedded system. Second, the accuracy of the control-flow integrity protection is based on the accuracy of the control-flow graph computed by the static analyzer. Unfortunately, it is almost impossible to estimate an accurate control-flow graph by only using static analysis. As a result, the instrumentation is often coarse-grained degrading the security. Another limitation is that trampoline approaches cannot protect embedded system interruptions/exceptions if not using a shadow call stack. Indeed, static binary rewriting techniques cannot predict wherein the code an interruption may be lifted. It follows that they also cannot determine their return location.

Finally, to enforce forward-edge control-flow integrity, Google Inc. [92] proposes to use trampoline instrumentation at compilation time. This approach is more compatible with software engineering concepts and can be easily integrated into a software development life-cycle. The general idea of their approach remains the same as for bin-CFI [69] and MoCFI [77] but is implemented in the GCC compiler as VTV [22] and the LLVM [23] compiler as IFCC. The main asset of using a compiler is that more high-level information is present in the program during the compilation phase and therefore it is much easier to determine the indirect control-flow transfers of an application. Also, the compilation phase

allows optimizing the trampoline routine implementation for each indirect branch. To assess the security level of their protection, [89] introduces the concept fAIR (forward-edge Average Indirect target Reduction), which evaluates the quality of control-flow integrity protection by only taking into account the forward-edges. They measured an average of 99.8% fAIR with a performance penalty ranging from 0.6-5.8% for the SPEC 2006 C++ benchmark.

Trampoline control-flow integrity takeaways

- Trampoline control-flow integrity redirects indirect control-flow transfer on trampoline routines that checks their destination target.
- Trampoline solutions are mostly implemented in software, and some of them require the binary rewriting techniques.
- Most binary rewriting techniques are coarse-grained and induce non-negligible size overhead.

1.3.2 Heuristic defenses

Like current anti-virus software that detects the presence of malware on systems, heuristic analysis can also be applied to detect control-flow attack signatures. The main purpose of the heuristic analysis is to track a program at run-time and recognize suspicious behaviors that look like a code-reuse or a code injection attack. To be accurate, the heuristic analysis requires a set of specific signatures that allows it to assess the risk of a system being under attack. These signatures are determined from the knowledge of control-flow exploits patterns. For instance, Figure 1.17 displays a Return-Oriented Programming attack on the right. On the left, one can observe a list of gadgets (short sequences of instructions that end with a branch) that are scattered in the "text" section address space of the application. These gadgets can be extracted using custom tools such as ROPgadget [94].

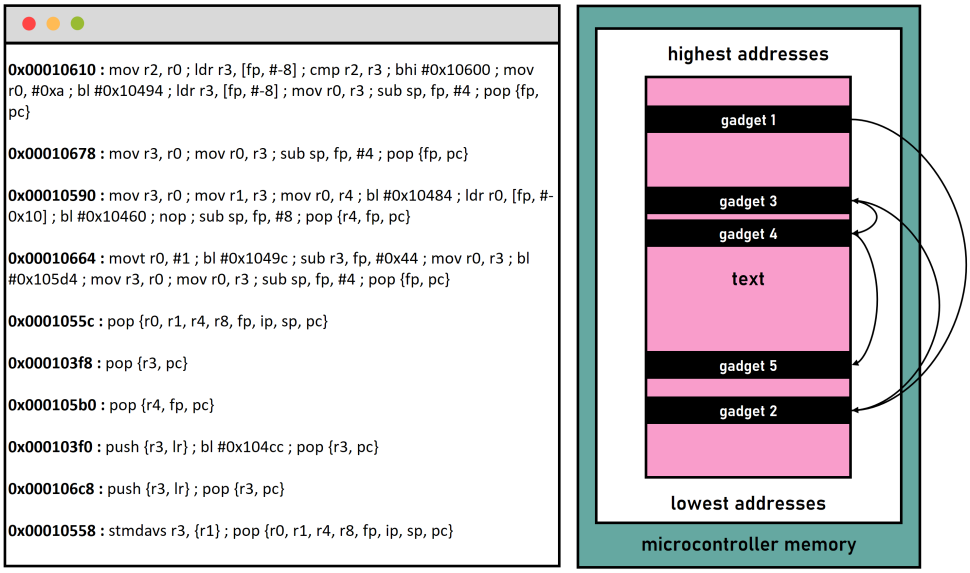

Figure 1.17: ROP signature

In Figure 1.17, some specific code-reuse attack characteristics can be extracted. On the left, it appears that the majority of code reuse attacks use gadgets which are small sequences of instructions (between 1 and 6) ending with an indirect branch. It turns out that the average number of instructions executed between two branches is very low, which is rarely the case in a normal application. Thus, one common signature of code-reuse attacks is a low ratio of instructions between branches. On the right of Figure 1.17, the system under attack is performing a lot of jumps all over the text section in a short period. Thus, a second signature specific to code-reuse attacks is the high ratio of branches over a short time window.

For hardware processor architectures that benefit from a speculative branching unit, code-reuse attacks induce a lot of address misprediction [95]. For instance, a prediction unit such as the Return Address Stack (RAS) cannot predict the destination address of a return in a code-reuse attack because the happening returns are not associated with call [96]. Thus a high level of branches misprediction is a signature of code-reuse attacks. Finally, one of the commonalities of return-into-libc attacks is that they try to execute a

system call or a sensitive function. As a result, the use of one or more sensitive system calls in a non-regular way may also be a sign of an attack.

In summary, by looking for signatures such as the number of consecutive indirect branches, the number of instructions between two branches, the number of branches over a short time window, the misprediction rate of a speculative unit and, the use of system calls, it is possible to determine whether a system is likely to be attacked.

kBouncer [97] is a heuristic engine implemented in Windows that leverages the use of dangerous function/system calls to detect return-oriented programming attacks. kBouncer [97] uses the Detours framework [98] to rewrite a binary and make it stop each time a sensitive function call is performed. Once stopped, kBouncer takes over and checks the Last Branch Record (LBR) unit of an Intel x86 processor. The LBR unit records all the last branches performed by the processor. From the LBR, kBouncer filters the sources and destinations of the last "returns" performed by an application. kBouncer checks that each return destination is preceded by a call. If not, a code reuse attack is detected and blocked. While the estimated runtime overhead of kBouncer is up to 6%, kBouncer suffers from many limitations. Indeed, kBouncer relies on the Last Branch Record (LBR) records; unfortunately, the latter may be overflowed or polluted by the context switches of the Windows operating system. Moreover, kBouncer is only designed to detect Return-Oriented Programming attacks. However, attacks such as Jump-Oriented Programming [60] are as practical and dangerous than Return-Oriented attacks. At last, kBouncer only intercepts dangerous function calls. Unfortunately, many attacks are not always using a sensitive function call in their exploit. Consequently, a wide range of attacks may evade the kBouncer engine.

ROPGuard [99] is an alternative to kBouncer [97] which operates in the same way. As kBouncer, ROPGuard is triggered each time a sensitive function is called in a program. Then, ROPGuard performs more advanced verification than kBouncer to determine whether a program is currently being exploited. First, ROPGuard checks that the stack pointer still belongs to the stack area. Many advanced exploits use the stack pivot technique to move the stack pointer is the heap or the BSS where a gadget sequence can be placed by an attacker. Second, ROPGuard checks if the return address of the called sensitive function belongs to an executable memory area and is preceded by a call instruction. Indeed, many Return-Oriented Programming attacks jump all over the program breaking the classical “call” and “return” structure. Third, ROPGuard tries to compute the call stack of the sensitive function based on the previous frame pointers stored in the stack. Return-oriented gadgets are injected in the memory space overwriting several stack frames and previous frame pointers. Finally, ROPGuard provides additional features such as preventing a program to make the stack executable and simulating a critical function to observe its return address. Unfortunately, by only hooking sensitive function calls, ROPGuard suffers from the same limitation as kBouncer.

To overcome the limitation of both kBouncer [97] and ROPGuard [94], ROPecker [100] enforces three signatures detection methods such as the number of instructions between branches, the number of branches over a time window and, the use of sensitive functions. ROPecker is purely implemented in software in the Linux kernel and operates in two

steps. First, before being executed, the monitored program is pre-processed by ROPecker. This processing phase constructs a gadget database of the application. Then, at runtime ROPecker restricts the execution of functions to their authorized pages by setting the other pages as non-executable. As a result, when a program tries to execute an instruction outside of the active page an exception is captured by the ROPecker engine. This exception triggers the heuristic algorithm of ROPecker. The latter leverages the Last Branch Record (LBR) history to check whether the last source/destination branches correspond to gadget addresses already stored in the gadget database. Besides, ROPecker analyzes the top of the stack and looks for gadget addresses to anticipate future attacks. Thanks to the gadget database ROPecker can detect jump-oriented programming attacks [54] where both kBouncer and ROPGuard fail.

Unfortunately, heuristic countermeasures are imperfect. Heuristic engines are weak at detecting call-preceded gadgets [72]. Also, heuristic engines may be defeated by introducing long gadgets that include several instructions and function calls. The main purpose of these long gadgets is to evade the heuristic engine by obfuscating the signature of a classical Return-Oriented Programming exploit.

To counteract evasion methods, SCRAP [101] proposes a heuristic analysis coupled with a hardware shadow stack. SCRAP is based on a modified implementation of an x86 processor. As ROPecker [100], SCRAP [101] noticed that many heuristic analyses [97,100] only focus on return-oriented programming attacks, but not on jump-oriented programming attacks. Moreover, SCRAP takes into account that the heuristic engine can be evaded using long intermediate gadgets that include calls. To overcome these two issues, SCRAP uses a counter to measure the length of gadgets in real-time. If a gadget length may happen to be under a threshold, a transition is performed in a finite-state machine. Conversely, if a gadget length is over a threshold, a step back is performed by the finite state machine. When a function call is performed, the current state of the finite-state machine is saved in a shadow stack and the finite-state machine is re-initialized. When the function returns, the state is restored and SCRAP continues to monitor gadgets. In terms of security, SCRAP [101] can detect any code-reuse attack, even the evasion techniques that introduce long intermediate gadgets. Also, the detector engine of SCRAP can be programmed to monitor up to 50 instruction gadget lengths without any false positive. Finally, SCRAP is implemented through the PLTsim simulator and reports an execution overhead of less than 2%.

Heuristic defenses takeaways

- Heuristic defenses act like anti-virus, they detect control-flow attacks signature.
- Heuristic defenses are completely transparent to binaries, they often do not require any code instrumentation.
- Heuristic protections are vulnerable to attacks that evade known control-flow signatures.

1.3.3 Software diversity

When a vulnerability is discovered, an attacker aims to develop an exploit and compromise as many systems as possible. The reason why large-scale exploits are usually functional comes from the fact that each instance of an application is the same at the assembly level. Thus, each instance of a program has the same set of gadgets and therefore is vulnerable to the same exploit. To reduce the effectiveness of large-scale exploits, one idea is to diversify the instances of a program in memory.

As previously studied in section 1.2.1.1, the Address Space Layout Randomization (ASLR) is a technique to diversify programs' heap/stack. To remind, the partial ASLR inserts the position of the stack/heap at a random offset each time an application is executed. In contrast, the complete ASLR places the whole application, including the "text" section at a random offset in memory. The original goal of the partial ASLR is to mitigate code injection attacks. In the same way, the full ASLR mitigates the so-called return-oriented programming attacks by shifting the position of the "text" section at each execution.

Unfortunately, the current implementations of the ASLR are weak. Indeed, partial ASLR can be evaded using Return-Oriented Programming attacks. Moreover, code-memory disclosures caused by dangling pointers may reveal the position of code memory pages allowing an attacker to retrieve gadgets on-the-fly [50]. Finally, the ASLR is also vulnerable to partial overwrite attacks and brute-force attacks [102,103] that aim at trying every possible offset to match an address with a fixed exploit. It turns out that the random offset protection used by the ASLR offers a coarse-grained software diversification.

According to the state-of-the-art of P. Larsen et al. [104], fine-grained software diversification can be obtained by randomizing the code structures of an application. To achieve diversification, several hardware/software transformation techniques can be applied to generate several mutations of the same program instance. Figure 1.18 illustrates a simple software diversification technique. Two instances of the same C code are represented in memory. However, the two applications do not have the same structure. The order of the functions in memory is different, and within the functions, the order of the basic blocks is different. It follows that the addresses of the gadgets found in the memory of both programs may be different. While Figure 1.18 only displays two diversification techniques such as function and basic block shuffling, the state-of-the-art of Per Larsen et al. [104] highlights other software diversification techniques such as instruction substitution, instruction reordering, and register substitution. At last, the state-of-the-art even mentioned some advanced obfuscation techniques such as control-flow flattening and opaque predicates to diversify software.

To enforce software diversity, Jackson et al. [105] proposed two extensions of the current GCC [22] and LLVM [23] compilers. These extensions aim at inserting random NOP instructions between the instructions of a program. This has the effect of inserting random offsets between instructions, complicating the deployment of accurate code-reuse attacks. Unfortunately, this model provides low entropy. The order of the functions and the basic blocks remains the same in memory and is frozen after compilation. It results that some

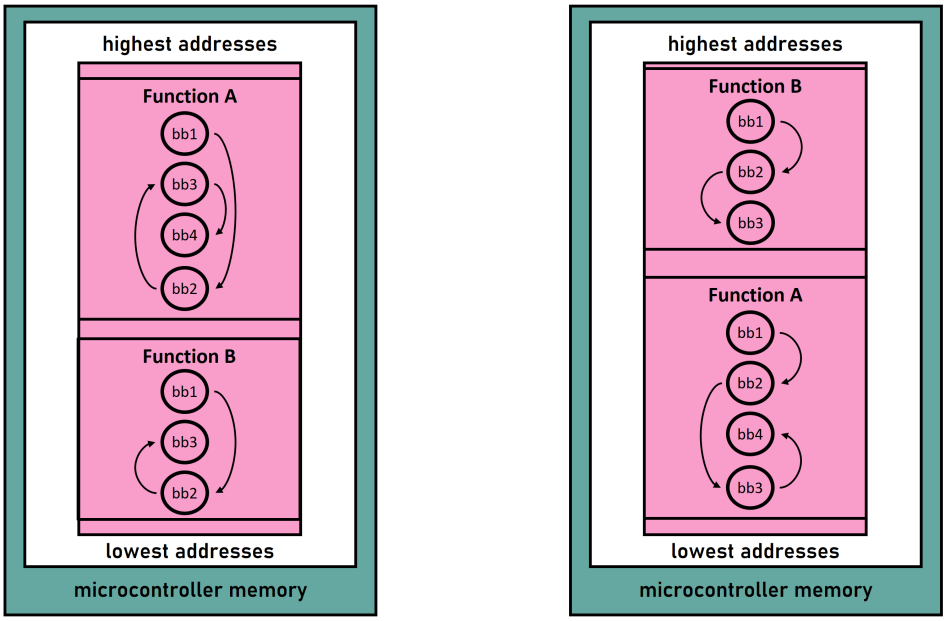

Figure 1.18: Software diversification

attackers may launch brute-force attacks such as for the address space layout randomization [106]. Moreover, in this model, entropy is directly related to the ratio of NOP instructions to be inserted into the program. Consequently, an increase in entropy for the benefit of security leads to a drastic degradation of performance. Jackson et al. measured up to 40% overhead in the SPEC2006 benchmark suite [107].

To improve the insufficient randomness of Jackson et al. [105], C. Kil et al. [108] introduced Address Space Layout Permutation (ASLP). ASLP randomizes the sections of a program during its initialization. To achieve this, C. Kil et al. [108] developed a custom binary rewriting tool that places static code and data segment to a random location in memory and performs permutation of code and data structure. At initialization, the memory layout of the sections of a program is changed by ASLP. For instance, one binary may have the stack above the BSS section, another one may have the stack below the text section. Concerning the random permutations, ASLP collects all the reference of a program at load time, permutes the functions, and the basic blocks within the functions, finally it rewrites the new functions in the binary. ASLP induces less than 1% execution-time overhead on average with 29 bits of randomness. ASLP requires the program to be randomized to have all the relocation information.

To offer finer randomization than [105, 108], J. Hiser et al. proposed to randomize every instruction to a program. To carry out instruction randomization, J. Hiser et al. implemented ILR [109] (Instruction Location Randomization) a framework that shuffles every instruction in a binary. To be executed in the right order, ILR maintains a fall through map where each instruction's successor is independent of its location. This map is used by a virtual machine to fetch the instructions in the right order and executes them. This approach enforces a high level of randomness; every instruction is located at a different address every time. Unfortunately, the fall through map more than double the

size of the application which is unreasonable for memory-constrained embedded systems.

Binary Stirring is a technique introduced by R. Wartwell [110] that dispenses with a virtual machine to maintain software diversification. They randomize both the basic blocks and the instructions at load time. To perform such randomization, they developed a tool that disassembles binaries into an intermediate representation. This intermediate representation is then followed by a load time phase that randomizes and rewrites the instruction of the program. Compared to ILR, STIR is much lighter and does not require a virtual machine. Moreover, STIR has a fairly acceptable run-time overhead of 1.6%.

Unfortunately, countermeasures based on binary randomization suffer from the Just-In-Time Return-Oriented Programming (JIT-ROP) attack introduced by L. Davi [106]. Such an attack leverages a memory disclosure to leak the text section of a program, disassemble it, and build on-the-fly ROP exploits. L. Davi [42] demonstrated that JIT-ROP is powerful enough to break any sort of randomization defenses previously exposed.

To prevent JIT-ROP, one solution is to prevent memory leaks by enforcing an "execute-only" policy on certain memory areas of the program's "text" section [111]. Another method proposed by SOFIA [112] is to cipher every instruction of a program based on its control-flow graph. At run-time, every instruction is deciphered using both the current and the previous program counter. Thus, if the previous program counter does not respect the intended execution-flow, the instruction deciphering fails. As a result, SOFIA prevents both JIT-ROP and various software control-flow attacks. On top of that, SOFIA is also able to thwart hardware fault injection attacks that aim at skipping instructions. Unfortunately, due to the encryption, SOFIA incurs a high run-time overhead up to 106% with a 23.3% penalty on the clock speed of the LEON3 processor.

Software diversity takeaways

- Software diversity produces multiple instances of the same software to prevent exploit development.
- Software diversity is highly vulnerable to memory disclosure attacks and JIT-ROP.
- The size and execution-time overhead induced by software diversity rely on the randomization technique and the entropy source.

1.3.4 Data-flow integrity

The previously discussed defenses protect against the second stage of memory-based attacks. To remind, the first stage of an attack is the corruption of either control or non-control-flow data. The second stage is the execution of a malicious payload by the corrupted program. Control-flow integrity, heuristic analysis, and software diversification take into account that an attacker already corrupted sensitive data. Their main purpose is to harden and/or detect the deployment of a functional exploit. This section takes a different approach, it focuses on defenses against the first stage of an attack, i.e. the initial corruption of sensitive data. However, what can be considered as programs' critical data?

Given the previous sections, control-flow data can be considered sensitive. This includes return address for indirect backward-edges and, function pointers for indirect forward-edges. Once corrupted, this data can be leveraged by attackers to hijack the execution flow. Besides, section 1.1.1 introduced data-oriented attacks, a practical class of attacks that modify the behavior of an application without relying on control-flow data. These powerful attacks unlock security restrictions and give more control of a program to an attacker. To remind, H. Hu and al. [65] demonstrated the Turing-completeness of data-oriented attacks using vulnerable loop variables and arbitrary writes in memory. As a result, non-control data loops and data pointers may also be considered sensitive. Finally, back to the discussion on critical systems, it seems that any non-control data inducing a faulty behavior in a system can be considered sensitive.

Unlike control-flow integrity that restricts the execution flow to the control-flow graph, data-flow integrity aims at protecting programs at the sensitive data granularity. Data-flow integrity can be achieved using cryptography, fat-pointers, isolation, and tagged memory. This section discusses each of these approaches mentioning their accuracy and costs.

1.3.4.1 Cryptographically enforced data-flow integrity

Cryptography is a discipline of cryptology that can be applied to ensure sensitive data-flow integrity. The cryptography certifies the confidentiality, authenticity, and integrity of a message. Thus to protect sensitive data, a cryptographic algorithm with a private key can be applied to attest to the integrity of sensitive data whenever accessed.

For instance, CCFI [113] an LLVM compiler extension proposed by A.J. Mashtizadeh et al. uses a Message Authentication Code algorithm (MAC) to protect the sensitive control-flow data stored in memory. The CCFI compiler ensures that each time a control-flow data is stored in memory its MAC is computed using the cryptographic extension accelerator of the Intel processor and is stored alongside. In the same way, each time a control-flow data is required by a program, its MAC is verified. According to [113], the MAC key is randomly generated at the program start and stored in registers that CCFI compiler reserves. The security of CCFI relies on two assumptions. First, the code never leaks the key because the compiler enforces that the instructions of the program never use the reserved registers. Second, attackers cannot execute the code that accesses the reserved registers because they would have to break control-flow in the first place. CCFI

induces an average of 23% execution-time overhead on the SPEC2006 [107] Benchmark suite for the programs that are only written in C. Although this overhead is high, the cryptographic approach ensures fine-grained control-flow integrity by protecting sensitive control-flow data. It worth mentioning that the solution does not tackle pure data-flow integrity but can be used to it by extending the compiler.

PointGuard™ [114] is another compiler extension based on GCC that protects pointers. To do so, PointGuard™ generates a unique stream cipher key at program initialization. Then, each pointer stored in memory is ciphered using the key. At runtime whenever pointers are accessed they are ciphered/deciphered using the key. In comparison with CCFI [113] that guarantees the pointers' integrity, PointGuard™ uses the confidentiality property of cryptography to protect sensitive data. As a result, if a ciphered pointer is maliciously modified, its decryption may result in an erroneous value causing the application to crash. PointGuard™ [114] reports an overhead ranging from 0 to 20% depending on the application. Unfortunately, PointGuard™ suffers from two limitations. First, PointGuard™ uses an XOR stream cipher with a unique key to cipher multiple pointers. As a result, it is vulnerable to many time pad attacks. Second, PointGuard™ does not enforce any specific security policy against memory disclosure. It follows that a memory disclosure may easily reveal the value of the key. Finally, as CCFI [113], PointGuard™ [114] does not enforce non-control data integrity leaving programs vulnerable to [65].

To overcome the limitation of PointGuard™ [114], S. Bhatkar and R. Sekar proposed Data Space Randomization (DSR) [115]. DSR is a C front end code instrumentation framework based on CIL [116]. Data space randomization ciphers every data in the memory using a mask. To avoid many time pad attacks, this mask is different for every ciphered data. As a result, DSR offers a higher level of security than PointGuard™. Besides, the solution induces an execution-time overhead ranging from 15%-28%, which is close to PointGuard™. Unfortunately, DSR is not binary compatible with external libraries and modules can't be compiled separately. DSR is, therefore, constraining for incremental compilation.

Finally, Pointer Authentication Code (PAC) [117,118] is a hardware extension of ARM 64 bits processors that use Message Authentication Code (MAC) to verify the integrity of pointers at runtime. The PAC hardware extension comes with an extended compiler and provides two sets of custom instructions (PAC* and AUT*) to authenticate pointers at program execution-time. The PAC* instructions trigger a hardware MAC accelerator that computes the MAC of a pointer using QARMA [119] block cipher. The result of the MAC is stored in the 26 upper top-bits ignore of the pointer. This comes from the fact that the ARM64 Linux uses a 40-bit address space by default, leaving 26 upper bits unused.

Once the pointer is re-accessed, the AUT* instruction is used to verify the MAC and restore the pointer original value. If the pointer authentication fails, the processor makes the pointer value an illegal address. When it comes to code pointers and return addresses protection the execution-time overhead induced by PAC is very low (under 0,5% on average) [117]. This low overhead is very encouraging; it demonstrates that fine-grained control-flow integrity can be achieved without inducing unwanted execution-time

overhead. However, concerning the data pointers, [117] claims that the run-time overhead largely depends on the memory profile of an application. The paper [117], reported up to 39,5% overhead.

Cryptographically enforced data-flow integrity takeways

- Cryptographically enforced data-flow integrity uses cryptographic algorithms to authenticate/protect sensitive data integrity.
- By protecting sensitive data, cryptographically enforced data-flow integrity ensures fine-grained control-flow integrity.
- Implemented with hardware support, cryptographically enforced data-flow integrity has a very low execution-time overhead.

1.3.4.2 Fat Pointers

Every programs' variable has a memory location that can be retrieved thanks to its address. Pointers are specific variables that dereference a memory location by its address. They are commonly used to efficiently dereference data structures such as strings, tables, trees, but also to call indirect subroutines (indirect forward-edges). In most memory-based attacks, attackers leverage in-memory pointers to modify the behavior of a program. By modifying the value of a pointer, an attacker can make it dereference another variable that is later reused by the program.

One way to prevent pointer hijacking is the use of Fat Pointers. Fat Pointers are extended pointers that include extra information such as their base and bounds. Figure 1.19 illustrates both a simple pointer and a fat pointer. One can observe that both pointers in Figure 1.19 dereference the same variable. However, the Fat Pointer has additional metadata: a base and a bound. These data refer to the beginning and the limit of the dereferenced object in memory. Hence, each time object A is dereferenced using the Fat Pointer, the validity of the latter is checked using the additional base and bound metadata.

The use of Fat Pointers is an efficient technique to thwart pointer hijacking attacks. Intel MPX [120], Watchdog [121], WatchdogLite [122], Softbound [123], and Hardbound [124] broadly follow the same Fat Pointer approach to ensure memory safety. For instance, both Softbound [123] and Hardbound [124] record metadata for every in-memory pointers. These metadata are stored in a disjoin memory in the virtual address space and they include the base and bound of every pointer. Whenever a pointer is used, the boundaries of the latter are checked using its associated metadata. Softbound [123] is a pure software approach that inserts bound checks at compile time. Softbound is compatible with the C language programming standards, however, the execution overhead cost of Softbound is very high, up to 79%. To overcome the overhead limitation of Softbound, Hardbound implements a hardware extension of an Intel x86 processor with a custom instruction to set the base and bound of a pointer in the disjoin memory. The hardware extension also implicitly monitors each access to a pointer and check its boundaries. As Softbound [123],

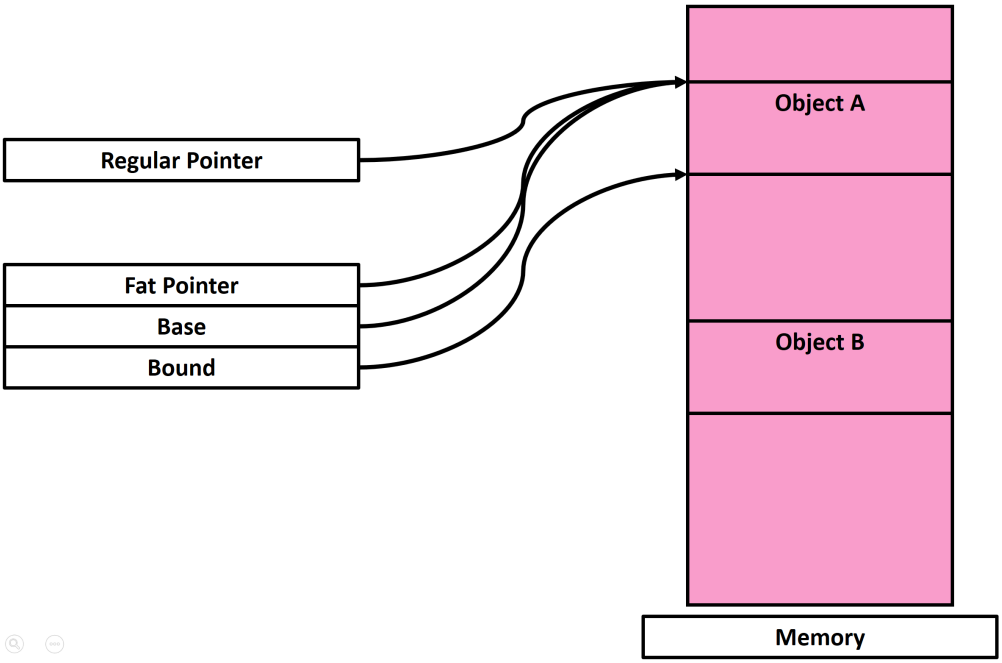

Figure 1.19: Fat Pointer

Hardbound [124] can thwart any pointer hijacking attacks with a fewer overhead, around 15% on average.

The Intel MPX [120] extension implements Fat Pointers' memory safety by improving the x86 architecture and its compiler. To support Fat Pointers, Intel MPX provides 7 new instructions and four 128 bits additional bound registers in the x86 architecture. The main purpose of a bound register is to keep both the lower bound and the upper bound of an in-memory pointer. The additional x86 instructions are used to create new bounds in a bound register, compare a pointer with its associated bound register, move a bound register to another one, and finally, spill the bound register in a dedicated table bound table. At the software level, the compiler toolchain implements all the new registers and the additional instructions of the MPX extension. It also modifies the calling convention such that every return address pointer is protected using the MPX extension. Unfortunately, Intel MPX suffers from many limitations [120]. First, it induces a very high execution-time overhead up to 150% with the GCC compiler support. Also, O. Oleksenko et al. founds that Intel MPX is not compatible with some C idioms and 10% of their evaluated programs crashed due to compiler bugs. Finally, Intel MPX exposes very poor security results. Despite Intel MPX is tuned for desktop computers, it does not take into account dangling pointers. Moreover, according to the evaluation performed by O. Oleksenko et al. [120], MPX only protects against 46% of the attacks included in the RIPE benchmark suite [85].

In the same purpose, Shakti-T [125] is a RISC-V [90] processor architecture that enforces data-flow integrity following the Fat Pointer approach. Shakti-T extends the RISC-V architecture by introducing new instructions and specific memory support for Fat Pointers. At execution-time, Shakti-T maintains a Pointer Limit Memory (PLM) to store the base and bound of every pointer. The base address of the PLM is conserved in a ded-

icated Pointer Limits Base Register (PLBR), and every pointer is associated with an ID stored alongside in the regular memory. To speed up the access to the PLM, the Shakti-T architecture implements a Base and Bound Cache (BnBCache). This cache is composed of two elements, a Base and Bound Index (BnBIndex) and a Base and Bound Look Up (BnBLookUp) table. Each index of the BnBLookUp maintains the base, the bound, the ID, and a validity bit of a pointer in the PLM. Moreover, the BnBIndex is an extension of the register file that holds an index for each general-purpose register to an entry in the Base and Bound Cache.

The Fat Pointer hardware support proposed by Shakti-T is driven by specific customized instructions introduced in the RISC-V instruction set. For instance, a special write instruction [125] (wrplm) is provided to populate the PLM memory with the base and bound value of a pointer. Also, special load instructions are provided to load the base and bound values from the PLM to the BnBCache, load a pointer with its ID into a general-purpose register and checks its validity using the BnBCache. In terms of security, no specific evaluation is performed by Shakti-T however, they claim to thwart both spatial and temporal attacks with around no performance overhead.

Fat Pointers takeaways

- Fat Pointers extend regular pointers with metadata such that they keep track of their base and bound.
- Fat Pointers require additional memory space to keep track of the limits of each pointer
- Fat Pointer ensures fine-grained control-flow integrity.

1.3.4.3 Sensitive data isolation

When executing a common C application in memory, it appears that security-critical and non-security critical data are interleaved. For instance, a function's stack holds the return address of the function (control-flow data), some buffers, pointers (code-pointers and data-pointers), some loop control data (non-control data) and also security-critical tokens (security-critical data). Unfortunately, due to this configuration, sensitive data that are spatially close to vulnerable buffers may be tampered by out-of-bound attacks.

One way to tackle this issue is to disjoin the memory space in two parts, one for the non-security critical data and the other for the security-critical data. Figure 1.20, displays this concept of sensitive data isolation.

In Figure 1.20, an additional memory area (safe memory space) is introduced in the memory layout of the application. This safe memory space is disjoined from the other data memory areas. Thus, by placing the security-critical data (pointers/return addresses) in the safe memory region and leaving non-sensitive data in the stack, it is conceptually harder and even impossible for a spatial vulnerability to reach this disjointed area.

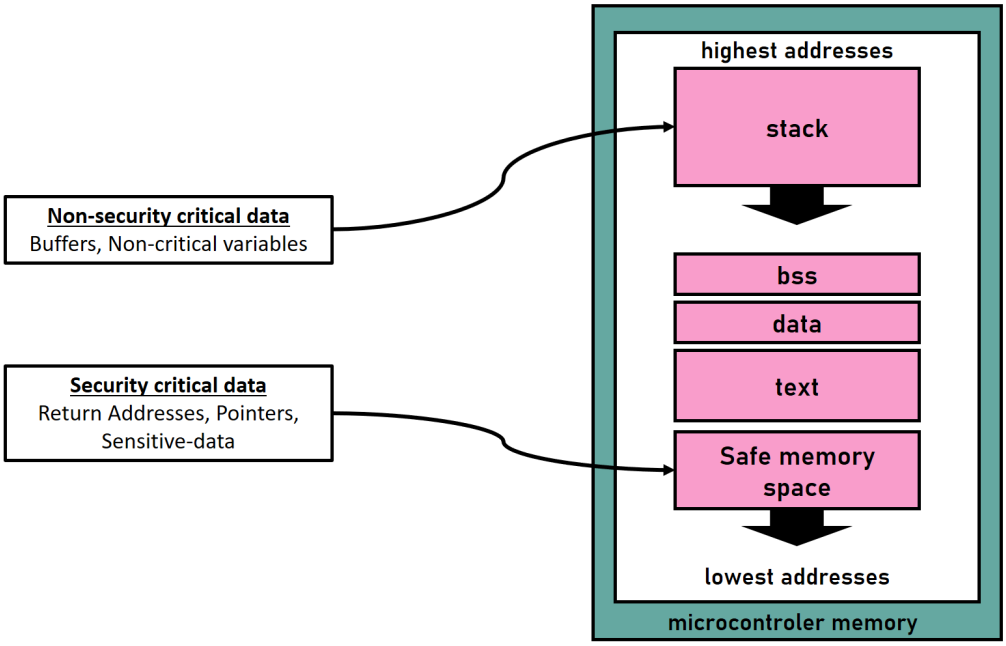

Figure 1.20: Sensitive data memory separation

To avoid any confusion, it is important to point out that the principle of sensitive data isolation in the example is different from Software Fault Isolation (SFI) [126]. SFI technology such as TrustZone [87] maintains different memory areas where sensitive parts of an application can be isolated. The main purpose of SFI is to segregate sensitive parts of application code or isolate some tasks in a multi-tasks application such that vulnerability does not spread through the entire system. While SFI restricts the attack surface of a memory exploit, it does not ensure sensitive data-flow integrity.

Code pointer separation (CPS) [127] introduced by V. Kuznetsov et al., follows the concept of memory space isolation. CPS creates two data memory space, one for sensitive code pointers and the other one for the non-sensitive pointers. To achieve sensitive code-pointers isolation, CPS maintains a disjoined safe stack memory area and a disjoined safe heap memory area. Every sensitive code pointer such as function pointers and return address pointers are always placed in the safe area while the other data/pointers remain in the regular stack. The isolation between the sensitive code pointers and the insensitive is guaranteed at the instruction level by the compiler. Indeed, V. Kuznetsov et al. [127] implemented a plugin in the LLVM compiler that identifies sensitive code pointers and generates the correct instruction flow to store them in the safe area. According to [121], CPS induces a very low execution-time overhead ranging from 8.4% to 10.5%. Unfortunately, CPS only protects code pointers with one level of dereferencement. Consequently, every pointer indirectly calling an indirect function pointer is not considered sensitive by CPS and then not protected by the safe stack. It follows that a memory-based attack can leverage such pointers to execute malicious code. To overcome this limitation V. Kuznetsov et al. proposed code pointer isolation [127] (CPI) an improved implementation of CPS. CPI aims at placing every sensitive code pointer in the safe area as well as every pointer that may dereference a sensitive code pointer. Both CPI and CPS thwart every at-

tack from the RIPE [85] benchmark suite. This is very encouraging since CPI induces less than 11% execution-time overhead. Unlike control-flow integrity, CPI does not perform any sanity checks on code pointers, it just prevents their corruption thanks to isolation. Unfortunately, as clearly mentioned in their implementation [127], CPS and CPI do not mitigate the so-called data-flow oriented programming attack [65]. Indeed, CPI and CPS only protect code pointers and not data pointers. In a recent attack, I. Evans [128] leverage data pointers that are not protected by CPI/CPS to leak the position of the safe memory and perform an arbitrary write on the code-pointers inside.

Data-isolation takeaways

- Sensitive data isolation maintains two different memory space, one for sensitive data and one for regular data. Data isolation is usually achieved with instruction level separation.
- Software-based data-flow isolation induce an acceptable execution-time overhead.

1.3.4.4 Tagged sensitive data

The previous section exposed the mechanisms used to isolate sensitive data. In the example of CPI and CPS [127], sensitive data isolation is achieved at the instruction level. The LLVM [23] compiler plugin of CPI and CPS ensures that any load and store instruction manipulates sensitive data using the safe memory. The principle of tagged memory broadly reuses the same instruction separation approach introduced by CPI.

First, tagged memories provide additional tag information to data objects. At compilation time, every read and write instruction is associated with the tags' object it modifies. At execution-time, the instrumented write instructions mark each written memory location with its associated tag. When an object is accessed using a read instruction, the tag of the object is compared with the tag of the instruction to verify that the object has not been overwritten by a different tag instruction.

Figure 1.21 displays a simple tagged memory with three in-memory adjacent objects and their respective tags. These tags are displayed using colors such as green, yellow, and red. In Figure 1.21, a store instruction is associated to object A and a write instruction is associated with object B. By controlling the offset of the store instruction, an attacker may use it to tamper object B. However, when object B is accessed with the read instruction, the maliciously modified tag of object B does not match the tag of its accessing instruction raising a data-flow violation.

Data Flow Integrity (DFI) [129] extends the Phoenix compiler framework introduced by Microsoft Research to enforce data protection. DFI enforces data integrity in three steps. First, the compiler performs an inter-procedural reaching definition analysis to determine the data-flow graph of a program. During this analysis, the compiler extracts all the data that may be read by an instruction, these instructions are said to use the value. Then, for each data read, the compiler determines the set of all instructions that

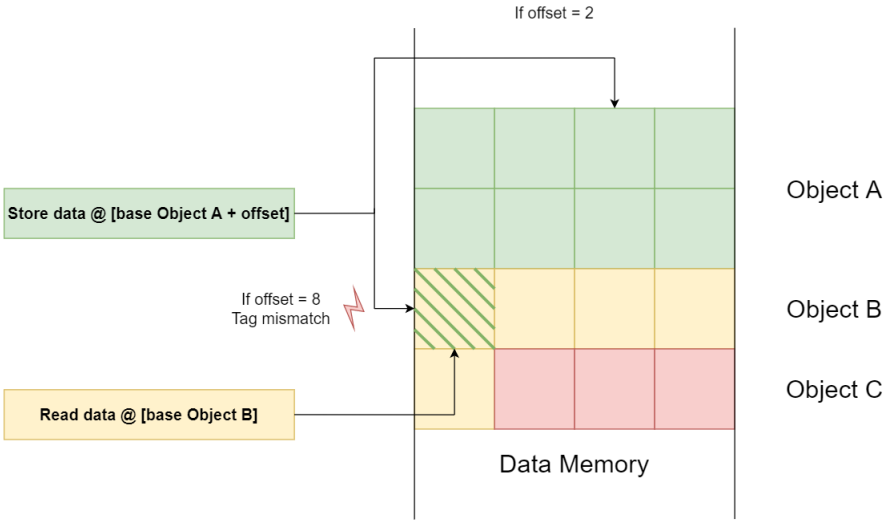

Figure 1.21: Tagged sensitive data

define it, these instructions are said to define the value. As a result, the compiler builds a data-flow graph where write instructions are associated with the value they may define. In a second step, the compiler instrument every read and write instruction to track the data-flow at execution-time. In the third step, at execution-time, the program maintains a Reaching Definition Table (RDT). The RDT records the last instruction that defined a memory location. Then, when this memory location is used by a read instruction; the program consults the RDT to check whether the last instruction that defined the current value is compliant with the precomputed data-flow graph. As a result, DFI is efficient at tracing and detecting data-oriented attacks. Unfortunately, applied to embedded systems it raises challenges. Indeed, DFI induces an execution-time overhead ranging from 43% to 104% [129]. Also, the size of the reaching definition table is around 50% of the instrumented application which may be prohibitive for memory-constrained embedded systems.

Write Integrity Testing (WIT) [130] also introduced by Microsoft Research follows the principle introduced by DFI. At compile-time, WIT uses an inter-procedural analysis to compute the set of memory objects that are defined by write instructions. Then, WIT assigns a color to each memory object and the instruction that can define them. During execution-time, WIT ensures that each write instruction defines a memory object that has the same color. The program also maintains a color table that is updated when objects are allocated and de-allocated. Unlike DFI [129] that checks the last instruction that defined sensitive data, WIT prevents memory exploits from defining sensitive data. Besides, WIT only instruments the unsafe write instructions. Thus, the color table maintained but WIT is smaller than the RDT of DFI with an extra size overhead of 12,5%. The average execution-time overhead induced by WIT is around 10% with a maximum of 25% on the SPEC CPU2000 [74]. In terms of security, both DFI and WIT are challenged with real-world data-flow and control-flow attacks on the NullHttpd HTTP server. It seems that they both detect any real exploits.

Hardware-Assisted Data Flow Isolation (HDFI) [131] and LowRISC [132] tagged mem-

ory are hardware approaches to Data Flow Integrity (DFI) introduced by Microsoft Research. For instance, HDFI uses two tag levels (IL1 and IL0) to separate sensitive data (IL1) from regular data (IL0). To trace the data-flow, HDFI extends the open-source RISC-V processor memory with an additional one-bit tag field for every word. Furthermore, to enforce sensitive data isolation, HDFI introduces a custom store and a custom load instruction in the RISC-V Instruction Set Assembly (ISA). To support these instructions, the processor logic is modified such that; every regular write instruction defines a data and set the tag field to zero, every special write instruction defines a sensitive data and set the tag field bit to one and every special load instruction checks that the used sensitive data has a tag set to one. Thus, a regular write that modifies sensitive data also changes its tag. When this sensitive data is re-used by a custom load instruction, the processor detects a wrong tag and raises an exception. Following the HDFI concept, it appears that each data requires two memory accesses, one for the tag and one for the value. Therefore, to not undermine the memory access performance, HDFI maintains an additional L1 and L2 cache for the tag. Finally, HDFI comes with both an extended GCC and LLVM compilers which ensure the instruction level isolation between sensitive data and regular data. In terms of security, HDFI detects any attack from the RIPE benchmark suite with an average overhead of 2% on the SPEC CPU CINT 2000 [131].

Tagged sensitive data takeaways

- Tagged memory are used to track every write instruction in a program and check if it corresponds to a static data-flow graph.
- Pure software data-flow integrity is expensive in both execution-time and memory size.
- Hardware assisted tagged memory are highly effective in security, inducing a very low execution-time overhead.

1.4 State of the Art Synthesis

This section summarizes the defenses previously discussed. It discusses the strengths and weaknesses of control-flow integrity, heuristic countermeasures, software diversity, and data-flow integrity. The main goal of this synthesis is to prepare the following Chapter in which the thesis highlights why critical medical devices face security issues and why the approaches proposed in the related state-of-the-art are not the most appropriated for them. In a nutshell, this synthesis contributes to exposing the gap between the existing security defenses and the security required by the life-critical systems.

1.4.1 Control-flow integrity discussion

Regarding control-flow integrity, Table 1.1 displays a summary of various control-flow integrity concepts approached in the state-of-the-art. These concepts are summarized regarding the **protection** they offer such as backward-edge (BW), and forward-edge (FW). Table 1.1 considers a security policy as fine-grained (FG) if a protected branch always targets the expected destination with 100% accuracy. Otherwise, Table 1.1 considers the policy coarse-grained (CG). Besides, Table 1.1 considers whether the implementation of the protection is either achieved at the software level, the hardware level, or both.

Table 1.1: Control-flow integrity policies

Name	Type	Protection		Implem.		Modularity	Deterministic	Costs	
		BW	FW	Soft	Hard			Exec	Size
Abadi [68]	Labels	CG	CG	✓	✗	✗	✓	+16%	+8%
Abadi [68]	Labels+SCS	FG	CG	✓	✗	✗	✓	+21%	N/A
CET [75]	SCS+Labels	FG	CG	✓	✓	✓	✓	N/A	N/A
HCFI [78]	SCS+Labels	FG	CG	P	✓	✗	✓	<1%	N/A
SmashGuard [81]	SCS	FG	✗	✓	✓	✓	✓	<5%	N/A
HAFIX [82]	Labels	FG	✗	✓	✓	✗	✓	<2%	N/A
HAFIX+ [76]	Labels	FG	✗	✓	✓	✗	✓	+1.75%	+13.5%
BB-CFI [83]	Branch Monitoring	FG	CG	✓	✓	✗	✓	<1%	<209kB
BBB-CFI [84]	Branch Monitoring	FG	CG	✓	✓	✗	✓	0.13%	13%
C-FLAT [86]	Branch Monitoring	FG	CG	✗	✗	✓	✓	<72%	N/A
Lo-FAT [89]	Branch Monitoring	FG	CG	✗	✓	✓	✓	2%	N/A
bin-CFI [69]	Trampoline	CG	CG	✓	✗	✗	✓	+4.29%	+139%
MoCFI [77]	Trampoline	FG	CG	✓	✗	✓	✓	<10%	N/A
LLVM-CFI [92]	Trampoline	✗	FG	✓	✗	✓	✓	<6%	N/A

CG: Coarse-grained, FG: Fine-grained, N/A: Not Available

The other elements in Table 1.1 refer to the assets of the C programming language exposed in section 1.1.1. Indeed, Table 1.1 evaluates whether the various control-flow

integrity protections are **modular**, **deterministic**, **efficient** with both a low **execution-time** and **size** overhead.

More specifically, Table1.1 considers protection to be modular when they are compatible with incremental compilation. It also considers protection to be deterministic when the extra cost induced by the security is always the same given a program and an initial state. Finally, the execution-time overhead and size overhead represents the costs of the security. Unfortunately, few countermeasures provide a consumption analysis, this is why Table 1.1 does not include these specific results.

Regarding the security, control-flow integrity is globally efficient at mitigating both code injection and code-reuse attacks. Unfortunately, each approach suffers from minor limitations that still leave opportunities for an attacker to execute a successful attack. For instance, coarse-grained policies cannot guarantee that indirect branches target valid destination sites with 100% accuracy. As a result, attackers can still perform attacks that comply with enforced coarse-grained control-flow protection [72]. Some of these attacks are practical, they leverage call-preceded gadgets that do not break the control-flow policy to execute malicious code.

Regarding the costs, it seems that pure software-based control-flow integrity protections induces higher execution-time overhead than hardware-software codesign. For instance, [68], and [86] induce a non-negligible execution-time overhead (21% and up to 72% respectively, see Table 1.1). Besides, purely software protections drastically increase the size of applications [69], which could be prohibitive.

According to Table 1.1, control-flow integrity is deterministic, these solutions rely on labels, shadow call stacks, trampolines, and branch restrictions that do not introduce randomness in protected systems. In terms of modularity, not all existing concepts in Table 1.1 are compatible with incremental compilation and external non-protected libraries. For instance, most control-flow integrity prototypes based on labels rely upon post-compilation binary instrumentation. A lack of modularity is often difficult to integrate within the regular software engineering processes.

In closing, hardware-software codesigns present the best tradeoff between security and performances. For instance, labels or branch-restriction policies coupled with hardware shadow call stacks induce both a very low execution-time and size overhead. Unfortunately, although shadow call stacks provide fine-grained backward-edge control-flow integrity, their integration within real systems is challenging. First, they require dedicated handlers to manage goto issues such as the **setjmp** and **"longjmp"**. Second, hardware shadow stacks are limited by the size of the buffer, a large call-depth may induce overflow managements. Finally, for multi-tasks systems, hardware shadow stacks should be assisted by operating systems and specifically schedulers to maintain stack coherency.

1.4.2 Heuristic defenses discussion

Table 2 summarizes the heuristic defenses exposed in the state of the art. Globally, heuristic defenses all work on the same principle such as recognizing attack signatures. These defenses use various heuristic metrics to determine the risk of a system being under

attack. In comparison with Table 1.1, Table 1.2 protection's column summarizes the number of heuristic metrics used by the heuristic engine to detect attack signatures. The other columns remain the same as Table 1.2.

Table 1.2: Heuristic defenses policies

Name	Enforced metrics	Implem.		Transparent	Modularity	Deterministic	Costs	
		Soft	Hard				Exec	Size
kBouncer [97]	1	✓	✗	✗	✓	✓	+6%	N.A
ROPGuard [99]	6	✓	✗	✗	✓	✓	+0.5%	N.A
ROPecker [100]	3	✓	✗	✓	✓	✓	+2.6%	N.A
SCRAP [101]	1	✗	✓	✓	✓	✓	<2%	N.A

One asset of heuristics is that they often require a few binary changes to be effective. Heuristics are transparent to protected programs and thus, fully modular and compliant with most of the C standards exposed in section 1.1.2. Besides, Table 2 reveals that heuristic engines induce low execution-time overheads. For purely software protection this low overhead can be explained by the fact that [97, 99, 100] often trigger the heuristic engine in specific cases such as dangerous function calls. For hardware-based protections the heuristic engine is fully integrated within processors' architecture, operating in parallel with the monitored program. This implies a very low execution-time overhead [101].

Regarding security, heuristic protections do not enforce any specific control-flow integrity policy. An attacker can still violate the application's control-flow graph or attempt data-oriented attacks. Also, like anti-virus, heuristic protections are highly vulnerable to memory-based attacks combined with evasion techniques. With knowledge of the heuristic detection algorithm, an attacker can craft an attack that mimics a normal behavior for the heuristic engine. For instance, the most heuristic analysis aims at detecting code-reuse attacks based on the number of branches executed by a program in a short time window and the number of instructions between these branches. To thwart heuristic defenses, researchers demonstrated that introducing intermediate long gadgets in a Return-Oriented Programming chain is efficient at tricking heuristic engines [101]. Also, as revealed in section 1.3.1.4, most heuristics offer incomplete protection. For instance, [97, 99, 100] do not prevent Jump-Oriented Programming attacks. To push it further, recent research demonstrates that signal-return oriented programming attacks [133] can launch a malicious code execution with a single gadget. To date, such an attack defeats any heuristic engine.

1.4.3 Software diversity discussion

Table 1.3 exposes various software diversity techniques explored in the state-of-the-art. Software diversity aims to create several morphs of a program to harden stable exploit development. Table 1.3 considers a software diversification coarse-grained when the protection relies on random-offsets introduced in the program code. Conversely, fine-grained software diversification implies complete randomization of code from a morph to another while preserving the semantic of the program. For instance, with coarse-grained software

diversification, the gadgets within a program remain the same from one morph to another, however, their in-memory position is shifted by random offset. In contrast, gadgets do not survive fine-grained software diversification. Table 1.3 also displays various diversification techniques.

Table 1.3: Software diversity policies

Name	Protection	Diversification technique	Implem.		Modularity	Deterministic	Costs	
			Soft	Hard			Exec	Size
ASLR	CG	random offset	✓	✗	✓	✗	N/A	0%
Full-ASLR (PIE)	CG	random offset	✓	✗	✓	✗	14% [108]	N/A
[105]	CG	random NOPs	✓	✗	✓	✗	<40%	<80%
ASLP [108]	FG	sections, functions, BB, data	✓	✗	✓	✗	<1%	<80%
ILR [109]	FG	instructions	✓	✗	✓	✗	13%-16%	14MB-264MB
STIR [109]	FG	BBs, instructions	✓	✗	✓	✗	1.6%	73%
SOFIA [112]	FG	cipher	✓	✓	✗	✓	106%	N/A

One can observe that software diversity is not deterministic. This is exactly the heart of the protection offered by software diversity. The general idea is to generate a random binary which in the presence of a vulnerability prevents exploit development based on Return-Oriented Programming techniques. While this type of protection raises the bar for exploit developers it remains incompatible with safety-critical embedded software guidelines [31].

Also, regarding security, software diversity becomes vulnerable when an in-memory vulnerability is coupled with a memory disclosure. In this case, an attacker leverages memory disclosure to de-randomize the application code. After de-randomization, the attacker disassembles the leaked memory and craft a Just-In-Time Return-Oriented Programming attack [106].

Regarding the costs, software diversification induces extensive execution-time overhead depending on the implementation. According to Table 1.3 and section 1.3.2, the execution-time ranges from less than 1% overhead, up to more than the double of the original application. Finally, dynamic software diversification seems to be difficult to apply to low cost embedded systems. First, fine-grained randomization often requires the support of a powerful operating system or a virtual machine with an intermediate language [109] that rewrites the binary on-the-fly. Second, according to Table 1.3, software diversification can induce prohibitive size overhead.

1.4.4 Data-flow integrity discussion

Table 1.4 summarizes data-flow integrity defenses explored in the state-of-the-art. To remind, data-flow integrity prevents both control and non-control data corruption. Table 1.4 displays wether the studied protection defenses protect Code Pointer (CP), Data Pointer (DP), and/or regular Data (D). Other assessments in Table 1.4 such as **Implementation**, **Modularity**, **Determinism**, remain the same as in Table 1.1.

According to Table 1.4, it seems that common data-flow integrity protections protect

Table 1.4: Data-flow integrity policies

Name	Type	Protection			Implem.		Modularity	Deterministic	Costs	
		CP	DP	D	Soft	Hard			Exec	Size
CCFI [113]	Crypto	✓	✗	✗	✓	✗	✓	✓	23%	N/A
PointGuard [114]	Crypto	✓	✓	✗	✓	✗	✓	✓	0-20%	N/A
DSR [115]	Crypto	✓	✓	✓	✓	✗	✗	✓	15-28%	N/A
PAC [118]	Crypto	✓	✓	✗	✓	✓	✓	✓	0.5%	N/A
Softbound [123]	Fat Pointers	✓	✓	✗	✓	✗	✓	✓	<79%	N/A
Hardbound [124]	Fat Pointers	✓	✓	✗	✓	✓	✓	✓	<15%	N/A
Intel MPX [120]	Fat Pointers	✓	✓	✗	✓	✓	✓	✓	<150%	<0.5%
Shakti-T [125]	Fat Pointers	✓	✓	✗	✓	✓	✓	✓	N/A	N/A
CPI/CPS [127]	Pointer Isolation	✓	✗	✗	✓	✗	✓	✓	10%	N/A
DFI [129]	Tagged memory	✓	✓	✓	✓	✗	✗	✓	43-104%	50%
WIT [130]	Tagged memory	✓	✓	✓	✓	✗	✗	✓	10-25%	12.5%
HDFI [131]	Tagged memory	✓	✗	✗	✓	✓	✓	✓	2%	<1%

CP: Code Pointers, DP: Data Pointer, D: Data, N/A: Not Available

code pointers thus ensuring fine-grained control-flow integrity. In table 1.4 there are also quite a few solutions that propose data pointers and even regular data protection [115, 129]. While protecting code pointers enforces fine-grained control-flow integrity, data pointers protection and regular data protection highly complicate data-oriented attacks. Unfortunately, directly from Table 1.4, it appears that the finer the security protection, the higher the execution-time overhead [129]. This is, even more the case for pure software implementations. Indeed, data-flow integrity requires to trace and check every read and write operation performed by a program at run-time. While this ensures high granularity protection, the process is cumbersome inducing high execution-time and size overhead. Like control-flow integrity, hardware support for data-flow integrity drops the execution-time overhead and the costs of the security.

As control-flow integrity, data-flow integrity does not rely on randomness. Every data-flow integrity protection reviewed in Table 1.4 can be considered deterministic. However, existing data-flow integrity protection is not always very modular, sometimes they require programs to be analyzed and instrumented as a whole to track each read and write operation [129]. As previously mentioned this is convenient with incremental compilation approaches. Another drawback of most data-flow protection such as [113,130] is the compatibility with non-instrumented code. Indeed, instrumented code badly interfaces with non-instrumented code, preventing any continuity in the tracing of the data-flow.

1.4.5 State-of-the-art discussion

In closing, this Chapter highlights a state-of-the-art of memory-safety defenses. In the presented state-of-the-art, the existing works are grouped according to four main categories such as; control-flow integrity, heuristics, software diversity, and data-flow integrity. All

these protections raises the bar for attackers. It should be mentioned that the four presented categories include many more contributions than those presented in this manuscript. Researchers try to solve the memory safety issue for more three decades by proposing the most innovative solutions. For instance, other areas of memory safety based safe-dialects such as Cyclone [134], and CCured [135] are not covered in this thesis's state-of-the-art. Also, the field of control-flow integrity and data-flow integrity is not restricted to software security. It also covers many areas such as hardware security with other threat models.

While this thesis gives a global overview of the current existing security concepts, research progresses. Over the past few years, it has become apparent that increasingly more defense leverage open-hardware architectures [90]. According to the Tables 1.1, 1.2, 1.3, and 1.4, it is through a close combination of hardware and software that the most robust and optimal protection are achieved. These results provide directions for the approach of this thesis concerning the design of efficient memory safety defenses.

Regarding security, this state-of-the-art synthesis highlights that data-flow integrity may be the most accurate defense. Indeed, data-flow integrity is accurate enough to prevent multiple attacks such as code-reuse attacks and data-oriented attacks protecting sensitive data at various granularities. However, this synthesis also revealed that fine-grained protection comes at a certain price such as time and size overhead. In front of that, critical systems have high-performance expectations, and to-date lack memory safety protections. To determine the finest defense for life-critical systems, the next Chapter study the root cause of memory safety issues in the context of medical devices, their security requirements, why the state-of-the-art can be improved, and the thesis's approach.

2 Approach

Summary of the Chapter

This Chapter follows the state-of-the-art. Although the previously exposed memory-safety defenses are effective, it seems that none of them are really integrated into life-critical medical devices due to several conceptual and constraint issues. As a first step, this Chapter exposes the gap and states the thesis problematic. From the problem exposure, this Chapter raises several criteria that should be taken into account when designing practical defenses for life-critical medical devices. Finally, following such specifications, this Chapter exposes the thesis approach to improve software security embedded systems.

Contents

2.1 Problem statement

According to the state-of-the-art, various defenses against memory-based exploits are available and efficient. Some of these defenses have been increasingly deployed in desktops and operating systems in recent years. Even if they are imperfect, they still increase system security. Unfortunately, it would appear that medical devices do not take advantage of existing defenses and remain vulnerable to simple attacks. For instance, [18] found multiple code injection attacks allowing remote code execution on commercial infusion pumps. These code injection attacks revealed that the system didn't possess even the most basic memory safety mitigations highlighted in the state-of-the-art. Also, the study conducted by [18] reveals that these pumps were not designed following a threat model and secure coding practices. So there is an inconsistency. On the one hand, security is an important concern for critical medical devices. These devices are in direct interaction with the human body and may cause irreversible health damages in case of security breaches. Knowing that the state-of-the-art of existing defenses provides various solutions, manufacturers should have implemented at least one of them in critical devices. Sadly, security studies of current systems reveal that critical medical devices are vulnerable to attacks that have been treated over 10 years in the past. Being aware of this inconsistency, this section investigates the security issue faced by medical devices and dissects it in several points, starting from explaining why wireless devices are still insecure and why existing defenses are not yet implemented.

2.1.1 Why critical medical devices are insecure?

The Internet of Things (IoT) is a competitive and fast-moving market. Manufacturers are rushing their products to get market leadership. Besides, there is no specific security standard for medical devices [136]. This lack of regulation encourages manufacturers to prioritize the functionalities of their products instead of spending time on security aspects. Security tests and advanced vulnerability research are also time-consuming, inconsistent with the dynamic of the IoT market. Unfortunately, many critical devices are written in unsafe languages such as C. The design of these devices involves complex problems from hardware to software such as complexity, third-party services (sometimes outdated), real-time constraints, number of lines of code, wireless connectivity. Once again, covering all these aspects with security tests is time-consuming and difficult. As a result, it appears that the integration of security is poorly compatible with time-to-market and lack of regulation.

Many existing medical devices on the market are to date very safe and approved by the FDA [4]. This emphasizes that medical device manufacturers are experts in safety and able to release high-quality products. Unfortunately, safety focuses on unintentional actions that may be achieved by a device and overlooks the risk of intentional security-related malicious actions. Most of the cybersecurity issues like remote code execution happen when a critical device is connected to the network. Connecting a medical device to the network is a relatively recent trend and cannot be blamed knowing the IoT market.

Unfortunately, wireless features expose critical devices to new attacks that manufacturers were not confronted with in the past. Are manufacturers disarmed when integrating security in medical devices? In any case, both [17] and [18] reveal a lack of strong defenses in medical devices. Probably, connectivity requires advanced security experts to be properly integrated. Maybe companies lack security engineers [10] and thus, security issues are left to non-security experts. One thing is certain, integrating security into a critical system cannot be improvised, leaving non-expert achieving such task often results in useless defenses that can't fend off an advanced attacker. The lack of security in medical devices can be explained by a lack of **security expertise** and especially by an increased attack surface due to **connectivity**.

Openness to research is also a means of improving systems' security. For instance, increasing big tech companies such as Intel, Google, and Microsoft offer bug bounty programs [137] allowing researchers to look for vulnerabilities in their products within a legal framework. Thus researchers may attack real systems and contribute to improving their security while being rewarded. Currently, there is no medical devices manufacturer that offer bug bounty platforms. Thus, many medical devices come with commercial constraints that prevent researchers to freely attack them and disclose vulnerabilities. Open-source systems also contribute to the improvement of closed-source systems. Take the case of Windows and Linux. By showing attacks and innovative defenses under Linux, researchers have contributed to the advancement of knowledge and concepts that have allowed Windows to improve its security. This does not mean that an open-source system is more secure than a closed-source system, or that all closed-source systems have to be open-source to be more secure. What is important is to have open-source systems close enough to closed-source systems that allow researchers to propose new security concepts applicable to closed-source systems. Unfortunately, medical device security is not open to research. Few open-source life-critical devices are available to researchers discouraging them contribute. Thus, the lack of **openness to research** and the lack of **open-models** remains an impediment to the improvement of critical medical systems security. One way to change the situation would be to provide researchers with open-source systems close to real systems.

2.1.2 Why current defenses are not implemented in medical devices?

Interestingly, regarding memory exploits, the state-of-art proposes efficient protections. However, when looking at the real attacks on medical devices, none of these protections seems to be deployed [15, 17, 18, 37, 138]. One can easily note that there is a gap, why no protections are integrated into these systems?

By reviewing state-of-the-art defenses, several points reveal that the existing defenses are not suitable for medical devices. First, many concepts offer an incomplete defense. For instance, most of the control-flow integrity solutions provide either backward-edge or forward-edge protection. So an attacker can always attack forward-edges when backward-edges are protected and vice versa. As medical devices may be life-critical, they require robust and complete protection. Second, most of the fine-grained approaches induce a

relatively high execution-time overhead. As a result, to reduce the execution-time cost, most fine-grained defenses are relaxed in coarse-grained defenses, leaving many more opportunities for attack. The major issue with memory safety is the trade-off between the execution-time cost and the accuracy of the protection. Unfortunately, wireless connected medical devices require both performance and accuracy. At the same time, these devices should meet very strict **real-time constraints** with the highest security protection. To remind, manufacturers use C because of its efficiency. Thus, they won't break this asset for incomplete security.

Most control-flow integrity approaches require application code to be instrumented afterward. This is not very **modular** and even **impractical** in the sense that developers can only perform security tests at the end of a system's development phase. While many implementations in the state-of-the-art provide the necessary **toolchains** to perform code instrumentation, control-flow integrity often breaks the modularity asset provided by the C language. Also, backward-edges control-flow integrity provided by shadow stacks faces compatibility with the **C standards** (see *"setjmp/longjmp"*).

Heuristic methods seem to be a good compromise for critical systems in comparison with control-flow integrity. Especially hardware-assisted heuristic protections, they induce a low execution-time overhead and do not require application code to be instrumented. Thus heuristic methods maintain the **modularity** and **efficiency** of the C programming language. Unfortunately, heuristic methods are incomplete security protection, they aim at detecting specific memory-based attack signatures such as code-reuse attacks. However, as exposed in the state-of-the-art, attackers use hybrid exploits to take over a system. As a result, attackers leverage the fact that heuristic defenses do not enforce a specific control-flow/data-flow integrity protection to evade the heuristic engine. Another issue with heuristic protections is time-to-check and false positives. Most heuristic engines have a certain detection delay between the moment an attack is performed and the moment it is detected. This is an issue for critical systems, an attacker may have changed critical data hard to restore afterward. Finally, the weakness of heuristic methods is the lack of **accuracy** and therefore the lack of **robustness** against advanced attacks.

The criticality of wireless medical devices prevents the implementation of certain types of defense. For instance, software diversity re-randomizes applications code at each execution by altering its instructions and structure (order of functions, basic blocks, and instructions). This practice makes the code non-deterministic and thus prevents an attacker from forging functional exploits in case of vulnerability. Unfortunately, **non-deterministic** code is strongly discouraged in safety-critical **coding standards** [31, 32]. Safety-critical software manufacturers use C for **determinism** that is broken by software diversity. Another problem with software diversity is the resources needed to implement it. In the state-of-the-art, most software diversity solutions involve a powerful virtual machine or a modified operating system. These supports require huge memory space and consume a significant amount of power. Unfortunately, critical medical devices are tiny and should consume as little as possible. Often the embedded operating systems are **real-time** and **lightweight**. They cannot implement virtual machines. Moreover, the execution-time

overhead of some software diversity protections is very high, up to 100%. This breaks the **efficiency** property of the C programming language. As a result, although software diversity complicates the development of successful exploits, it does not satisfy most of the safety coding guidelines required by critical medical devices. Therefore, software diversity methods are not the most appropriate defense for medical critical-devices.

Data-flow integrity differs from the other approaches because it mitigates the memory-safety issue at the root cause: the initial critical-data corruption. This approach prevents an attacker from modifying sensitive data thanks to the data-flow graph. In comparison with control-flow integrity, data-flow integrity is finer because it does not rely on control-flow graphs. Data-flow integrity could protect either control-data and non-control data. Unfortunately, according to the state-of-the-art, pure software implementations of data-flow integrity induce a very high **execution-time overhead** (see Chapter 1, section 1.4). Also, some implementation of data-flow integrity such as DFI [129] are incompatible with third-party libraries, that again break the **compatibility** asset of C. Regarding hardware-software defenses, PAC [118] can achieve fine-grained control-flow integrity through control-data integrity with less than 2% execution-time overhead. However, for critical non-control data, the execution-time overhead rises to 40%. That always brings up the tradeoff between **accurate security** and acceptable **performance overhead**.

In terms of practicality, the identification of sensitive non-control data is hard to perform automatically, most of the compiler and instrumentation toolchains provide code-pointer protection and leave the implementation of sensitive code-data protection to the user. Knowing the lack of **security expertise** in the industry [10], it is impractical to let developers instrument their code to protect sensitive non-control data. Thus the lack of proper software support does not ease the implementation of security in critical devices. For the moment, it seems that software data-flow integrity is not implemented in medical devices [18]. Software data-flow integrity induces too much high execution-time overhead breaking the **efficiency** asset of the C programming language. Concerning hardware-software approaches, there is no dedicated implementation for critical embedded systems. Most implementations such as PAC [118] are designed for personal desktop or mobile phone computers. This availability issue is also a reason why data-flow integrity is not currently used. Finally, even if data-flow integrity protections are promising in terms of security, it needs more research to better understand the assets of the C programming language.

Following this discussion, one can understand why critical medical systems do not integrate memory safety defenses. Most existing concepts exposed in the state-of-the-art are primarily intended for personal desktops and clouds than critical embedded systems. To support this argument, it's enough to recognize a few concepts are challenged on a real life-critical system. Unfortunately, life-critical systems behave differently from personal computers. A security countermeasure can stop a traditional cloud application from executing malicious actions. Conversely, on critical systems, safety depends on security, and security issues are impacting safety. For a life-critical system, detecting security attacks is the first step, the second is defining how the system should react and recover without

harming the user.

Most existing memory-based protection degrades the assets of the C programming language. Many protections degrade **the portability**, **the modularity**, **the efficiency**, **the toolchain support**, **the determinism**, and **the standard of the C programming language** in favor of security. Security researchers often forget that **performance** and **portability** are the main barriers [40] for a countermeasure to be accepted. On top of that, many papers claim that they solved the memory safety issue until a new attack appeared [40]. As a result, memory safety protections are hardly accepted by manufacturers. Besides, critical medical devices are not only facing memory-safety issues. Cryptography, authentication, privacy, integrity, defense-in-depth, and safety should be considered and implemented along memory-safety. Hence, to be accepted along with other security countermeasures, memory protections must have as few constraints as possible.

2.2 Important memory safety criteria for medical devices

From the problem statement (section 2.1), this section highlights several criteria that security countermeasures should take into account to run in consistency with critical devices. This thesis considers that proper memory safety protection for critical systems should adhere to the following criteria:

- *Robustness*: A memory exploit can take full control of a program by diverting its control-flow. Regarding critical devices, such attacks endanger the life of its users and may even kill. Such a disastrous scenario is unacceptable. The critical device security should be at the highest level. Unfortunately, memory exploits are powerful. They leverage both control and non-control data at the lowest granularity to make an application execute malicious action. To restrict an attacker's capabilities at the lowest, critical-devices must enforce at least fine-grained protections. A vigorous defense is not inviolable, but the higher the granularity of the protection, the more an attacker's abilities are decreased.

- *Performance*: The performance penalty of a countermeasure is one of the first criteria to be considered before adopting a defense. Most countermeasures in the state-of-the-art undermine performances. The performance penalty is generally measured by the execution speed decrease of an application. As manufacturers want to optimize their products at their best, a high execution-time overhead may be prohibitive for real-time constrained medical systems. Even if 10% overhead seems acceptable, manufacturers may not pay more 5% overhead for a security solution [40].

- *Space overhead*: To be integrated into embedded systems, memory safety countermeasures should not induce unwanted memory overhead. Medical systems, especially in-body critical devices, are restricted in size. Manufacturers tend to reduce the space taken by an application to its lowest. As a result, protection carrying a large amount of metadata that increases the size of a binary over 100% is impractical. Currently, there is no official number determining the memory overhead threshold accepted by manufacturers. Finally, memory safety countermeasures are going to be implemented along with other hardware-software protections. It further emphasizes that low size overhead is very important to leave space for other protections.

- *Modularity*: This criterion applies to software countermeasures based on both compilation or binary rewriting. As the C programming language allows developers to break complex programs into several sub-modules, their hardening must be achieved independently. Non-modular countermeasures are impractical in large scale projects were several developer teams are working on different parts of a system. Only modular countermeasures can be integrated into a practical software development lifecycle.

- *Compatibility*: Most countermeasures in the state-of-the-art are not compatible with all the C standards. For instance, the shadow stacks have issues with the classical

"setjmp/longjmp" functions. Even the use of *"setjmp/longjmp"* is prohibited in critical systems, many countermeasures are still not compatible with external non-instrumented libraries. While these libraries can be vulnerable to memory corruption, they allow incremental deployment. Finally, complex binary instrumentation protections are sometimes incompatible with verification and static analysis tools. The use of the latter is common for critical systems to validate safety.

- *Determinism*: Critical embedded systems must be deterministic. Safety-critical standards prohibit any code practices that lead to non-deterministic behavior [31]. As a result, a safety-critical compatible countermeasure cannot use software diversity or polymorphic code. Finally, deterministic protection can be formally verified. Unlike probabilistic protections, deterministic countermeasures can thwart certain types of attacks with over 100% success.

- *Practicality*: The ease of integration of a security countermeasure is important. As mentioned in the previous section, manufacturers do not always have detailed expertise in security. Thus, to be properly integrated, a security countermeasure must require minimum human interaction. Any security solution requiring a developer to manually instrument code is impractical and error-prone. Therefore, an ideal memory safety protection for critical systems must be easy to use and transparent to the user.

- *Safety*: Once life-critical systems have been developed by manufacturers, they are certified with safety standards. From the moment a system is certified it is assumed to be safe to use and immutable. Unfortunately, this assumption is inconsistent with security. The security of a system constantly evolves. Countermeasures introduced yesterday are no longer effective against tomorrow's attacks. Besides, combining security and safety is not an easy matter. The security of a critical system directly impacts its safety. Thus, ideal memory safety countermeasures must be compatible with safety. For instance, the detection of a security issue related to memory corruption must be highly configurable to switch to recovery routines that keep the critical device in a safe mode.

To summarize, integrating practical memory safety protection in life-critical medical systems is challenging. To be accepted, this thesis considers that security countermeasures should closely follow the previously exposed criteria such as **Robustness**, **Performance**, **Space overhead**, **Modularity**, **Compatibility**, **Determinism**, **Practicality**, and **Safety** to better fit life-critical software security. With an awareness of the criteria, the following section unfolds the thesis's approach to improve embedded device security.

2.3 Approaches

This thesis aims to propose practical memory safety defenses adapted to life-critical medical systems. For this purpose, this section proposes consistent approaches with the problem statement in section 2.1 and the exposed criteria in section 2.2.

However, as this thesis unfolds in the life-critical system field, it is relevant to work with a coherent system workbench. Such a system aims to model the behavior of a life-critical device and serves as a security system workbench to evaluate the efficiency of security countermeasures. Unfortunately, as previously stated, with the lack of openness to research in the medical area, the existence of such a platform is less. Referring to section 2.1.1, the lack of openness to research does not encourage security improvements in medical devices. Indeed, there are almost no open-source medical devices, and it is therefore almost impossible to design dedicated defenses and assess their effectiveness. To fill this gap, this thesis proposes an open security-oriented system workbench platform that models the behavior of a life-critical device. The main goal of such a platform is to further allow the design and assessments of security countermeasures dedicated to safety-critical systems. More specifically, the system workbench is a wireless insulin pump. Such a system is considered as a life-critical medical system and can be simply modeled on off-the-shelf microcontrollers or FPGA. The details and the novelty of the system workbench are highlighted in Chapter 3. The platform is inspired by a survey on existing open platforms and existing insulin pump models.

Back to memory safety, the thesis's approach aims at following the criteria exposed in section 2.2. According to the state-of-the-art synthesis in section 1.4 and to achieve the best balance between **robustness**, **overheads** (time and size), **compatibility**, **practicality**, **determinism**, and **safety**, this thesis enforces that ideal memory safety protections involve both hardware and software. Hardware means that countermeasures require to either change a processor architecture, interface with a processor or add additional external independent hardware modules for security support. Software means that countermeasures require a dedicated toolchain such as compiler, libraries, binary instrumentation tools, and/or finally, real-time operating systems or bare-metal support.

To address software security, this work proposes two approaches. The first one assumes that both the hardware layer (processor) and all the software layers (real-time operating system, compiler, standard C libraries) can be modified to enforce dedicated security support. The second approach considers that most of the chips used in medical devices are fixed based on closed source processors. This unfortunately prevents any hardware architecture modifications. Therefore, the implementation of memory safety defenses is limited to the software layer, i.e. the real-time operating system, libraries, and the entire code generation toolchain. It should be mentioned that even in some cases, neither the compiler used nor the real-time system is accessible.

To fill the criteria exposed in section 2.2, both approaches are top-down and globally divided into two phases. A first phase (design-time) in which critical application developers access an inherently secure toolchain. This toolchain hides the security complexity to

developers allowing application code to be generated with security assets adapted to the hardware. The second phase (run-time) is the execution of the generated critical application that benefits from support provided by both hardware and software. The main goal of both top-down approaches is to simplify the integration of memory safety defenses to non-security experts and ensuring robust run-time protection.

Figure 2.1 displays an overview of both generative approaches.

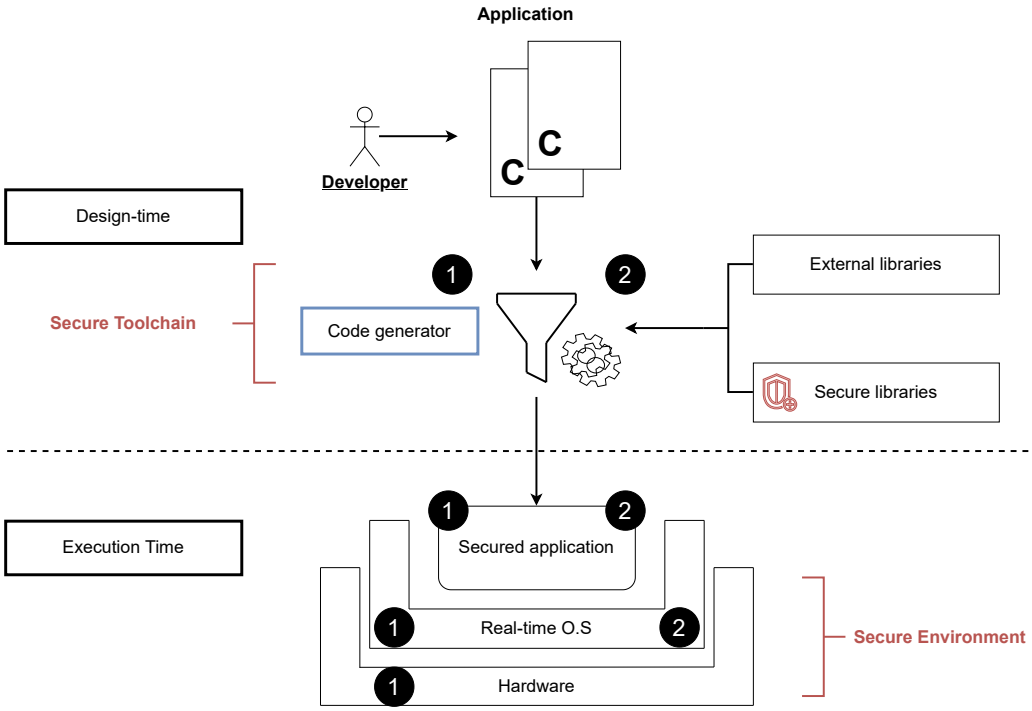

Figure 2.1: Generative approaches

Referring to Figure 2.1, the first approach (1 in 2.1) provides security support from hardware to software. To ensure the finest tradeoff between robustness and performances, the run-time protection engine is rooted in hardware. As revealed in the state-of-the-art, data-flow integrity seems to be the finest security approach to tackle the memory safety issue. Thus, the run-time protection engine tracks sensitive data such as function pointers, return addresses, induction variables, and security-critical data to detect their violation. When a violation happens, the run-time protection detects it, logs it, and maintains the system in a safe state. Of course, the run-time protection is deterministic and not based on randomness.

To be efficiently used, the hardware support is accompanied by a software toolchain. This toolchain, used at design-time allows the developer to easily generate a secure code adapted to the hardware security support. Following section 2.1, providing a secure toolchain removes the complexity of security to developers while maintaining modularity and compatibility with the C standards. This first approach results in a framework called TrustFlow discussed in Chapter 4.

The second approach (2 in Figure 2.1) does not consider hardware modifications. It assumes that the latter is immutable and security features can only be implemented at

the software layer. Since this approach has fewer degrees of freedom than the previous approach, it is likely to be less robust. However, this approach aims at demonstrating that it is still possible to integrate a certain level of security in critical applications without degrading the criteria.

As the first approach, the second approach is top-down (see Figure 2.1). To ensure a robust memory exploits protection, the software run-time environment approach should also tackle the memory safety issue at its root: the initial corruption of critical data. Unfortunately, as revealed in the state-of-the-art in Chapter 1, tracking the data-flow entirely in software is costly. Thus, to not undermine the performance, this approach is more relaxed and focus on control-flow integrity. While this approach is less robust than pure data-flow integrity, it stills raises the bar for attackers.

Regarding the other criteria such as safety and determinism, the run-time environment aims at providing the same assets as the first approach without performing any change at the hardware layer. Of course, the secure run-time environment should be accompanied by a design-time software tool-chain with the same assumptions as approach 1. This second approach results in a framework called BackGuard discussed in Chapter 4.

To summarize, this thesis proposes three contributions. First, an open-source life-critical medical device system workbench for security purposes. The main purpose of such a platform is to serve as a relevant system to integrate and further benchmark security solutions dedicated to life-critical medical devices. This contribution may assess both approaches 1 and 2. Second, this thesis proposes TrustFlow and BackGuard that follows two previously exposed approaches. One relies on both hardware and software and the other completely on software to ensure memory safety. These two contributions aim at providing a user-friendly framework where developers can easily integrate security in critical systems taking into account all the criteria outlined in section 1.1.3.

3 SecPump

Summary of the Chapter

This chapter presents SecPump a life-critical system workbench tailored for security assessments. Presently, medical devices are closed-source and protected by commercial constraints. These constraints prevent this thesis from performing attacks on these devices as well as implementing appropriate countermeasures. To fill this gap and conduct the approaches exposed in Chapter 2, this Chapter presents SecPump. SecPump is an open wireless insulin pump model dedicated to security. The platform is specifically tailored for countermeasures development against the numerous security flaws related to medical devices. First, this chapter highlights the motivations of such a platform, its assets over existing work, and its interests for security. Second, the chapter details the functioning of the system and its various operating modes. The results show that SecPump is relevant enough to simulate the behavior of a classical insulin pump. Then, both hardware and software security vulnerabilities on such a platform are discussed and a set of memory safety exploit is showcased. These demonstrations are thereafter used to assess security countermeasures. Finally, SecPump aims at being open-source. Its assets over the existing work are presented in the last section of the Chapter.

Contents

3.1 Motivation

To further anticipate the design of appropriate security protections that target life-critical medical systems, this work requires a representative system workbench. Of course, such a system workbench should closely follow the assumptions highlighted in 1.1.1 and models the behavior of an off-the-shelf life-critical medical device. Unfortunately, most of the critical medical devices on the market are proprietary. They come with commercial constraints and intellectual property that even prevent this work to freely attack them, discloses vulnerabilities, and modify any components to integrate security features. To remind, currently, no specific bug bounty program [137] encourages security research on real medical devices.

Due to this lack of openness to research, the design of innovative security protection in medical devices is difficult to implement and to evaluate. Neither the software nor the hardware of existing devices on the market can be modified to accommodate innovation in the field of security. As a result, it seems that without a representative model, medical device security is not an easy matter for this academic thesis. Also, combined with the fact that real medical devices are expensive, it seems that studying their security is a challenge.

To address these issues and to further benchmark safe security protection this chapter proposes SecPump; a wireless medical device security system workbench. More specifically, SecPump is a platform that mimics the functioning of a real insulin closed-loop system. This thesis decided to model a wireless insulin pump because it is a critical system that has been actively developed and connected to various networks [13, 138] over the last few years. Overall, an insulin pump is a consumer device often used outside of hospitals. With fewer functionalities, it can be easily modeled on an off-the-shelf microcontroller with integrated wireless protocols. The main purpose of SecPump is thus to model the activity of a wireless insulin pump without requiring the purchase of expensive external mechanical components that are often not very useful in hardware or software security assessments.

In a second step, this work proposes to make SecPump open-source in the hope that it will be used in further security researches. Indeed, the platform aims at contributing to the awareness of security and the acceptance of new concepts compatible and maybe implementable in closed-sources systems. Overall, there is comparatively little information on the web regarding attacks that target critical embedded systems, rather than mobile and personal computers application. Although some methods of attacking personal computers and embedded systems are similar, implementing countermeasures on a critical system requires consideration of additional issues such as safety assessment and real-time constraints [136]. These constraints are not required for most other types of computing systems. Thus, such an open platform opens up new opportunities in the design of innovative security assessments dedicated to embedded systems. On the red-team side, SecPump can be used as a system workbench to show the impact of security vulnerabilities on a cyber-physical system. For instance, both software attacks (e.g., memory corruption, mal-

ware injection) and hardware attacks (e.g., side channel, glitch attacks) can be triggered on the pump. On the blue-team side, SecPump can be extended to accommodate novel countermeasures by considering determinism, safety, and real-time aspects.

Initially, this chapter reviews the open-existing platforms that inspired the design of SecPump. Section 3.3 exhibits the wireless insulin pump model, its software architecture, and the various algorithms that govern the functioning of the device. To showcase that SecPump is a suitable target for security assessments, section 3.4 displays various security attacks performed on the wireless pump. Finally, the last section is dedicated to the open-source assets of the platform regarding the related work. It shows how SecPump complements the existing platforms.

3.2 Open Source Medical Devices

Wireless insulin pumps are an excellent example of life-critical medical devices. These devices are used by both hospitals and individuals in the treatment of diabetes. More precisely, the purpose of these devices is to mimic the pancreas achieving fine-grained glycemic control. Emergent wireless insulin pumps are proved to be fairly effective at delivering perfect doses of insulin maintaining blood sugar at a correct threshold. Usually, a healthy, non-diabetic person has a blood glucose level of around 85mg/dL in the fasting period. When the latter eats a meal, sugar enters the bloodstream. As a result, the pancreas releases insulin so that the glucose in the blood is absorbed by the muscles to produce energy. The natural production of insulin by the pancreas lowers glycemia to keep it between 80 mg/dL and 120 mg/dL. With a type I diabetic, the pancreas is deficient and loses its ability to produce insulin. Thus, when a meal is taken, the blood sugar level does not decrease and can lead to health problems. Above 180 mg/dL the body starts to release glucose in the urine, and a patient is considered hyperglycemic when the glycemia exceeds 270 mg/dL.

Although many insulin pump devices are proprietary, several models, academic implementations, and guidelines have been proposed so far [139]. This section reviews the existing open-source medical devices and studies their feasibility to serve as a platform for security assessments.

The Generic Infusion Pump (GIP) project [139] launched by researchers from the FDA [4], the Center for Devices and Radiological Health (CDRH), and the Office of Science and Engineering Laboratories provides several methods and guidelines in the design of critical infusion pump devices. Many documents from this project are open-source and aim at serving as a safety design reference model to identify several hazards in various medical infusion pumps. These documents are intended to be used by both academia and manufacturers. More precisely, it encourages life-critical devices manufacturers to refer to the guidelines early in their design process to validate basic safety properties.

Several works, particularly in the field of safety, have been carried out on the GIP project. An implementation of a formally verified infusion pump has even been made available by the University of Pennsylvania [139]. However, the latter is slightly different from the expectations of this work. First, the proposed model is a classical hospital infusion pump that is not wearable and generally not available to the general public like insulin pumps. Second, the proposed implementation is generic and hardware-independent, it runs on a complete Linux operating with virtual layers that exhibit deterministic infusion operations. This work is rather positioned in the emerging field of IoMT. It requires a wearable medical device model much more restricted than GIP. An ideal model would if possible be based on low power microcontroller(s) with a firmware written in bare-metal or based on a lightweight real-time operating system such as FreeRTOS [29]. Finally, the GIP model does not consider the security issues related to the connection of a medical system. However, it seems that the IoMT is following a different trend [9]. Increasingly more manufacturers are implementing network protocols in their devices [18] to enable

remote access control and data gathering. As a consequence, an ideal security-oriented model should implement at least one wireless feature commonly integrated into IoT.

The Open-Source Syringe Pump library [88] provides an open implementation of a syringe system based on Raspberry Pi along with RepRap 3-D printers plans. The main purpose of the Open-Source Syringe project is to provide a customizable inexpensive medical system for research purposes affordable by all. The major components of the system involve a Raspberry Pi, a stepper motor, and the mechanical parts such as a sliding screw and other pieces that should be printed in 3-D. Figure 3.1 displays an overview of the Open-Source Syringe pump. First, The Raspberry Pi hosts a web page accessible from the network. This web interface allows an operator to calibrate the syringe, adjust its position, and define the injection speed. Once set up, the control of the syringe is achieved with a python script running on Raspbian the Raspberry Pi GNU/Linux open-source operating system. The latter interfaces between the drivers and the mechanical parts of the system to perform an accurate control. The mechanical parts of the syringe integrated with a stepper motor must be printed using an open-source RepRap 3-D printer. In comparison with commercial pumps, the open-syringe project claims that the entire system can be built by spending less than 5% of the price of a real commercial pump.

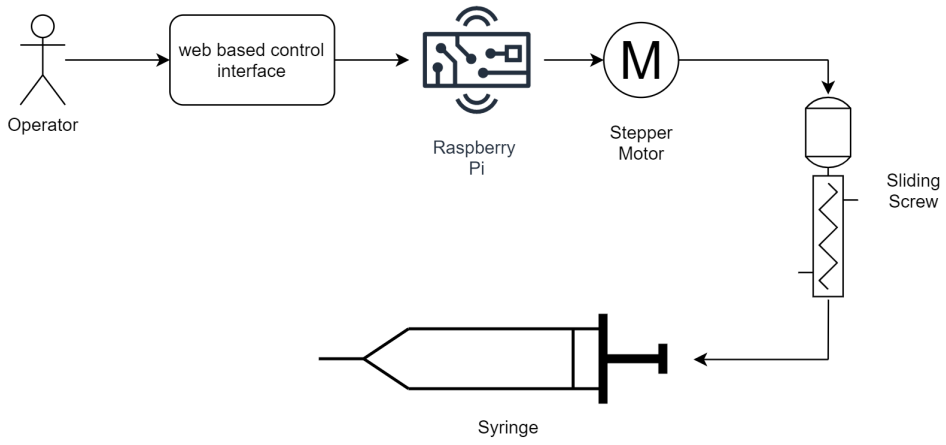

Figure 3.1: Open-Syringe

Unfortunately, accessing a 3D printer, and buying certain parts to create a functional system is convenient. Moreover, buying or accessing a 3D printer can be expensive and/or time-consuming. To mostly focus on security aspects, this research requires a portable demonstrator without relying on external mechanical components. Although mechanical vulnerabilities are relevant to real medical devices, they are beyond the scope of this thesis. Finally, from a certain point of view, the open-source syringe project is much more aimed at students and researchers willing to learn how to build a medical device rather than performing security assessments. For instance, the open-syringe does not model the impact of a security vulnerability directly on the safety of the user's medical device.

The Open Artificial Pancreas project (openAPS) [8] provides an open-source connected artificial pancreas using a Raspberry-Pi. This project, initially launched by Dana Lewis

in 2014, aims at automating insulin injection for diabetes using an open-source computer system that controls a commercial insulin pump. Originally, insulin injection using commercial pumps was achieved manually by diabetics. First, the latter had to prick itself to measure his/her blood sugar. Then, depending on the measurement, the diabetic had to self-inject an insulin bolus to bring his/her blood glucose back to an acceptable threshold. Since this method is not the most practical for patients, Dana Lewis tried to improve it by creating one of the first closed loops systems to automate the process. Such a system involves four components displayed in Figure 3.2. The first component (1) is a wireless Continuous Glucose Monitor (CGM) placed under the skin that constantly measures a patient's blood sugar. The measured value is sent to a remote smartphone (2) via a wireless protocol so a patient can visualize it. Then, the smartphone sends the glucose measurements to an off-the-shelf Raspberry Pi computer (3) via Bluetooth. The Raspberry Pi receives the Bluetooth data from the smartphone and processes it to define the amount of insulin to inject so that the patient blood sugar remains within a pre-defined targeted blood sugar level. Finally, once the insulin quantity is computed by the Raspberry Pi, it is transmitted to a commercial pump (4) via a radio protocol to process the injection. The injection regulates the glycemia of the patient closing the loop of the whole system.

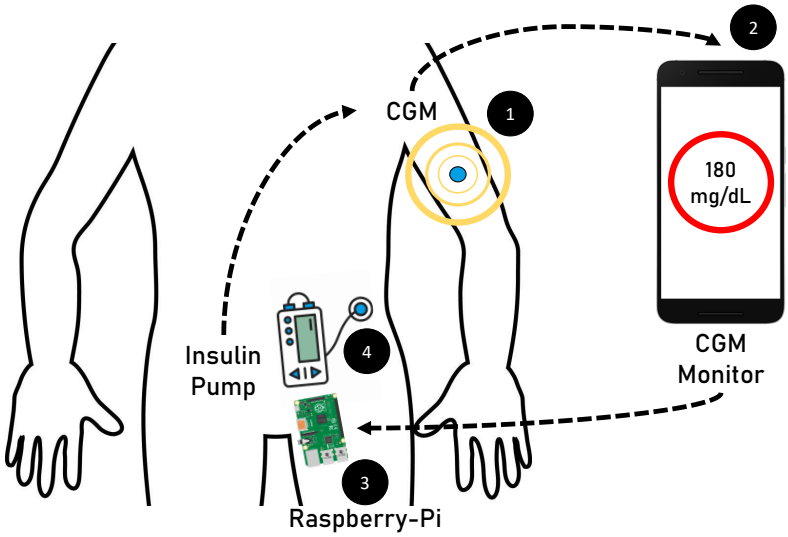

Figure 3.2: OpenAPS closed-loop

Unfortunately, OpenAPS is not completely open-source. The closed-loop system involves a closed source commercial insulin pump from Medtronic (4). The ad-hoc control of the closed-source pump with the Raspberry Pi was achieved by Ben West [8]. The latter found a vulnerability in Medtronic insulin pumps allowing external computers to send commands to the insulin pump. So even if such a system perfectly models a wireless insulin pump, implementing countermeasures is restricted to the Raspberry Pi and not the heart of the pump.

At the end of the day, finding a simple open-source implementation of a cyber-physical system such as a medical device is not an easy matter. No existing open-source work

is really "plug-and-play", it either requires printing additional parts, buying expensive components or spending time implementing the model. Furthermore, it seems that the existing open-source cyber-physical systems are more appropriate for learning how to build a life-critical system than for security assessments thereof. Besides, most of the presented models do not integrate the latest low-power and portable IoT wireless technologies such as Bluetooth Low Energy, LoRa, etc. Consequently, they are not well suited to model recent IoMT devices and consider a wide range of security threats.

To perform accurate security assessments, this work requires a **portable** open-platform that closely models a medical device functioning with fewer **constraints**. In other words, such a demonstrator should be implemented on an inexpensive off-the-shelf **microcontroller** and/or an **FPGA** and should not rely on any additional sensors, mechanical components, and/or interactions with a real diabetic patient. Only the most security-sensitive components relevant to the purpose of this thesis should be supported to conduct the study. Finally, an appropriate IoMT medical device model should implement at least a low power **wireless** feature to communicate.

3.3 SecPump

Presently, no open-platform offers a portable, security-oriented, hardware-software based model implementation of a wireless medical device. Thus, to further model cyber-threats on life-critical devices, this chapter proposes SecPump a wireless infusion pump system workbench primarily based on an STM32 Nucleo F446-RE board.

3.3.1 A wireless pump model

The design of SecPump is inspired by the OpenAPS [8] closed-loop system. More precisely, Figure 3.3 displays how the SecPump system models the behavior of a wireless insulin pump.

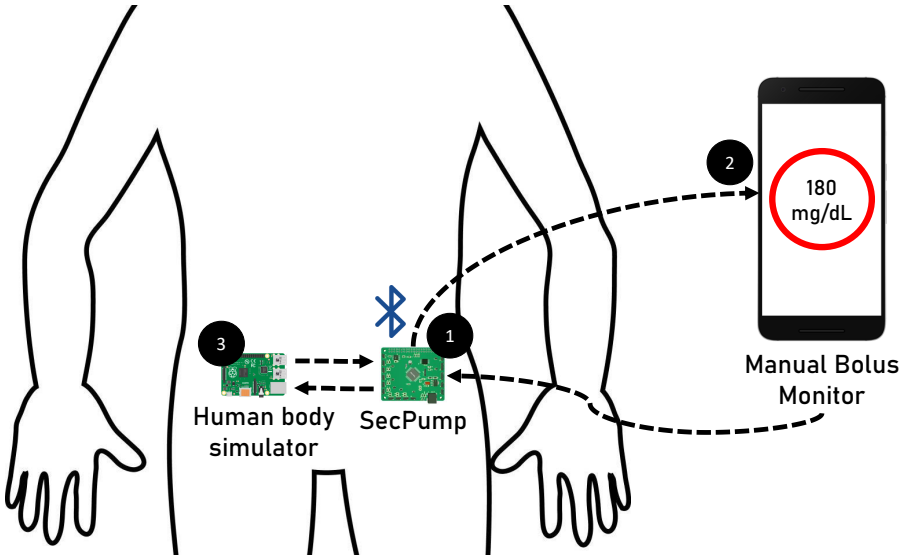

Figure 3.3: SecPump

In Figure 3.3, three major components rule the closed-loop system; SecPump (1), the smartphone (2), and a Raspberry Pi (3). The Raspberry Pi simulates the human glycemia behavior of a diabetic. It may seem logical, but simulating the glycemia behavior using an embedded system allows having a global model that operates without a real patient. In comparison with the previous Figure 3.3, SecPump regroups the Continuous Glucose Monitor (CGM) and the insulin pump into a single embedded system. SecPump reproduces the functionalities of a conventional insulin pump and CGM without the need for additional external sensors and mechanical components. Instead, SecPump simulates the behavior of these components using software routines.

SecPump is only the insulin pump of the whole system. To operate it requires a diabetic who is simulated by the Raspberry Pi. To represent the physical link between a theoretical diabetic and the wireless pump, SecPump is connected via the serial port to the Raspberry Pi. The Raspberry Pi models the glycemic regulation in response to insulin injection performed by the pump. The relationship between blood insulin levels and glucose is based on Bergman's differential equations. These will be detailed further

in the Chapter.

To regulate the simulated diabetic glycemia, SecPump integrates a Continuous Glucose Monitoring routine that performs glucose requests to the Raspberry Pi at regular intervals. Then, the CGM sends the glucose measurements to an external smartphone and updates the internal controller routine values. According to the operating mode (detailed in section 3.3.2), the controller routine injects (sends) an amount of insulin to the Raspberry Pi to regulate glycemia.

3.3.2 SecPump software model

Currently, SecPump proposes to regulate diabetics' glycemia using two operating modes: a manual mode and an automatic mode. Simply, the manual mode requires a smartphone. The latter should connect to the pump using Bluetooth Low Energy (BLE) protocol. Once the connection has been established, the CGM routine of the pump sends the simulated measured blood glucose at regular intervals to the smartphone. According to the measured glycemia, the patient can decide to manually trigger an insulin bolus injection by the pump. The amount of insulin to inject by the pump is sent through BLE from the smartphone. Then, the pump simulates the injection to the human glycemia simulator using the serial interface. Conversely, in automatic mode, SecPump regulates the blood glucose autonomously without requiring any manual bolus injection from the diabetic. Changing from one mode to another can be achieved using a BLE command. The functioning of SecPump and its modes are described in the following sections.

3.3.2.1 Manual mode

The manual mode requires the SecPump's user to monitor the glycemia from its smartphone and inject insulin accordingly. In other words, the manual mode can be compared to an open-loop regulation system such as in Figure 3.4.

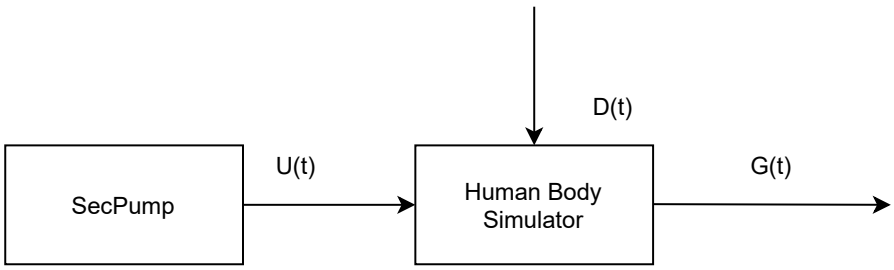

Figure 3.4: Open-loop regulation system

In manual mode, SecPump regularly measures blood glucose through the serial interface with the diabetic simulator (Raspberry Pi). This value, G(t) in Figure 3.9, is sent via BLE to a remote smartphone. The diabetic can thus proactively track glucose highs and lows and perform an insulin bolus injection U(t) (in Figure 3.9) to regulate glycemia.

Upon receipt of a BLE insulin bolus request, the pump triggers the sequence displayed in Figure 3.5.

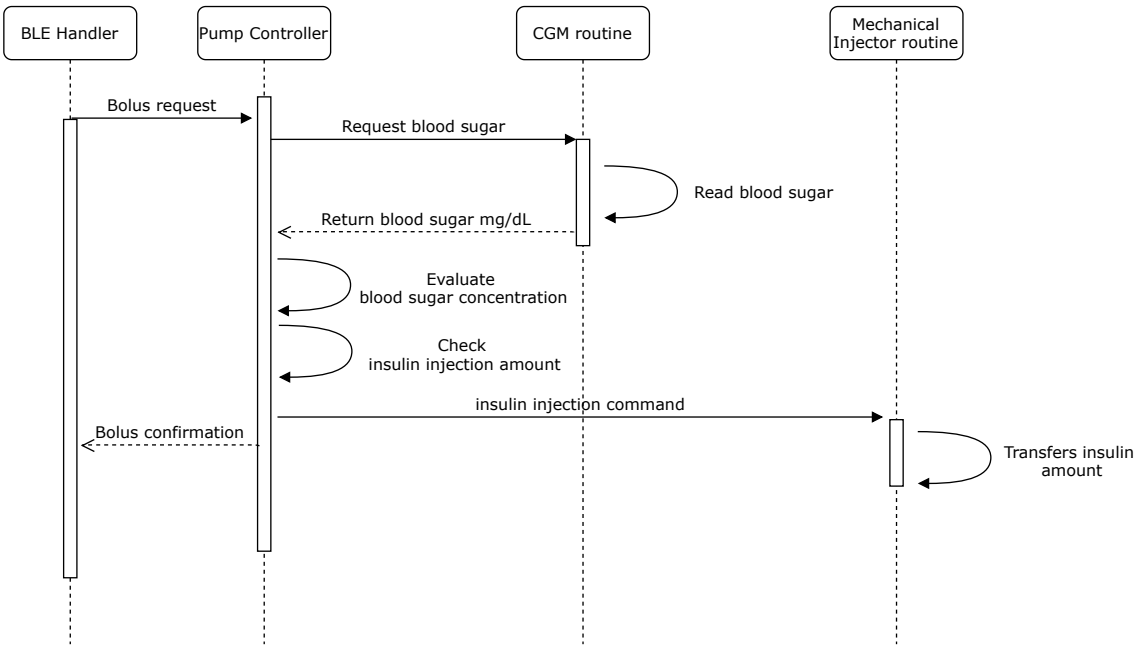

Figure 3.5: SecPump manual injection sequence

An insulin injection request involves three routines. First, the reception of a BLE packet triggers an interruption processed by a dedicated BLE routine. The packet is processed through a finite state machine and the extracted amount of insulin to inject is transmitted to the pump controller logic routine. The controller checks the Continuous Glucose Monitor routine to verify if the insulin request is overestimated with the current blood glucose. More precisely, the controller requests the CGM routine to read the serial port connected with the Raspberry Pi that simulates the diabetic glycemia in real-time. Upon reception, the controller checks both glycemia and insulin injection history. These checks ensure that the user is not trying to self-inject insulin with a low blood sugar level or a lethal dose. Finally, the insulin value is transmitted to the Raspberry Pi. This simulates a real insulin injection. Regarding the diabetic side, the Raspberry Pi simulates the absorption of insulin and decreases glycemia accordingly.

3.3.2.1.1 Human body modeling Modeling the balancing feedback between insulin and glucose is one of the most important parts of the platform. Although this thesis is not related to research in medicine, the balancing feedback must be representative. Indeed, to be efficiently used as a system workbench for security and safety assessments, SecPump should accurately model the insulin/glucose kinetics.

Unfortunately, modeling the blood glucose relation using software is not an easy matter. While several mathematical models have been proposed and evaluated so far [140], [141], none of them can reproduce the human body with 100% accuracy. One of the most famous mathematical modelization of the glucose-insulin relation is the modified Bergman's minimal model [141]. This model is usually used to simulate the glucose-insulin kinetics for a diabetic patient using four coupled differential equations stated below:

$$\frac{dG(t)}{dt} = -(p_1 + X(t))G(t) + p_1 G_b + D(t) \tag{3.1}$$

$$\frac{dX(t)}{dt} = -p_2 X(t) + p_3(I(t) - I_b) \tag{3.2}$$

$$\frac{dI(t)}{dt} = -p_2 I(t) + \frac{U(t)}{V_I} \tag{3.3}$$

$$\frac{dD(t)}{dt} = -d_{rate} D(t) \tag{3.4}$$

With the parameters described in Table 3.1.

Table 3.1: Modified Bergman's minimal model paramters

Parameter	**Unit**	**Description**
G(t)	mg/dM	Blood Glucose concentration.
X(t)	min^{-1}	Effect of active insulin.
I(t)	mU/L	Blood insulin concentration
D(t)	mg/(dL.min)	Meal disturbance function.
U(t)	mU/min	Exogenous insulin.
G_b	mg/dL	Basal blood glucose concentration.
I_b	mU/L	Basal blood insulin concentration.
V_I	L	The volume of insulin distribution pool.
p_1	min^{-1}	Glucose clearance rate independent of insulin.
p_2	min^{-1}	Rate of clearance of active insulin.
p_3	$\text{L}/(\text{min}^2.mU)$	Increase in uptake ability caused by insulin.
p_4	min^{-1}	The decay rate of blood insulin.
d_{rate}	min^{-1}	The decay rate of the meal disturbance

The discussion regarding the exact details of theses parameters is beyond the scope of the thesis study. For more in-depth details, this thesis refers the reader to more advanced publications in the field of medicine [141–143] that inspired the presented implementation. All these equations are integrated into the external Raspberry Pi using Python. The modeling of blood glucose is at the heart of equations (1) and (3). The integral of equation (1) gives the glucose concentration over time that is transmitted to SecPump. Of course, this equation is coupled with the disturbance equation (4) and the Active Insulin Effect equation (2). In more detail, equation (4) represents a patient's meal intake, and thus the perturbation induced on the blood glucose level given by equation (1). Finally, equation (3) models the insulin concentration kinetics over time. This equation depends on U(t) the function that usually models the insulin produced by the pancreas. However, in this study, U(t) is replaced is the output of the artificial pancreas: SecPump.

As the aim of the external Raspberry Pi is to simulate a diabetic patient, several

assumptions about the differential equations are outlined above. Initially, the Raspberry Pi simulates an untreated diabetic patient. As a result, it considers that the blood sugar of the patient and its basal value is extremely high as no insulin is produced. This leads to the following initial conditions:

$$G(0) = G_b = 280mg/dL \tag{3.5}$$

$$I(0) = I_b = 0 \tag{3.6}$$

$$X(0) = 0 \tag{3.7}$$

With these parameters [143]:

$$p_1 = 0.028735min^{-1} \tag{3.8}$$

$$p_2 = 0.028344min^{-1} \tag{3.9}$$

$$p_3 = 5.035e-5L/(min^2.mU) \tag{3.10}$$

$$p_4 = 0.05min^{-1} \tag{3.11}$$

$$V_I = 12L \tag{3.12}$$

The initial state of the blood glucose simulation is displayed in Figure 3.6. The measurements are taken over a simulated period of 24 hours considering that the theoretical patient is fasting. The blue curve represents SecPump in manual mode, no insulin is injected. The red curve represents the blood glucose level of the simulated diabetic patient.

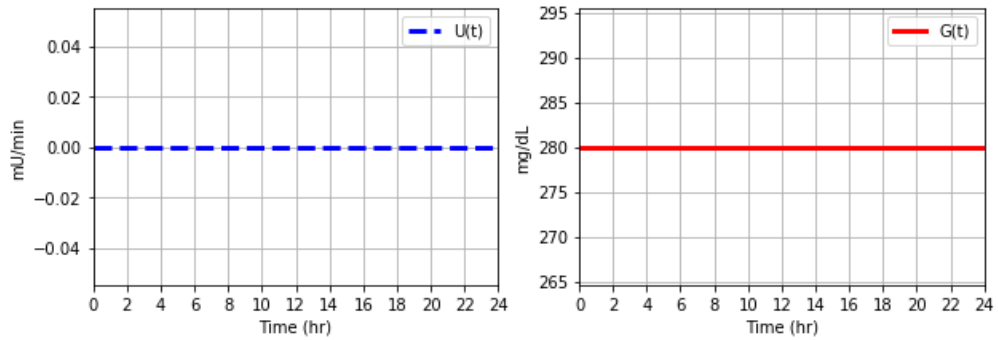

Figure 3.6: Manual mode initial state

In Figure 3.6, the blood glucose level is extremely high. According to the initial assumptions, no insulin is produced by the diabetic patient and no insulin is injected by

SecPump. As a consequence, the Raspberry Pi simulates the glucose that remains within the blood and is not absorbed by the human muscles.

To lower the blood glucose level, an injection sequence should be triggered using BLE. By first considering the system with no perturbations (no meals, D(t)=0), a bolus step response of the whole system with 17mU/min of insulin is displayed in Figure 3.7.

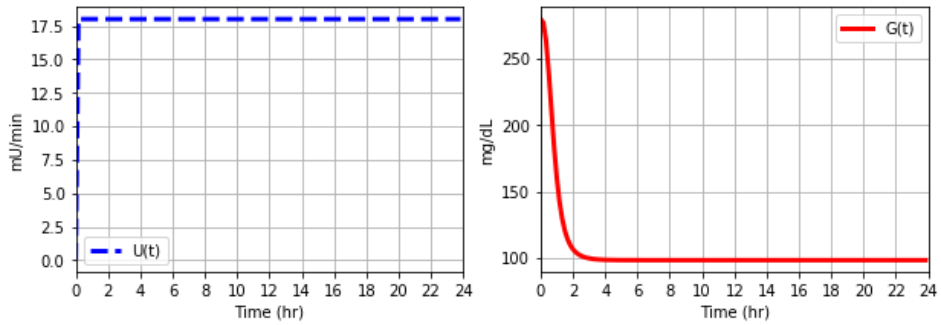

Figure 3.7: Manual mode step insulin injection

According to Figure 3.7, the step response reaches a new steady-state of around 100 mg/dL in two hours. This steady-state is within the blood glucose limits of a healthy person without excessing 100 mU/min insulin injection rate [143]. These measurements validate that the whole system operating in manual mode is reasonable for modeling a real insulin pump.

Unfortunately, the manual mode does not fully simulate an artificial pancreas. The two previous graphs result from measurements taken without meal disturbances. When random disturbances due to meal intake are introduced by the diabetic simulator, the blood glucose level in the human body still rises to 200 mg/dL (see Figure 3.8). In Figure 3.8, the glycemia simulator is programmed such that it introduces disturbances at mealtimes of 8:00 am, 1:00 pm and 7:00 pm respectively. These disturbances follow the differential equation (4) with a $d_{raterandom} value between 0.5 and 10 mg/(dL.min)$ [140].

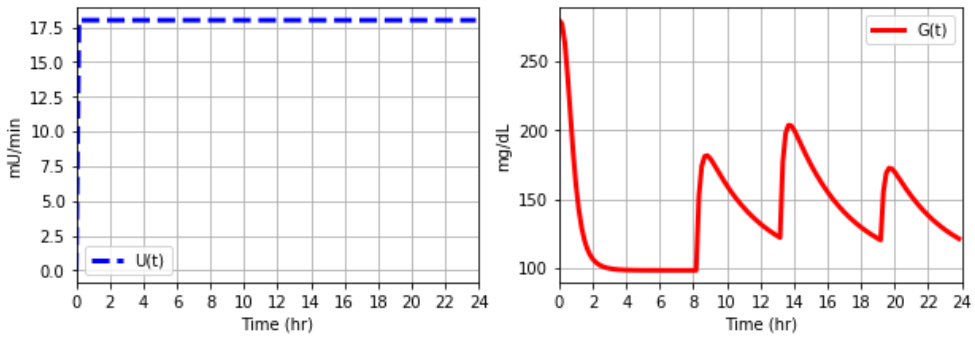

Figure 3.8: Manual mode meal perturbation simulation

Of course, to better regulate the blood sugar, the diabetic can manually adjust the pump injection rate according to each meal. The diabetic can also use an algorithm such as [8] on the smartphone to control the pump remotely. However, better regulation can be automated and practically implemented within the model.

3.3.2.2 Automatic mode

To achieve fine-grained insulin regulation on the model, SecPump proposes an automatic mode that regulates blood sugar using an integrated PID. This integrated PID is activated when switching from manual mode to automatic mode. It requires a target value at which the pump should maintain the patient's blood glucose. As a whole, the automatic mode operates as a closed-loop regulation system displayed in Figure 3.9.

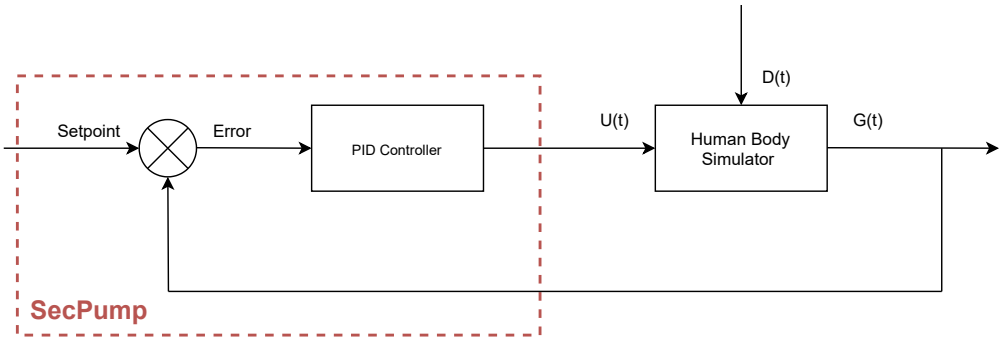

Figure 3.9: Closed-loop regulation system

The SecPump system measures blood glucose from the Raspberry Pi simulator. Using the setpoint and the measured glucose the PID determines the amount of insulin to inject within the system. More precisely, the main sequence of the pump in automatic mode is displayed in Figure 3.10.

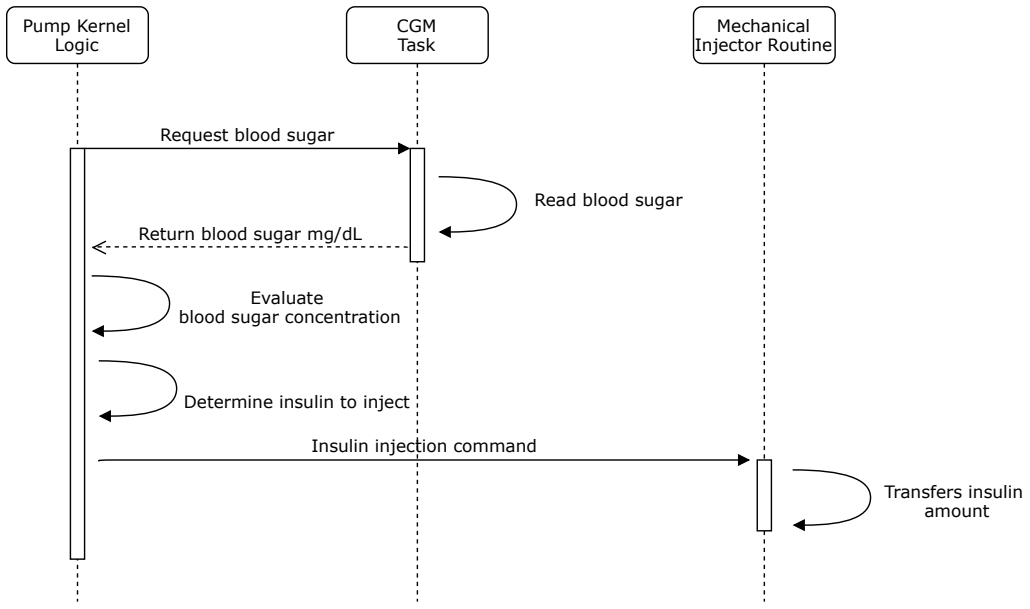

Figure 3.10: SecPump automatic injection sequence

The automatic mode is driven by two routines. First, the pump controller frequently checks the value of the blood glucose updated by the CGM routine. Glucose measurements are added to a pump's internal measurement history. According to the measured values the controller determines the error between the blood glucose and the setpoint. From this error, the PID injects a certain amount of insulin to lower the blood glucose.

3.3.2.2.1 PID Regarding the insulin regulation, the output of the PID (output of the pump) has a direct impact on equation (3) of the Bergman differential equations. As these equations are non-linear, we developed a simple Python interface based on the Jupyter framework to manually tuned the coefficients of the PID. This interface makes it possible to graphically model the behavior of the controller before programming the pump. In this thesis, the PID coefficients of SecPump are tuned empirically such that the blood glucose stimulated by the human simulator stays between 60 and 180 mg/dL [143]. We are aware that there are several techniques to determine the values of a PID accurately, even for non-linear systems. However, they are beyond the scope of this thesis. It should be mentioned that this can be the subject of ongoing work and further contribution to the pump. Also, the PID implemented in SecPump follows the Oral Glucose Tolerance Test (OGTT) [144] by ensuring that blood glucose of the diabetic simulator drops bellows 140 mg/dL at least 2 hours after meal intake.

The real-time functioning of the PID implemented in SecPump is displayed in Figure 3.11. In Figure 3.11, the blue curve represents the amount of insulin injected by the pump while the red curve represents the simulated blood glucose. Initially, the blood glucose is high and brought back to the setpoint around 85mg/dL by the PID in less than 4 hours. Finally, regardless of the magnitude of a rise in glucose following a simulated meal, the PID regulates the insulin injection so that the glucose level is below 140mg/dL at least two hours after the meal [144].

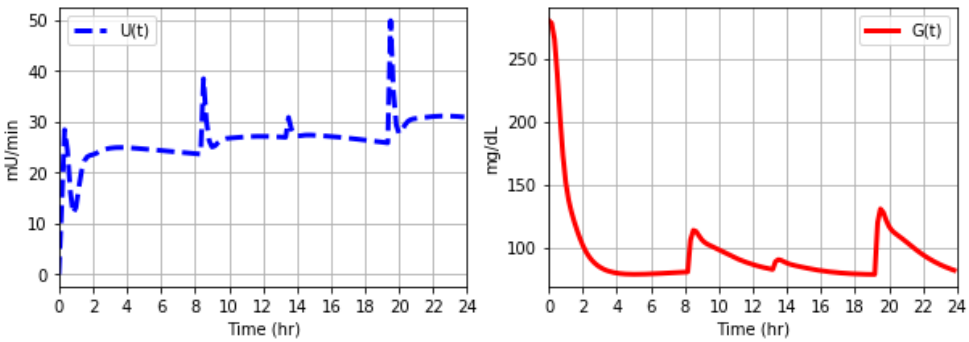

Figure 3.11: PID blood glucose regulation

Kc=-0.07, tauI=1, tauD=1.20

3.3.3 SecPump variants

To remind, in Chapter2 section 2.3, two approaches are proposed regarding the implementation of security in life-critical devices. One approach is hardware-software oriented and the other completely software oriented. Thus, to perform appropriate security assessments, several variants of SecPump are developed accordingly. Table 3.2 summarizes these variants.

The first variant in Table 3.2, SecPump-BLE, is the implementation described in the previous section. It targets an off-the-shelf STM32 microcontroller and supports the Blue-

Table 3.2: SecPump variants

SecPump version	Possible extension	Pump communication	O.S support	Target architecture	Lines of Code
SecPump BLE	Software	BLE/Serial	Bare-metal	STM32 ARM	18k
SecPump RT-BLE	Software	BLE/Serial	FreeRTOS	STM32 ARM	27k
SecPump RISC-V	Hardware Software	Serial	Bare-metal	Digilent Xilinx Arty-35T RISC-V	7

tooth Low Energy communication with a remote smartphone. This variant is flexible, it may be easily compiled without the manual mode. In other words, the Bluetooth Low Energy (BlueNRG) expansion module of the STM32 board is not indispensable to operate the pump. Of course, it removes wireless functionalities, but the latter can be replaced with serial communication.

The second variant of SecPump, SecPump-RT-BLE is just an enhancement of SecPump-BLE. While the previously exposed variant's sequences are orchestrated by a finite state machine, this version is based on the real-time system FreeRTOS [29]. Simply, all routines of SecPump-BLE are implemented and scheduled by the FreeRTOS operating system. On both SecPump-BLE and SecPump-BLE-RT, only the software layer of the implementation is open source. While it allows performing both hardware and software attacks demonstration on the platform, the integration of security is more restricted. Indeed, the ARM architecture is proprietary, it does not allow the integration of invasive hardware security blocks into the processor core. Regarding previous approaches discussed in section 2.3, both variants are mostly fitted for software-based security extensions, so the second approach.

Finally, SecPump RISC-V is an adaption of SecPump-BLE ported to run on a RISC-V processor using the open-source port on a Digilent Xilinx ARTY-35T provided by SiFive [145]. As the FPGA does not support wireless connectivity, any communication with the pump is performed through the serial interface. In comparison with the other variants, SecPump RISC-V is fully open-source from hardware to software. Indeed, unlike the ARM architecture, the RISC-V architecture running on the Digilent Xilinx ARTY-35T is open-source. It allows modifying the hardware layer to integrate security features and as the previous version, all the software layers as well.

3.4 Security Assessments

Above all, SecPump is a life-critical system workbench dedicated to security assessments. Usually, when performing security assessments on a system it is important to define the **context**, the **threats** and their impact, the **scenarios** of attacks, the **risks** and the **countermeasures**. As SecPump is a cyber-physical system, it is vulnerable to the same as known attacks as other existing embedded systems. However, as SecPump models a life-critical medical device, the consequences of related security vulnerabilities are a little different. Indeed, an attack can directly impact the safety of the system and even cause irreversible health damage to the user. The interesting point with security issues on life-critical devices is that they can cause unexpected safety issues that were not considered during the design. As part of the SERENE-IoT project Work Package 2 (WP2), we carry out an in-depth security study of a wireless pump inspired by the EBIOS method [146]. While insulin pumps are not only subject to memory safety issues, this section focuses on it to stick to the research direction of the thesis. The section 3.4.1 focuses on the memory safety threats of such a platform. It showcases a **scenario** of attack on the pump and the according **risks**. Finally, section 3.4.2 is an opening to hardware security.

3.4.1 Software Threats

At the software level, several modules of the pump can be targeted by an attacker. Regarding memory safety, the most critical software module remains the BLE connection. A memory safety vulnerability in the wireless module leaves the pump open to a remote attack. Conversely, without a wireless connection, memory safety issues can be considered less **threatening**.

Based on the memory safety **threats** linked to the connection of such a system to the network, we have developed several attack **scenarios**. To do so, an intentional stack-based buffer overflow bug is introduced in the BLE packet parser by removing a common boundary check in the SecPump software. This introduced vulnerability model a software bug as it is possible to find in existing wireless pumps on the market [17, 18]. Two memory exploits are developed accordingly to demonstrate the impact of such security vulnerabilities.

The first exploit is a control-flow attack. It leverages the buffer overflow vulnerability to overwrite a return address on the BLE's routine stack. The exploit diverts the execution flow on an injected sequence of address instructions scattered into the SecPump code address space. This attack breaks the control-flow graph of the insulin pump by combining both Return-Oriented Programming [53] and Jump-Oriented Programming techniques [60]. The executed gadget chain mimics an "insulin injection" sequence and transmits the insulin amount to the diabetic simulator using the serial interface. The impact of the attack is displayed in Figure 3.12. The blue graph displays the amount of insulin injected by the pump in real-time, the red graph shows the simulated blood glucose level, the black curve shows the concentration of insulin in the patient's blood and the green curve shows the effect of active insulin. One can observe that the attack is performed at

around 4 pm in the graph. At this time, the controller injects up to 300 mU/min amount of insulin inducing a significant drop in glucose and a significant lethal increase of insulin in the diabetic.

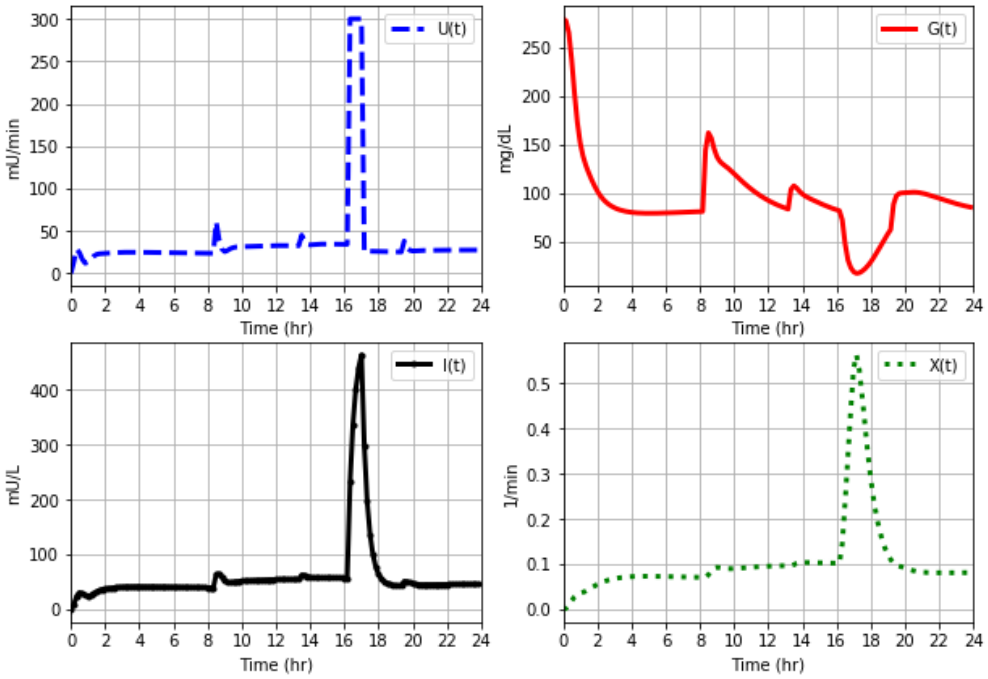

Figure 3.12: Remote code execution in SecPump

The second exploit is a non-control data attack. As the previous attack, this exploit leverages the same buffer overflow to perform arbitrary writes in the pump memory. Instead of overwriting a return address stack, this attack triggers a data pointer on the stack that indirectly modifies the amount of insulin to inject to the human body. Unlike the previous control-flow attack, this data attack does not break the control-flow of the application but results in the same significant threats and results displayed in Figure 3.12.

Of course, these attacks are feasible because the demonstrator does not implement any specific security protections. However, they can be reuse to access further security protections. Also, these attacks showcase a worst-case scenario that may impact a critical wireless device.

3.4.2 Hardware Threats

Hardware attacks are fully applicable to SecPump. Indeed, by having physical access to a device an attacker may achieve a wide range of attacks. For instance, by using passive and/or active hardware attacks such as side-channel and fault injection an attacker leverages both the physical properties and the physical access to the device to leak/modify sensitive assets such as cryptographic keys [147]. While the pump is developed without any particular security countermeasures, its cryptographic module and other sensitive parts may be jeopardized by hardware attacks. As for software attacks, SecPump is the perfect target for research in lightweight cryptography that should be robust to hardware attacks

and integrated into constrained critical systems [148].

Also, hardware attacks can target both the control-flow and the data-flow of an application. This causes undefined code behavior leading to both security and safety issues. The interesting fact with hardware attacks is that it induces disturbances into the system. These disturbances result from attacks but can also be due to interactions with the environment, which is the case for life-critical medical systems that should operate under any conditions. For instance hardware attacks can be used to induce system failures such as desynchronizing the Real-Time Clock of the pump, inducing false positives/negatives during safety checks, flipping bits from the RAM and registers, and even simulates peripheral errors (CGM for instance). As a result, SecPump is a suitable platform to also assess novel safety protections and recovery mechanisms. For more in-depth details about hardware security assessments on life-critical medical systems, this section refers to interesting works that used SecPump as a security system workbench [147].

3.5 Comparison with other works

This work also proposes to release SecPump as an open platform for the community. We believe that such a platform is a great asset and may encourage further researches regarding life-critical devices' security. This section compares SecPump to the related work exposed in section 3.2. It should be mentioned that the purpose of this contribution is not to compete with the related work, on the contrary, SecPump proves to be a complementary platform with a focus on security.

Table 3.3: Comparison table

	openAPS	**GIP GIIP**	**Open Syringe**	**SecPump**	**Commercial Pumps**
Open-source	◐	●	●	●	○
Price	$750+	-	<$100	$30 - $130	$260 - $5000
Practicality	◐	◐	◐	●	-
Connectivity	Bluetooth Serial	-	Ethernet Wifi	BLE	-
Security-oriented	○	◐	◐	●	○

●= *Yes;* ◐= *Partial;* ○= *No;* - = *Not defined*

Table 3.3 compares the existing insulin/infusion pump platforms regarding five criteria. Whether they are **open-source**, the **financial costs** to get a functional demonstrator, if it is simple to set up the demonstrator (**practicality**), whether the existing work handles **wireless** features, and finally if the platform is well suited for **security** assessments.

According to Table 3.3, three works offer a fully open-source IoT platform; Open-Syringe [88], SecPump, and the University of Pennsylvania that provides an implementation of a Generic Patient Controlled Analgesic infusion pump that follows the GIP guidelines [139]. They mostly provide complete code to implement a functional medical device. In comparison, openAPS [8], provides a Raspberry-Pi based artificial pancreas that controls a commercial insulin pump. Unfortunately, while the artificial pancreas insulin regulation algorithm running on the Raspberry-Pi is fully open-source, the commanded pump is not, making the whole system not entirely available.

Price is also a limiting acceptance factor for academic research. According to [88], a commercial pump can cost up to $5000 which is not always affordable. Also, even if the openAPS project is based on a Raspberry-Pi that costs around $50, it communicates with a closed-source commercial insulin pump that costs around $700. Both open-syringe and SecPump are affordable. The open-syringe project claims that the syringe can be built by spending less than $100 including, Arduinos, the Raspberry-Pi, and printing the 3D mechanical parts. However, it does not include the 3-D printer price. Regarding SecPump, the BLE-RT version only requires an off-the-shelf STM32 F446-RE with the

BluNRG extension board. The total cost less than $30. However, the implementation price of SecPump rises when the platform is implemented on an FPGA. For instance, the SecPump RISC-V version requires a Xilinx Digilent Arty 35-T FPGA that costs about $130. Finally, the GPCA implementation [139] provides a generic code that is not tied to any specific hardware. According to the project, it should be ported/adapted to a real infusion pump hardware or a microcontroller.

For practicality, Table 3.5 excludes the commercial pumps. They are supposedly not requiring any manual implementation to work. The term practicality in Table 3.5 specifically targets the ease of implementation of an open platform to quickly perform primary technical security assessments. GIP provides many clear design documents for life-critical system developers. However, it does not provide an implementation tied to specific hardware to begin security assessments quickly. Thus, to perform security assessments, a primary step is to develop a prototype or adapt the specifications. Also, in comparison with SecPump, the GPCA implementation is more safety-oriented than SecPump, but it does not model the human body interactions. In addition to purchasing the components, both Open-Syringe and openAPS requires several steps to reach a functional demonstrator. For instance, Open-Syringe requires the printing of several 3D parts with the assembly and connection of several motors. Unfortunately, not everyone possesses a 3D printer. On the other hand, openAPS requires a compatible commercial insulin pump with the correct firmware version to be connected with the Raspberry Pi. In comparison, SecPump is more practical, it does not require any mechanical parts to be assembled, it only needs the open-source code to be compiled and flashed to an off-the-shelf microcontroller.

When dealing with security in IoT, connectivity is an important concern. Almost all the medical devices in Table 3.5 integrates a wireless communication protocol. This commonality is a great advantage for security research. Due to the IoT, embedded systems may be increasingly connected to the network and exposed to various new attack vectors. Researchers should be able to work on these protocols to improve their security and find better ways to integrate them along with critical applications. The variety of wireless features offer diversity in attacks targeting protocols.

Finally, regarding security, OpenAPS and commercial pumps come with commercial constraints that prevent attackers to freely attack them. Only the Raspberry Pi of the OpenAPS project can be targeted. On the other hand, GIP aims at improving secure design in medical systems providing open-source methods and guidelines. However, GIP is not an implemented platform tailored for security assessments. Finally, Open-Syringe is not security-oriented. The framework aims at teaching how to build open-source medical devices. However, it should be noted that Open-Syringe has been adapted in software security works to assess the robustness of control-flow integrity countermeasures [86].

3.6 Conclusion

This chapter introduces SecPump, a new open medical device platform tailored for security assessments. Presently, medical devices are closed source and protected by commercial constraints. This lack of openness to research makes it difficult to perform security assessments on these types of devices. Unfortunately, commercial constraints do not protect medical devices from security threats. It has been discovered that these devices are highly vulnerable to various cyber-attacks ranging from data theft to lethal security exploits [9].

To conduct relevant security contributions in the field of life-critical devices, this thesis requires a representative model to work with. Unfortunately, after reviewing the state-of-the-art, we did not find a completely open-source platform that models a wireless insulin pump at a lower cost, and that is tailored for security and safety assessments. As a result, this chapter proposed SecPump, a security-oriented wireless insulin pump system workbench that mimics the behavior of a real insulin pump. SecPump requires an off-the-shelf microcontroller or FPGA and is connected to an external Raspberry Pi that simulates the behavior of a diabetic's glycemia. More precisely, the Raspberry Pi simulates the insulin-glucose kinetics of a diabetic following the extended Bergman minimal model [142]. SecPump interacts with the Raspberry Pi model by acting as an artificial pancreas. It regulates the insulin of the simulated diabetic thanks to a custom simple PID controller. The resulting implementation demonstrates that the PID can keep the insulin level of a diabetic within an acceptable range passing the OGTT test. We consider these measurements sufficiently representative of a real system to perform various security assessments.

As the second step, some security **threats** and **scenarios** in relation to the thesis are showcased on the platform. Two intentional memory safety vulnerabilities are introduced and exploited on the pump. The result shows that a simple linear buffer overflow vulnerability in the wireless communication protocol of the pump can induce several security and safety issues. These exploits are not new and follow the attack techniques widely explored in the state-of-the-art. However, they aim at being further reused to assess the approaches highlighted in 2.3. Regarding hardware security, SecPump is suitable to serve as a system workbench for both attacks and defenses. However, the study of hardware-related security issues is considered out of the scope of this thesis.

Finally, this work aims at being released as an open-source platform for the community. We believe that such a minimalist and inexpensive platform may promote security innovations in life-critical medical devices. It may open new security opportunities different from personal computers by considering additional issues such as safety and real-time constraints.

Final note: The wireless model of the pump presented in this chapter has significantly evolved throughout the thesis and culminated in the final version incorporating a PID insulin controller coupled with a diabetic simulator. SecPump has been published in the 2020 IEEE Embedded Systems Letters journal [149]. The STM32-based version

of SecPump also inspired the design of a lightweight cryptographic AES, resistant to side-channel attacks. The related paper: “CONFISCA: an SIMD-based CONcurrent FI and SCA countermeasure with switchable performance and security modes” will soon be submitted for publication. SecPump-BLE-RT also served as a security demonstration support for the “BackFlow: Backward Edge Control Flow Integrity Enforcement for Low End-ARM Real-Time Systems” publication at the University Booth Demonstration at IEEE DATE 2020. Finally, within the Serene IoT framework, the security analysis performed on SecPump contributed to the LCIS security analysis of WP2. The pump model will be open-sourced on Github at the end of this thesis. This will include the code of the platform, the documentation, and the security analysis.

4 TrustFlow

Summary of the Chapter

This Chapter introduces TrustFlow, the first approach of this thesis. TrustFlow is a novel hardware/software co-designed framework that provides efficient fine-grained control-flow integrity protection for critical embedded systems. TrustFlow is composed of an LLVM-based compiler toolchain that generates a secure code. The latter is then executed on an extended RISC-V processor that keeps track of sensitive data using a trusted memory. The obtained results show that the contribution is practical for developers, providing a high level of trust in life-critical embedded systems with an execution-time overhead of less than 1%.

Contents

4.1 Motivation

Recent studies revealed that many critical Internet of Medical Things (IoMT) devices on the market are vulnerable to intrusion and software exploits [9]. For instance, in 2017, the CVE-2017-12718 [18] revealed several buffer overflow vulnerabilities allowing remote code execution on commercial infusion pumps. This case demonstrates that today's embedded IoT systems are beginning to face the same type of attacks as servers and personal computers. However, in the case of an infusion pump, a software exploit can directly threaten a patient's life. Unfortunately, as manufacturers are increasingly connecting devices to the Internet without sufficient security measures, serious security issues such as CVE-2017-12718 are expected to grow in the coming years.

According to both Chapter 1 and 2, one possible reason for this poor security trend comes from the fact that **time-to-market**, **innovative features**, and **costs** often take precedence over the integration of security. Also, another potential reason is that manufacturers of embedded applications lack security experts [10]. Of course, healthcare manufacturers are well trained about safety issues and various hazards in medical devices for decades. Many medical devices on the market have been validated by recognized institutions [4] and to date are very safe. However, healthcare manufacturers are less aware of cybersecurity issues. Indeed, connecting a medical device to the network is a relatively recent trend. Exposing wireless devices to the Internet raises new cybersecurity issues that manufacturers were not confronted with in the past. The **integration of security** in embedded devices is a **difficult** task, and without security expertise, the design and the integration of the latter are usually poorly achieved.

In contrast, when a new critical device is released, hackers try to find programming errors to exploit them. One of the most common weaknesses is a memory corruption vulnerability. Such vulnerabilities give an attacker the ability to inject arbitrary code into the application at execution-time and divert its execution flow. Considering a life-critical medical device, such attacks can give the hacker the ability to modify life-critical values harming a patient's life [12, 15, 18, 37].

Interestingly, regarding memory exploits, Chapter 1 of this thesis revealed an abundant state-of-art of effective protections. However, despite all the efforts that have been made to enforce memory safety, it seems that emergent wireless medical devices are still vulnerable to a wide range of memory exploits. For instance, [17, 18] found multiple code injection attacks allowing remote code execution on commercial infusion pumps. These code injection attacks revealed that the system did not possess even the most basic memory safety mitigations highlighted in the state-of-the-art. Security is an important concern for critical medical devices. They are in direct interaction with the human body and may cause irreversible harm in case of a security breach. Knowing the existing state-of-the-art defenses, manufacturers should have implemented at least one of them in critical devices. Unfortunately, this is often not the case. As extensively discussed in Chapter 2, several factors are limiting the acceptance of memory safety protection in life-critical devices:

1. The **time-to-market** dynamic imposed by IoT is incompatible with advanced and

complex **security tests** that are time-consuming. Manufacturers want to release new innovative devices full of features to get to the market first.

2. Life-critical device manufacturers lack **security resources** [10]. While they have the necessary resources concerning safety issues, wireless features expose critical devices to new attacks that they were not confronted with in the past.

3. Many concepts offer an **incomplete** defense [40, 82]. Most of the CFI solutions provide either backward-edge or forward-edge protection. So an attacker can always attack forward-edges when backward-edges are protected and vice versa. As medical devices may be life-critical, they require **complete** protection.

4. Memory safety is a trade-off between the **execution-time cost** and the **accuracy** of the protection. It is often considered that memory protection inducing more than 5% execution-time overhead will never be accepted by manufacturers [40]. Unfortunately, a lot of research work does not take this constraint into account [129].

5. Most of the hardware-based countermeasures lack **software support** [40] needed to enable them with minimal developer effort. As long as developers have to rewrite some parts of the software by hand, security solutions may be unpractical and hardly accepted.

6. Among all the protections, the state of a system after detecting an attack is never tackled. Detecting an attack is only the first step, the second step is to move the system to a **safe state** that does not impact the users' safety. None of the existing countermeasures exhibits innovative techniques to recover from memory attacks.

7. Finally, according to section 1.4, many security defenses, and concepts break the C programming language's modularity and portability [40]. They require an application code to be instrumented afterward. This is not modular and even impractical in production software development environments. It forces developers to perform security tests at the end of a system's development phase. Besides, instrumented codes are often not compatible with non-instrumented third-party libraries.

To summarize, effective memory safety protection for life-critical embedded systems should closely follow the criteria listed above and in section 2.2 to have a chance to be integrated into medical devices. To fill this gap, this Chapter presents TrustFlow, a **practical framework** for memory safety in critical systems. Trustflow aims at keeping a sufficient tradeoff between **simplicity**, **security**, **safety**, and **performances**. TrustFlow is designed to provide fine-grained control-flow integrity support using data-flow integrity. The main contributions of this Chapter are summarized below:

- **TrustFlow environment**, an innovative hardware design that can **prevent**, **detect**, and **treat** memory-based exploits at the **data granularity**. TrustFlow is an enhanced shadow stack (trusted memory) interfaced with a RISC-V processor pipeline.

- **TrustFlow framework**, an innovative software **toolchain** able to generate secure code for the **TrustFlow environment**. The toolchain provides several control-flow integrity levels. The **TrustFlow framework** integrates a static analyzer that can determine the security cost of an application before its implementation on real hardware.
- **A security evaluation** of the TrustFlow framework based on the RIPE benchmark suite.
- **A benchmark** of the TrustFlow environment that exposes: the memory footprint of real representative critical applications, the execution-time overhead induced by TrustFlow using CoreMark [150] benchmark suite, and the hardware cost of our implementation.

4.2 Approach

As TrustFlow deals with life-critical embedded devices, it follows the assumptions exposed in section 1.1.3. To remind, here are the major assumptions concerning the design of medical applications and the attacker's capabilities that the TrustFlow framework considers in this Chapter.

Critical software assumptions: TrustFlow assumes that critical embedded systems mostly follow "The Power of Ten Rules for Developing Safety Critical Code" [38,46]. These guidelines are not limited to but mostly forbid:

1. *Dynamic memory allocation after initialization*: the use of dynamic memory allocation with "*malloc*" introduces memory fragmentation and potential memory leakage (dangling pointers). The usage of dynamic memory allocation is **nondeterministic** and thus cannot be **safe**.

2. *More than one level of pointer dereferencing*: The use of pointers is one of the major sources of **programming errors**, **safety**, and **security** issues. Although their use is practical, it must be minimized and restricted to the smallest scope as possible.

3. *Usage of recursion and goto statement*: recursion introduces cycles into control-flow graphs that complicate the verification task of **static analyzers**. Besides, it is not an easy matter to cover all the tests to determine the upper bound of a recursive function. As a consequence, recursion can induce prohibitive unexpected large usage of stack memory in life-critical systems.

4. *Function pointers*: The use of function pointers is discouraged according to the power of ten rules. Indeed, function pointers prevent **static analyzers** to prove the absence of recursion. Also, they restrict the types of checks that can be performed by static analyzers. The rules claim that the use of function pointers should be justified if used. According to the reviewed rules [38], it seems that "constant function pointers, for instance, stored in lookup tables, pose no risk to safe execution or code analysis". Function pointers are thus considered by the TrustFlow framework.

Threat model assumptions: As this work focuses on memory corruption, it assumes that executed critical software may contain software bugs such as *buffer overflows* that can be turned into successful exploits. To be more precise, this work assumes that the attacker is powerful and may corrupt any in-memory data they want, either locally (on the interface of the embedded system) or remotely (through a communication protocol). Thus, this threat model considers software attacks that are performed at execution-time with the following assumptions:

1. The running critical application software is statically verified and does not contain any malware. We consider that the attacker possesses the firmware and can find all vulnerabilities. However, the application code is immutable, the attacker cannot modify it nor upload a malicious process into the application data-space before triggering a successful memory exploit.

2. While embedded systems are also prey to hardware attacks. Hardware attacks are out of the scope of this work. It considers that software attacks are operated remotely and not physically on the system. Hardware attacks can also be used to learn how the system operates. The threat model has no problem with this. In 1, it already considers that an attacker possesses the firmware code and exploitable vulnerabilities. The aim of TrustFlow is not to obfuscate or complicate the vulnerability research process, but to accept vulnerabilities and prevent successful exploits.

3. Software attacks such as row-hammer and cache side-channel attacks are considered out of the scope of the study.

Approach overview: To address the criteria exposed in section 2.2, TrustFlow follows the top-down approach proposed in section 2.3. Unlike most existing defenses [40], this approach tackles the memory safety issue from initial software design-time to its execution on the hardware. This thesis truly enforces that security should be integrated as early as possible in the software design process with simple tools that require **minimal human efforts**. Then, during execution-time, the security assets enforced by the software should be reinforced by hardware primitives. Figure 4.1 displays the overview of the approach in line with section 2.3.

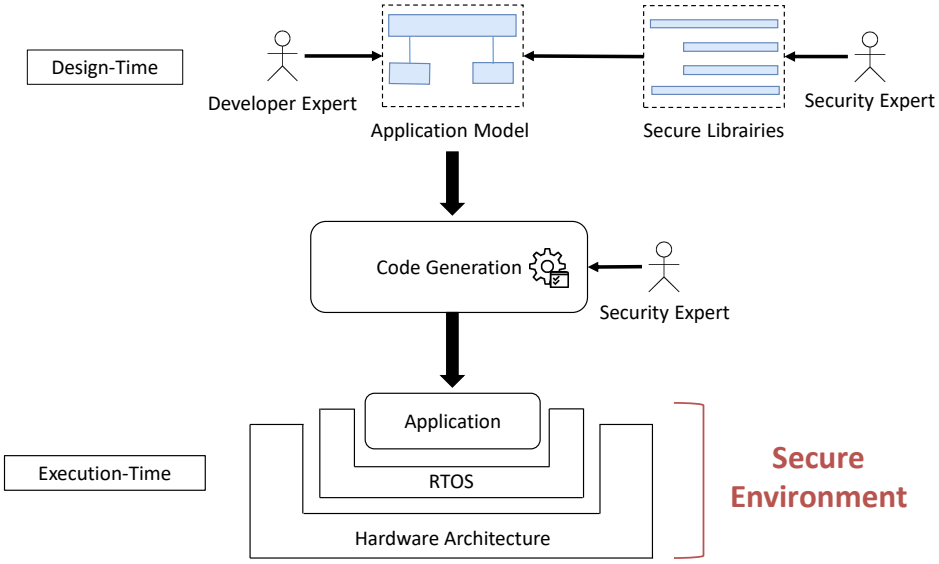

Figure 4.1: TrustFlow top-down approach

In Figure 4.1 the top-down approach is divided into two phases. First, at execution-time, the application is protected in a secure environment that provides hardware-based data-flow integrity. The secure environment assumes that the executed low-level program may contain any kind of memory bug that can be turned into an exploit. Taking this into account, the hardware extension tracks the integrity of both control-flow and non-control sensitive data. If sensitive data is maliciously corrupted, the safe environment detects it, logs it, and heals the corrupted data without impacting the users' safety.

To be efficient, the secure environment must identify sensitive data-flow. In the top-down approach, the sensitive data-flow is identified at design-time thanks to a secure toolchain and secure libraries. This software support is composed of specific libraries and a code generator used at design-time (see Figure 4.1). These libraries allow developers and security analysts to mark sensitive data in their code using annotations. Then, during code-generation, the toolchain produces a code where control-flow information is protected as well as sensitive data annotated during the design-time. As a result, software support allows the developer to easily generate a **secure code** that fits the security features provided by the hardware secure environment with **minimal effort**.

4.2.1 A trusted environment

The secure environment aims at ensuring sensitive data integrity at execution-time. To guarantee sensitive data integrity, one idea is to store them in a safe, trusted, and physically isolated memory in hardware. Such a trusted memory would isolate sensitive data making them unattainable from the untrusted memory corruptable by adversaries. However, while this principle protects sensitive data from any corruption, it cannot detect a memory exploit attempt. The process of detecting an attack remains as important as the protection of sensitive data for a critical system. Indeed, the detection of an attack and its report makes it possible to be aware of threats and anticipate security updates in real-time. Furthermore, as a complement, detecting an attack helps developers of embedded system software to be aware of its failures. To perform both detection and data-flow integrity, the TrustFlow environment uses a trusted memory displayed in Figure 4.2 that handles a copy of sensitive data at run-time. While this principle can be compared to a voter for sensitive data, it allows tracing the data-flow integrity, detecting a memory violation, logging the anomaly, and finally, healing the corrupted data with a healthy one leaving the system in a safe state. Another argument in favor of duplication and detection rather than just dividing the memory spaces into two parts (one for sensitive data, one for insensitive data) is security tests. Embedded applications have far fewer software execution-time evaluation tools support than custom desktop applications [25]. Memory safety issues are much more silent in embedded systems and more difficult to track [151]. By splitting the memory space, this issue may not be improved. Although the security level may be the same, the detection of silent memory corruption may be much harder.

To manage interaction with its trusted memory, TrustFlow provides both a custom load and a custom store operation. The custom store operation of sensitive data is similar to the store of non-sensitive data. The only difference is that the value of the sensitive data is duplicated in the trusted memory displayed in Figure 4.2. Of course, when sensitive data is updated using a custom store operation in the current memory the same operation is performed in the trusted memory. The custom load operation provided by TrustFlow works as similarly as the regular load of non-sensitive data. When used, the custom load fetches both sensitive data from the current memory and sensitive data from the trusted memory. Then, it verifies its integrity.

In the case of data-flow violation, the violated data in the regular memory differs from

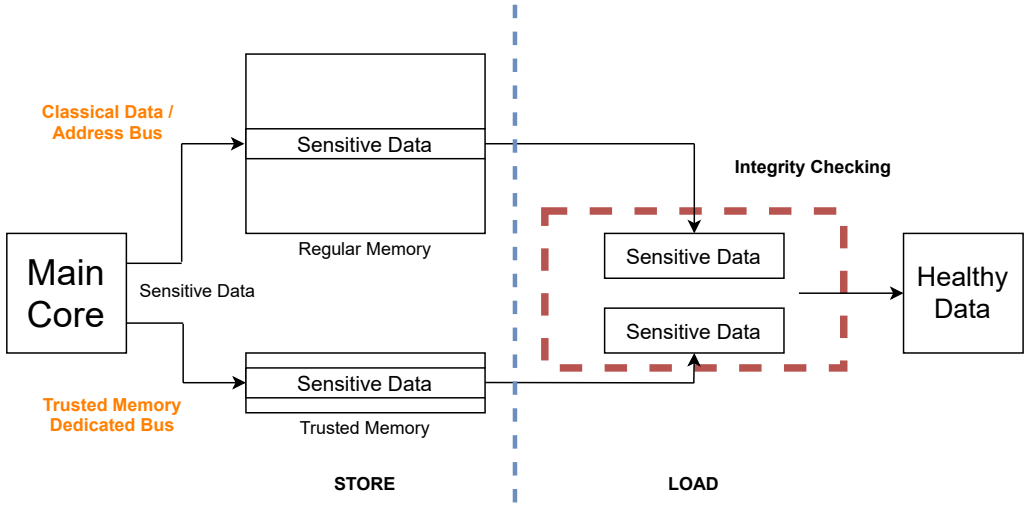

Figure 4.2: TrustFlow Trusted Environment Concept

the one in the trusted memory. TrustFlow logs this issue, raises an exception, and replaces the corrupted data with the healthy one. Considering a classic load and store processor architecture the implementation of such hardware support for data-flow integrity induces the following challenges:

- **ISA extension:** The basic instruction set of the target load and store processor must be extended to support the two new custom instructions;
- **The trusted memory:** The trusted memory should be integrated within an existing load and store processor architecture. The latter interacts with the main core;
  - **Trusted memory size:** Obviously, the size of the trusted memory restricts the number of sensitive data that can be protected at the same time. Also, the trusted memory needs to be aware of the liveliness of the stored sensitive data. Freed data may take up valuable space in the trusted memory;
  - **Real-time constraints:** The interactions with the trusted memory and the integrity checks should be fast;
- **Main core logic:** The target processor should perform sensitive data verification without inducing unwanted execution-time overhead. The processor should heal corrupted data on the fly without inducing any hazards.

4.2.2 A secure toolchain

The previously presented secure environment requires high-level software support to protect sensitive data. The environment only secures data that uses custom instructions. This means correct instructions should be used **only** and **always** to manipulate sensitive data. If this property is guaranteed, the assembly code may ensure instruction-level separation between sensitive data and non-sensitive data controllable by adversaries.

In the scope of this work, TrustFlow considers that every in-memory code-pointer can be corrupted by an adversary. This includes backward-edge control-flow information such as stack return address and indirect forward-edges such as function pointer. To remind, function pointers are rarely used in embedded system programming and discouraged by [46] but are still included in the threat model.

Obviously, after code generation, it would be impractical to have developers manually instrument the sensitive data-flow with custom instruction. Thus, for practicality and accuracy, the TrustFlow toolchain automates the instruction selection. It generates an assembly code where each sensitive data access is performed with the custom instructions.

To provide a degree of freedom, the toolchain provides several security levels. Such levels aim at protecting either backward-edges, backward-edges and forward-edges, forward-edges, or everything including spilled registers. Besides, these security levels provided by the toolchain can be used to perform security tests at different granularities. Finally, the TrustFlow toolchain provides a secure library with inline assembly routines that allow developers to manipulate custom sensitive data that are not considered by the threat model.

Unfortunately, the trusted memory size is limited. The amount of the trusted memory used by an application is directly proportional to the amount of sensitive data. Therefore, an excess of sensitive data in an application can lead to an overflow of the trusted memory. To anticipate this issue, the TrustFlow toolchain suite proposes a trusted memory static analyzer. This tool statically estimates the trusted memory cost of a program before any implementation on the real hardware. As a result, the static analyzer informs developers early in the design stage about the cost of the security as well as the validity of its application regarding the hardware environment. Regarding critical devices, this tool allows to formally prove that an application fits the hardware security support. The TrustFlow toolchain workflow is displayed in Figure 4.3.

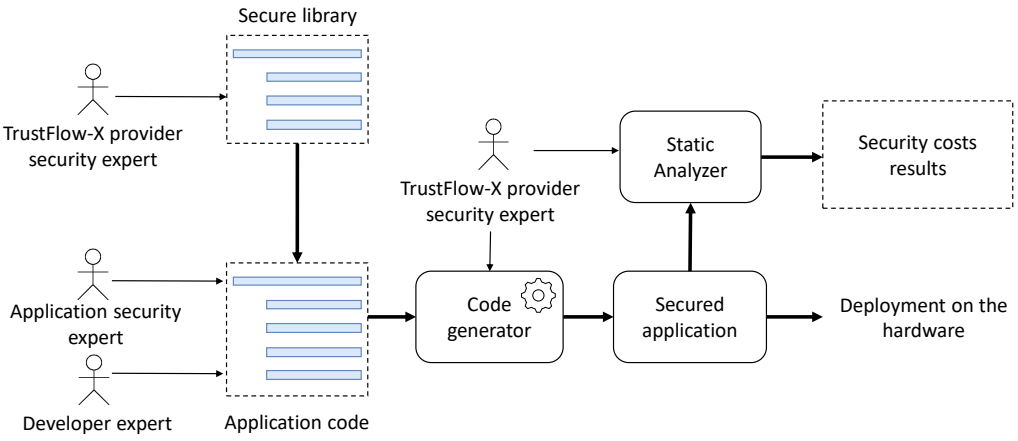

Figure 4.3: TrustFlow Toolchain Concept.

First, a secure library is included in the development phase. This library allows developers and the security expert team to mark custom sensitive data. Then, the code passes the code generator provided by the TrustFlow security experts. After this step, sensitive

code-pointers are protected. At this stage, the application is ready to be deployed, but, the latter can also be passed through the static analyzer to be verified.

A toolchain that respects the traditional software development flow can only be implemented with the help of a compiler framework infrastructure. However, as for the secure environment, it raises several challenges listed below:

- **ISA extension:** The targeted compiler should support the custom load and store instructions introduced by the hardware environment.
- **Data selection:** The targeted compiler should automatically select the correct load and store instruction to achieve instruction-level separation. It should also provide multiple security levels.
- **Manual sensitive data annotation:** The toolchain suite must provide a library that enables developers to annotate custom sensitive data at the source code level. Then, the targeted compiler should interpret these annotations and make sure that these data are always accessed using custom instructions.
- **Static analyzer:** Given a binary, the static analyzer should be able to compute the trusted memory footprint of an application as accurately as possible.

4.3 Implementation

The TrustFlow framework is implemented within the RISC-V [90] instruction set architecture and the Clang/LLVM compiler infrastructure [23]. More precisely, the secure hardware environment extends the RISC-V Rocket-Chip generator in Chisel [152]. The TrustFlow hardware extension is a trusted memory driven by a modified pipeline. Besides, TrustFlow integrates a custom hardware correction algorithm that heals systems from detected memory-based attacks. At the software level, an extended version of the LLVM compiler is provided. This compiler provides instruction-level separation support for sensitive data access. During code generation, the compiler identifies each sensitive data. At instruction selection time, the compiler uses the custom instructions to secure sensitive data-flow (code pointers). The critical application code can be analyzed using the custom LLVM based static analyzer. The latter determines the maximum amount of live sensitive data used by an application.

4.3.1 Environment Implementation

At the hardware level, TrustFlow integrates two major components: a trusted translation lookaside buffer (TLB), and an extended processor pipeline with two custom instructions. The TLB handles sensitive data and is used as a fast lookup memory. On the other hand, the processor pipeline is directly connected to the TLB to quickly store and retrieve sensitive data. The whole design is a 5-stage, in-order, 32bit RISC-V processor with a 16 kB data cache and a 4kB instruction cache. TrustFlow extends the RV32I basic instruction set with two new secure instructions given below:

- ***sws rs, imm(rd):*** stores a 32-bit word and duplicates the sensitive data in the trusted memory (secure store instruction)
- ***lws rd, imm(rs):*** loads a 32-bit word and checks the integrity of the data using the trusted memory (secure load instruction).

The two new instructions introduced by the TrustFlow extension inherit the properties of the classic RISC-V load (*"lw"*) and store (*"sw"*) instructions.

4.3.1.1 Translation Lookaside Buffer Design

Given the real-time requirements of life-critical embedded systems, TrustFlow's trusted memory is a fast Translation Lookaside Buffer (TLB) implemented as content-addressable memory (CAM) [153]. This TLB can perform lookups in a single clock cycle. Figure 4.4 displays how custom instructions interact with the TLB. The trusted TLB is implemented as an associative memory [153]. The TLB search key is the sensitive data address in the regular memory, and the result is the value of the sensitive data itself.

When a valid custom store instruction stores sensitive data (see Figure 4.4), the sensitive data address is committed as a key (V(0) in Figure 4.4) in TLB, and the data itself is stored both in the regular memory and as a value in the TLB (V(1) in Figure 4.4). The

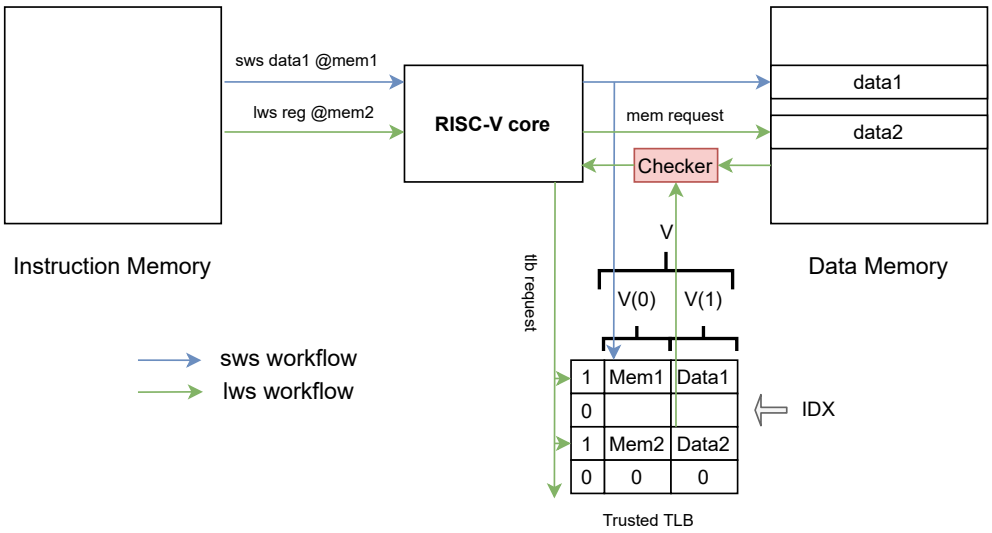

Figure 4.4: Trusted Memory Custom Load and Store.

sensitive data value/address vector is always placed in a free entry of the TLB pointed by IDX. Conversely, if the key V(0) is already present in the TLB the corresponding value V(1) is updated.

With a custom load (see Figure 4.4), the TLB uses the sensitive data address as a key to retrieve the matching data value. Then, both data from the regular memory and data from the TLB are compared by the checker. In case of mismatch, TrustFlow replaces the corrupted data with its healthy copy coming from the trusted memory and raises a *"DFI"* exception.

Each entry in the TLB is indexed by a vector V. The IDX value always points to free entry (0) of the TLB. If IDX does not point on a valid entry in the TLB, the latter is full, raising an overflow exception.

One can see that freed data may occupy precious space in the trusted TLB. According to the assumptions in section 1.1.3, the only freed data in the TLB are part of the freed stack frame. In other words, all data located below the stack pointer. Thus, when the stack pointer (sp) increases, all data that belong under it are considered as freed. To erase freed data, the trusted TLB acts as a garbage collector. Every time the stack pointer frees memory, all entries in the TLB that belongs to the freed stack slot are erased. By defining V as a vector stored in the trusted TLB, the latter follows the simple Algorithm 1 to eliminate freed data.

Algorithm 1: Hardware Garbage Collector

```
foreach V ∈ TLB do
    if V(0) ∈ [Stack_base; Stack_bound] then
        if V(0) < SP_Value then
            V_valid = 0
        end
    end
end
```

4.3.1.2 Processor Pipeline changes

To remind, the trusted memory is driven by two custom instructions. When used, these custom instructions pass the 5 stages of the processor pipeline as regular instructions. Before digging into architectural details, here is a brief outline of the 5 stages of the RISC-V processor pipeline displayed in Figure 4.5:

- **Instruction Fetch:** The program counter register (PC) points on the instruction to fetch from the instruction memory and put it in the instruction register.
- **Instruction Decode:** The previously fetched instruction is then decoded into control signals. The operands of the instruction are moved to immediate fields or general-purpose registers.
- **Instruction Execute:** The instruction is executed. Usually, at this stage, either an arithmetic register/register operation or register/immediate operation is performed by the arithmetic-logic unit.
- **Memory Access:** If the instruction is a load or a store, the memory is accessed at this stage. Otherwise, the instruction is simply forwarded to the next stage. Usually, for a load or store instruction, the accessed memory address comes from the arithmetic-logic unit and the data to store from "rs2".
- **Write Back:** The results of an operation performed by an instruction are written to a destination register within the register file.

The custom load and store instructions trigger the TrustFlow extension during the memory access stage. At this stage, TrustFlow processes sensitive data address pairs needed to be stored/checked in/from the trusted TLB.

TrustFlow is aware of the validity of the instruction flow in the pipeline and replicates the regular memory access to the trusted TLB. For instance, each invalid custom "load" with the normal memory due to a pipeline stall (cache miss) is also canceled with the TLB inhibiting the checker. Likewise, a canceled "custom" store instruction is never committed to the trusted TLB. TrustFlow performs the data integrity checks during the write back stage of the pipeline. The checker is triggered by a custom load and verifies that data fetched from the data cache do not differ from the trusted TLB. In the case of discrepancy, a data-flow violation is logged and the faulty data is replaced on-the-fly by the checker.

A stage by stage simplified pipeline outline of TrustFlow is given below, the blue components in Figure 4.5 represent the important TrustFlow modules.

Healing data on the fly by replacing corrupted one is an important feature of TrustFlow. Another solution would be to revert the execution flow to a previous state and re-execute and entire section. However, it would allow an attacker to replay his attack. Also, memory exploits can be seen as a non-deterministic disturbance. They are unpredictable and therefore not appreciated by critical deterministic systems. As a result, reverting the execution flow due to a non-deterministic attack may be inconsistent with deterministic

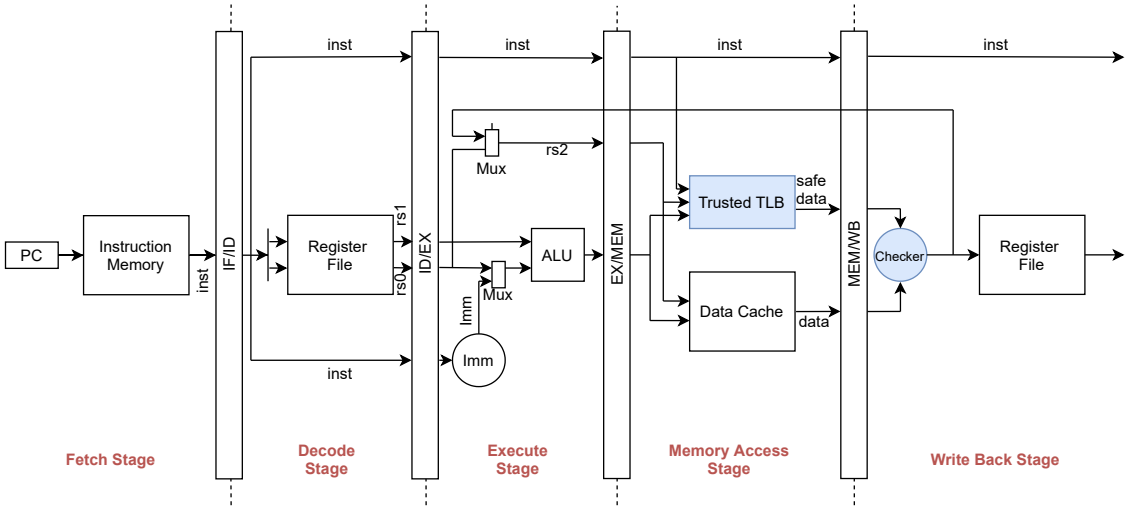

Figure 4.5: Simplified RISC-V Core with the TrustFlow Extension.

critical systems. By healing data on the fly, TrustFlow does not interrupt the execution flow maintaining determinism. However, it simply logs data-flow issues that can be tackled by a real-time, deterministic developer's custom task afterward. For in-depth details about the TrustFlow pipeline, we refer the reader to Annex A.

4.3.2 Toolchain implementation

To ensure instruction-level separation, the TrustFlow toolchain extends the Clang/LLVM [23] compiler with additional passes. With strong compiler support, the toolchain simplifies the integration and accuracy of security. Also, the second goal of the toolchain is to deliver verification options regarding the amount of trusted memory consumed by an application.

4.3.2.1 Required compiler changes

The custom instructions introduced in the RISC-V ISA resulted in an extension of the RISC-V LLVM backend.

As pointed out earlier, the goal of the compiler is to protect both indirect backward and forward-edges preventing memory exploits. By design, the compiler spills the return address (backward-edge control-flow information), and some registers of a calling function in the prologue of the called function. Also, when a called function returns to its caller, the return address and the spilled registers are restored during the epilogue. By targeting the calling convention and more precisely the in-memory return address stack an attacker can divert an indirect backward-edge. To address this issue using the TrustFlow environment, both the prologue and the epilogue should use custom instructions when spilling/restoring the caller register and the return address in a stack frame.

By leveraging an in-memory bug an attacker may also target indirect forward-edges such as function pointers. To protect indirect forward-edges, function pointers should be manipulated with custom instructions. However, protecting every in-memory function

pointer is less trivial as for the backward-edges. Indeed, the data-flow of function pointers must be analyzed to determine when and where they are spilled and restored in/from the memory. Each time it happens, it must be achieved with a custom instruction. To perform such protection, the compiler must analyze and determine each function pointers' flow. Then, during instruction selection, each load/store involving a function pointer should use a custom instruction. Finally, to determine the trusted TLB footprint of an application, the compiler integrates a static analyzer that crosses the application code and determines the worst-case trusted TLB usage.

4.3.2.2 Introduced passes workflow

Multiple passes are introduced in the LLVM compilation pipeline to achieve fine-grained data-flow integrity. Figure 1.6 displays a simplified representation of the LLVM backend pipeline with the TrustFlow introduced passes.

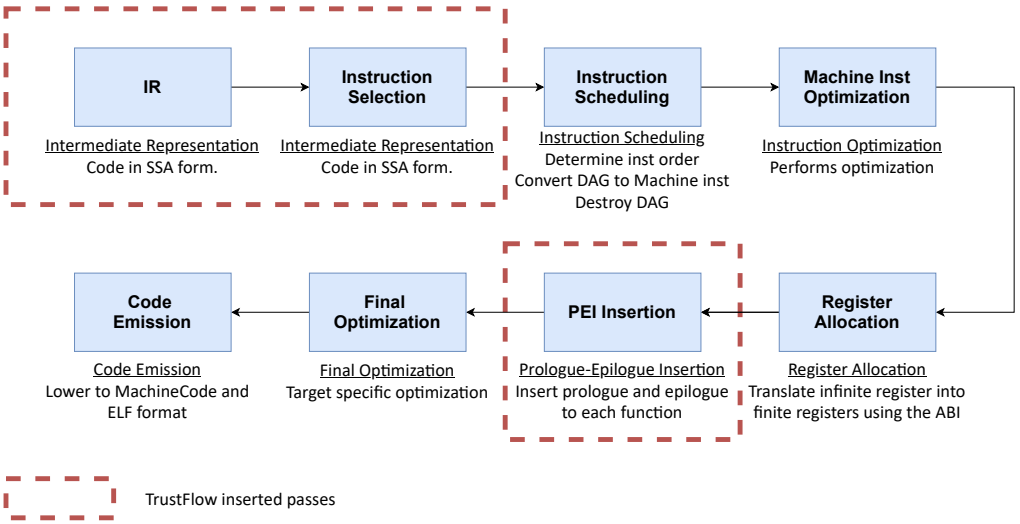

Figure 4.6: LLVM Backend Compilation Pipeline.

In Figure 4.6, the passes dealing with the forward-edge control-flow integrity happen earlier than the backward-edge control-flow integrity passes in the compilation pipeline.

The first function pointers pass occurs on the Intermediate Representation (IR). At this stage, the code is strongly typed and still in the Single Static Assignment (SSA) form. Using the LLVM IR, the pass parses each instruction and their operand type to determine whether they handle a function pointer. If an SSA load/store instruction manipulates a function pointer, the pass annotates it with LLVM metadata. Before instruction selection, the compiler translates each SSA instruction into Selection Dag Node (SDNode). The SelectionDAG pass is extended to handle the SSA metadata to the SDNode structure. During instruction selection, the SDNodes are converted into machine SDNode which handles target instructions. A custom post instruction selection DAG pass operates at this level and translates both load and store machine DAG that handle metadata into custom load and store machine DAG.

The backward-edge control-flow integrity is performed after the register allocation

pass during prologue-epilogue insertion. To protect backward-edges, the prologue/epilogue insertion pass is modified to spill/restore function return address with custom instructions. Also, at this stage, the improved compiler offers an option to secure every spilled register with custom instructions. For more details about the prologue-epilogue insertion, we refer the reader to Annex B.

4.3.2.3 Static analyzer

After secure code generation, developers have no idea about the cost of their program in the trusted TLB. They can perform tests on the hardware to determine whether the trusted memory overflows or not. This approach is unpractical and time-consuming. To safely use the TLB, developers require a practical tool that ensures that their application fits the TLB according to the security level.

To do so, the TrustFlow comes with a static verification compiler extension that estimates the worst-case maximum amount of trusted memory used by an application. This pass is implemented as a "Machine Module" pass within the RISC-V LLVM backend. Also, the pass is only compatible with full Link-Time Optimization (full-LTO). During compilation, the full-LTO optimization merges separated modules into a single monolithic Intermediate Representation module. This monolithic module is then passed to the LLVM backend allowing the static analyzer pass to see the program as a whole and perform inter-procedural analysis. The static analyzer pass traverses each function evaluating its trusted memory costs. Figure 4.7 displays on the left various compiled functions with their trusted memory costs. For instance, the vertex (function) 1 cost 3 entries in the TLB and calls vertex 2, 3, and 4.

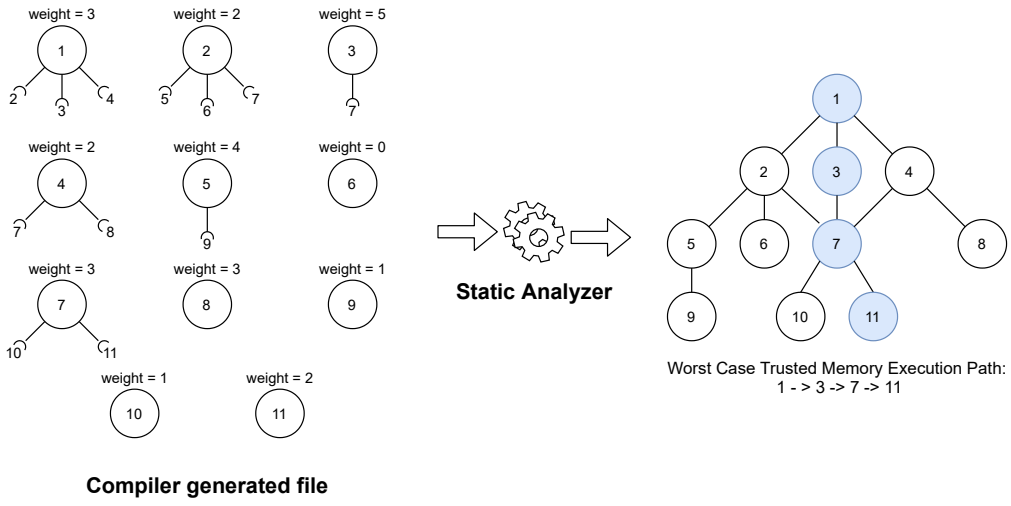

Figure 4.7: Static Analyzer.

Then, the static analyzer pass reconstructs the entire program control-flow graph. This operation results in the construction of a weighted directed graph. More precisely, as we forbid recursion in our approach [46], this constructed directed graph is acyclic (DAG).

Unfortunately, static control-flow graphs are often imprecise due to indirect forward-edges targets. To generate a fine-grained control-flow graph the static analyzer requires

little human interaction. To do so, the Clang front end of the compiler is improved and provides a high-level annotation that can be used by developers to mark functions that handle indirect control-flow transfers with the name of their potential target(s). With this support, the compiler can construct a fine-grained weighted directed acyclic graph.

To determine the maximum usage of trusted TLB by an application, the static analyzer determines the deepest path in the constructed weighted DAG. As verification, the static analyzer makes sure that the generated call graph does not contain any cycles. For space reasons, we do not detail the cycle detection algorithm performed by the static analyzer. However, the cycle detection is based on a depth-first search algorithm that colors visited vertices.

The deepest path problem for a weighted directed acyclic graph is known to be a linear problem. To solve it, the static analyzer uses a classical topological sorting algorithm on the DAG to sort all the vertices. This algorithm performs a linear ordering of all vertices such that for every direct edge between two vertices A and B, vertex A comes before vertex B in the ordering.

Once the vertices are sorted in a stack S, the static analyzer initializes the distance of every vertex as negative infinity (NINF) and the distance of the root vertex (application entry point) as 0. Then, the vertices are processed one by one by popping the stack. For every vertex being processed, the static analyzer updates the distances of its child(s) using the distance of the one being processed. The entire algorithm is summarized in Algorithm 2 snippet.

Algorithm 2: Deepest Path Algorithm

```
foreach vertex ∈ DAG do
    dist[vertex] = NINF
end
dist[root] = 0 ;
while S ≠ ∅ do
    vertex = S.pop();
    if dist[vertex] ≠ NINF then
        foreach child ∈ vertex do
            if dist[child] < dist[vertex] + child.getWeight() then
                dist[child] = dist[vertex] + child.getWeight();
            end
        end
    end
end
deepth = 0;
foreach d ∈ dist do
    if d > deepth then
        deepth = d;
    end
end
Time Complexity O(V+E)
```

As a result, the deepest path corresponding to the worst-case trusted memory usage

is displayed. Another “feature” algorithm implemented in the static analyzer prints the longest path for the developer. This critical path can then be optimized to reduce the trusted TLB footprint of its application.

4.4 Evaluation

This section assesses the TrustFlow framework. The toolchain requires about 1500 lines of C++ code to be added to the Clang/LLVM compiler infrastructure. Also, the TrustFlow environment is synthesized on a Xilinx ARTY-35T FPGA and requires 400 lines of Chisel code to be added in the RISC-V RocketChip generator. To validate the security of the concept, TrustFlow is confronted with the RIPE benchmark suite [85]. Then, we evaluate the costs of the TrustFlow environment in hardware. This includes a trusted memory size benchmark regarding different life-critical medical applications, execution-time overhead induced by the TrustFlow extension, and the additional LUTs and Flip-Flops required by TrustFlow. Finally, the main goal of this section is to demonstrate that an acceptable level of security can be achieved on a wide range of programs with a very small trusted TLB and at the lowest costs.

4.4.1 Security evaluation

The RIPE benchmark suite [85] provides different attack patterns to test the coverage of countermeasures against memory attacks. These attacks use five different dimensions: *the location*, *the target code pointer*, *the overflow technique*, *the attack code*, and *the function abused*. Following the assumptions [46], this evaluation considers out of scope all attacks involving the dynamic memory allocation from the benchmark suite. Also, as TrustFlow detects sensitive data corruption at execution-time, the evaluation is focused on the targeted code pointers regarding its location and the overflow technique. In the evaluation, the benchmark is compiled with 3 different options: backward-edge protection (B), backward-edge and spilled registers (BS) protection and finally, backward-edges spilled registers, and forward-edges (BFS) protection. The results are summarized in Table 4.1.

Table 4.1: TrustFlow Security Benchmark.

Protection	**Sensitive data**											
			function pointer			**longjmp**						
	return address	base pointer	stack	heap	data/bss	stack variable	heap	data/bss	data stack	spilled stack data	data data/bss	
B	D	X	X	O.S	X	X	O.S	X	X	X	X	
BS	D	D	X	O.S	X	D	O.S	D	X	D	X	
BFS	D	D	D	O.S	X	D	O.S	D	X	D	X	

X=undetected, D=Detected and healed by TrustFlow, O.S=Out of the scope of the TrusFlow-X framework.

After compilation with full protection and execution on the environment, TrustFlow

protects and detects any attack from the benchmark suite that targets: function return addresses, stack function pointers (both in the stack or pass as a parameter), data section function pointers and BSS function pointers. Besides, the TrustFlow framework is also able to protect and detect any attacks that target function pointers residing close to a buffer within a structure located either in the stack, the BSS, or the data section. However, these function pointers should only be used with one level of dereferencement. Of course, following the threat model in section 4.2, none of the attacks corrupting a target code pointer in dynamic memory is detected. From Table 4.1 it also appears that the security difference between "B" and "BS" is less. Only two additional direct attacks and indirect attacks are detected by the "BS" option.

Several data-oriented attacks are also implemented in the benchmark suite. At this moment, the improved compiler is not able to automatically protect security-critical data such as numerical values used for a branch condition, or token used for privileges. However, the TrustFlow environment provides the necessary support to ensure its integrity. Finally, any data leakage that does not corrupt sensitive data is not detected by TrustFlow.

4.4.2 Environment evaluation

4.4.2.1 Trusted memory benchmark

While TrustFlow is capable of ensuring fine-grained CFI through fine-grained DFI, its security level is ensured by the trusted memory that has a fixed size. To prove that TrustFlow can handle a wide range of life-critical applications respecting our software assumptions [46] (see section 2.2) with a reasonable fixed TLB size, this section assesses the amount of trusted memory consumed by various medical applications that follow our software assumptions:

1. Inverse Radon Transform from the HERMIT benchmark suite [154]. This application is commonly used in portable ultrasound and MRI devices. It allows reconstructing the output image from a 159KB 2D sinogram input data file for visualizing abnormal openings in the body.

2. SecPump [149] a fully open-source wireless insulin pump. The SecPump project aims at serving as a medical platform for security evaluation.

3. The open-source syringe pump project [88]. Open-syringe proposes an inexpensive Arduino based medical for research purposes. This project is easily portable on RISC-V and highly representative of a real medical device.

4. ARM Mbed TLS library [155]. While not entirely dedicated to medical devices ARM Mbed TLS is a cryptographic library reference for embedded systems.

The applications are benchmarked using the static analyzer. The results outputted by the static analyzer represent the worst-case trusted TLB usage according to a path taken within an application. The static analyzer does not guarantee the coverage of the longest

path and if it is frequently taken. The trusted memory benchmark is achieved with the following assumptions:

- The applications are compiled and evaluated without recompiling external libraries;
- Five data-flow integrity levels are applied; backward-edges ("ret"), forward-edges ("pointer"), backward-edges and spill registers ("spill"), backward-edges and forward-edges ("ret+pointer"), and finally, forward-edges and backward-edges with spilled registers ("spill+pointer");
- The compilation options recommended for the evaluated applications are kept as provided in their dedicated Makefile, we only changed the compiler optimization level to measure the impact on the trusted memory. Three compiler options are considered: "O0, O2, Os"

The results of the benchmark are displayed in Figure 4.8.

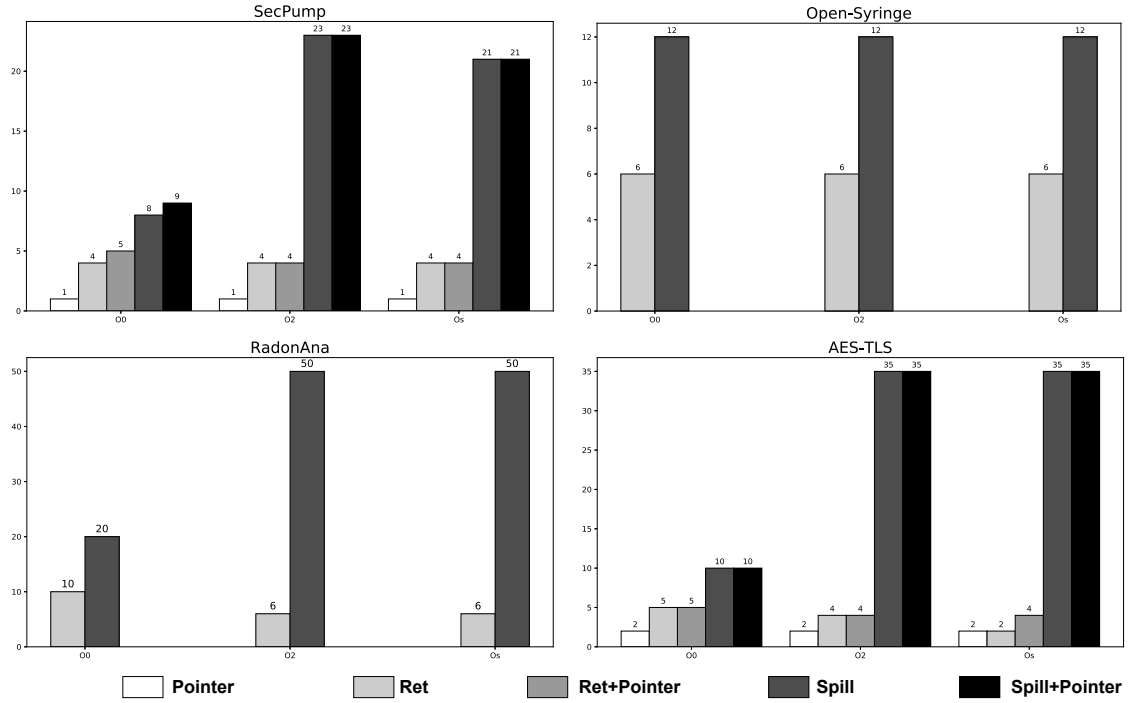

Figure 4.8: Trusted Memory Footprint.

The Y-axis is the maximum entries required in the TLB. The X-axis is the compiler optimization.

According to the results, only a reasonable amount of trusted memory is used by all evaluated critical applications. The "Inverse Radon Transform" application consumes the most number of entries in the trusted TLB. More precisely, 50 entries with full protection for an application totaling about 7000 lines of code. However, 50 entries of 64 bits (8 bytes) only represent 400 bytes of a trusted memory, which is less than a kB. Securing spilled registers highly increases the consumption of trusted memory. For each function call, every register spilled in memory is saved by the compiler using a custom instruction.

Besides, a better compiler optimization option increases the number of spilled registers and, consequently, the consumed trusted memory. Indeed, the compiler tends to favor the use of registers per function instead of memory accesses to improve the execution speed.

Comparing with the previous security evaluation, the cost of protecting spilled registers is very high for a security level very little different from the backward-edge protection. Depending on the threat model and system constraints, securing the spilled registers may not be worth the costs. Conversely, according to the previous security evaluation, protecting function pointers (forward-edges) render a wide range of attacks ineffective. According to the measurements, the effect of function pointer protection is ineffective on the trusted memory. Function pointers are rarely used in critical embedded systems [38] (even never used, see Figure 4.8, RadonAna, and Open-Syringe).

The results outputted by the static analyzer represent the worst-case trusted TLB usage according to a path taken within an application. To remind, the static analyzer does not guarantee the coverage of the longest path. To verify the accuracy of the static analyzer and to validate its measurements the static measurement of RadonAna is compared with a dynamic analysis using 6 various compiler optimization options. We selected RadonAna because it totals around 7000 lines of codes, 130 vertices, a large corpus of input that covers many different paths in the application, and because the application seems to consume the most memory.

An interesting point, more related to section 4.4.1 is that TrustFlow's static analyzer detected several vulnerabilities in the HERMIT benchmark suite. Some of these vulnerabilities included infinite call sequences causing deadlocks in the benchmarked application.

The results of the static analysis versus the dynamic analysis are displayed in Figure 4.9. The RadonAna application is executed 20 times with different sinogram data files. The worst dynamic case is displayed in Figure 4.9.

The dynamic evaluation comparison in Figure 4.9 confirms the trustworthiness of the static analyzer. It also confirms that in real execution the worst-case evaluation given by the static analyzer is not always attained. Finally, the dynamic benchmark reinforces the static benchmark and confirms the hypothesis that a small trusted memory with a hardware garbage collector can secure reasonable embedded applications.

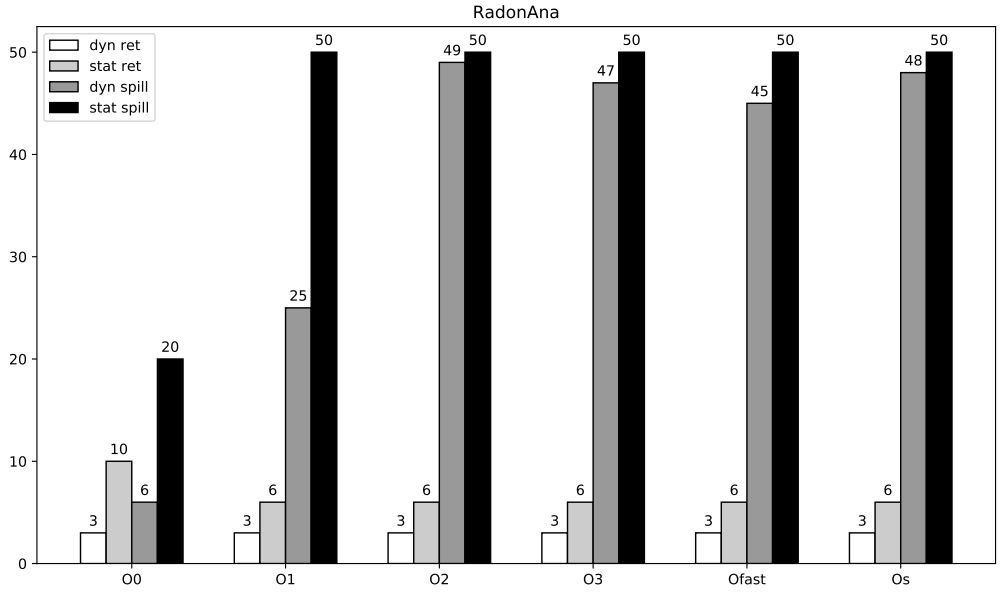

Figure 4.9: Dynamic versus Static Evaluation.

The Y-axis is the maximum entries required in the TLB. The X-axis is the compiler optimization.

4.4.2.2 Execution time overhead

Since TrustFlow is dedicated to bare-metal/RTOS embedded systems, it is not relevant to port the well-known benchmark suite SPEC2006 CPU [107]. Moreover, TrustFlow cannot support the Linux Kernel. So this assessment turned to the CoreMark benchmark suite [150]. CoreMark contains 4 algorithms such as list processing (find and sort), matrix manipulation (common matrix operations), state machine, and CRC (cyclic redundancy check) that are designed to run on devices ranging from 8-bit microcontrollers up to 64-bit microprocessors. Thus, CoreMark is easily portable on bare-metal architectures such as TrustFlow.

To measure the execution-time overhead induced by the TrustFlow environment, the processor is synthesized on a Xilinx ARTY-35T FPGA with a fixed TLB size of 384 bytes. The CoreMark benchmark suite is compiled using the TrustFlow's Clang/LLVM extended compiler using optimization level 3 with 5000 iterations. The measurements revealed a processor execution ticks overhead of 0.0087% for the backward-edge protection and 0.021% for the spill protection. As a result, the execution-time overhead induced by TrustFlow is negligible. This can be easily explained by the fact that all operations performed by TrustFlow in hardware are carried out in parallel to the main pipeline. Thus, any custom instruction triggers an independent hardware extension that only performs a simple check operation at the write-back stage.

Also, it should be mentioned that TrustFlow does not induce any size overhead on the final binary of the benchmark. Custom load and store instruction just replace the classic load and store instruction for sensitive data.

4.4.2.3 Hardware costs

The TrustFlow hardware extension overhead can be divided into two distinct parts: (1) a trusted external fast memory with a garbage collector and (2) additional instructions with an in-pipeline checker. Like any hardware implementation, the TrustFlow core consumes additional LUTs and Flip-Flops.

The additional hardware cost due to the two new instructions is very low compared to the hardware cost induced by the trusted memory. The reference RISC-V RocketChip processor [152] consumes a total of 16671 LUTs and 9905 Flip-Flops. The addition of two new instructions in the core, the checker, and the digital logic that ensures memory management increases the number of LUTs (16789) by 1.03% and the number of Flip-Flops (10020) by 1.16%. However, when adding a fixed trusted memory of 384 bytes (48 entries), the design consumes a total of 18643 LUTs (11% increase) and 12099 Flip-Flops (22% increase).

In terms of performance, the two instructions are similar to the load and store instruction. Both custom instructions manage the trusted memory in parallel with the main memory. Similar performances are measured between the custom load and store instruction and the regular ones. The core modifications induced by the addition of these instructions have mostly no effect on the timing performance of the core. TrustFlow is able to broadly reach the same running frequency on the embedding core. However, a negligible timing delay may impact the system performances in case of data discrepancy due to the checker and the data correction.

4.5 Discussion

It is fair to affirm that TrustFlow is first designed to secure critical bare-metal applications. In theory, TrustFlow can boot a lightweight real-time operating system such as FreeRTOS. Indeed, the TrustFlow's trusted memory is a table, mapping address and sensitive data pairs that are compliant with multi-threading. Unfortunately, a simple small trusted memory such as the one exposed in this Chapter is not enough to store all the critical data of the tasks shared by an application on a single processor. Besides, the current hardware implementation of the TrustFlow memory garbage collector does not support multiple stacks. The garbage collector invalidates part of the trusted memory data depending on the value of the stack pointer. Unfortunately, with a multi-tasks application, the real-time operating system maintains several stacks inducing stack pointer changes at each context switch. Thus, the current hardware garbage collector that follows algorithm 1 may evict valid sensitive data in the trusted memory if the stack of the new active task is at a higher address than the stack of the suspended task.

To address multitasking, one solution would be to map trusted memory registers. These registers may allow a real-time system to configure the garbage collector with the address ranges corresponding to the different stacks in memory. The global idea would be to make the garbage collector aware of the different stacks. However, this improvement may introduce new limitations. Indeed, the number of trusted memory registers will limit the number of tasks protected by trusted memory. Although the concept lends itself to it, it would induce high hardware costs to keep the same performances as TrustFlow while protecting all the sensitive data per process.

Another way to handle multiple tasks would be to save and restore the trusted memory per process. Taking the FreeRTOS [29] real-time operating system as an example, the task control block (TCB) structure can be extended such that it handles valid stack entries from the trusted TLB. Then, during a context switch, the scheduler of the operating system can, therefore, save the state of the task trusted memory in the extended TCB, and restores the following running task trusted memory. Of course, this concept should be coupled with software isolation (SFI) such that the TCB of suspended tasks cannot be corrupted by a memory violation in a running task.

Finally, the TrustFlow hardware trusted memory is size limited. This work provides an accurate static analyzer such that developers can easily optimize their application to fit the trusted memory. During the evaluation of the concept, and like other work related to shadow stacks [78, 82, 156] we did not face any overflow issue with a fixed trusted memory. However, considering manufacturing TrustFlow for critical systems at a large scale, giving trusted memory depth metrics is not enough. Indeed, the overflow issues related to the trusted memory must be supported. One way to deal with it would be to raise an interruption when the trusted TLB is full and to transfer the highest stack data to the current memory or an extended TCB for a real-time system. However, once again this mechanism should be coupled with SFI to protect the transferred data.

At the software level, TrustFlow ensures control-flow integrity through sensitive data-

flow integrity using compilation. Unfortunately, any third-party libraries that are not compiled with the TrustFlow Clang/LLVM extension are unprotected. Thus, these third-party libraries constitute new attack surfaces not covered by the protection. To deal with this case, the TrustFlow framework can be associated with a patching application. The latter could easily disassemble third-party libraries, identify the load and store instructions that manipulate sensitive data and patches them using custom instruction. However, this process is less precise than a regulation compilation workflow. It could introduce hazards in the assembler. Pre-compiled application codes are not trivial to analyze. Some of them are obfuscated, making static analysis very difficult, requiring error-prone human interaction.

Regarding function pointers protection, the current Clang/LLVM extension handles one level of dereferencing. Thus, in the case of function pointers that are dereferenced across multiple structures, the compiler is not able to generate the correct instructions. Although this goes against the assumptions made in section 1.1.3, and the safety-critical code practices proposed by [38, 46], this issue requires manual code instrumentation by the developer at the assembler level.

Finally, at present, the TrustFlow compiler extension cannot protect the induction variables. These variables are usually increased or decreased within loops and are the targets of Data-Oriented Programming attacks [65]. We are currently working on a pass similar to the function pointer identification pass to protect such variables.

4.6 Comparison with related work

In comparison with other memory safety protection work, TrustFlow is an enhanced instruction driven shadow stack [157] that can ensure fine-grained CFI protecting sensitive data-flow. In this section, we discuss a coherent selection of well-known hardware-software co-design CFI protections and their limitations with TrustFlow.

HCFI [78] is an extension of SPARC architecture that enforces CFI. HCFI enforces fine-grained backward-edge CFI thanks to a hardware shadow stack that stores the call stack. As TrustFlow, this policy provides an accurate security level but can only guard a limited call depth. Concerning forward-edge CFI, HCFI enforces a policy label thanks to two additional instructions. Although restricted, HCFI's indirect forward-edges can still target multiple locations making the overall protection a little coarse-grained. In comparison, TrustFlow enforces fine-grained forward-edge integrity. Both HCFI and TrustFlow are limited by the size of the trusted memory. However, HCFI consumes twice less memory per sensitive data than TrustFlow because it does not store the address of sensitive data as a key in its trusted memory. Besides, HCFI provides specific support for recursion. TrustFlow supports recursion but prohibits it through its assumptions. Finally, both solutions induce less than 1% execution-time overhead.

At the software level, the HCFI protection is ensured with binary instrumentation scripts. While this process is scalable for small applications, it is difficult to deploy in a production environment. The code generation is heavy and requires several steps each time a binary had to be generated. Conversely, TrustFlow provides an easy to use, accurate compiler support.

HAFIX [82] is a hardware backward-edge CFI based on labels and an isolated memory. Each time a function is called, a custom instruction is executed by the processor. This activates a unique function label in an isolated memory area. When the function terminates, another custom instruction deactivates the label from the trusted memory, and the processor checks that the target return address is part of an active function. Compared with TrustFlow, HAFIX does not enforce forward-edge CFI, leaving room function pointer hijacking attacks. In terms of backward-edge CFI, HAFIX always ensures that a function returns to an active function which is part of the call stack. Unfortunately, it does not guarantee that the function main return to its most recent caller which TrustFlow does. While execution-time overhead of both HAFIX and TrustFlow is very low, HAFIX has an advantage over TrustFlow. HAFIX overcomes the regular shadow call stack size limitation thanks to label activation/deactivation. However, HAFIX proposes special instructions just to tackle *"setjmp"* and *"longjmp"*. TrustFlow uses an innovative technique based on the stack pointer value to maintain stack coherency. Finally, HAFIX is accompanied by a compiler extension that generates a secure code using the additional instructions. Thus, the security implementation is straightforward to use for non-security experts which is consistent with the philosophy of TrustFlow.

FIXER [156] proposes a decoupled and flexible RISC-V compatible coprocessor to enforce both forward-edge and backward-edge control-flow integrity. FIXER targets hybrid

processors based on an unmodified RISC-V core interfaced with a reconfigurable FPGA core. The FPGA part of FIXER implements a shadow call stack for fine-grained backward-edge and a policy matrix to enforce forward-edge protection. More precisely the FPGA is interfaced with the RoCC interface of the RISC-V processor and interacts with it using the dedicated accelerator instructions. FIXER proposes developers to annotate each function call and returns with tags at the source code level to protect their application. These tags are then translated into instructions that interact with FIXER thanks to a binary instrumentation script that parses assembly files generated by the GCC compiler. While this technique is automatic, we argue that manually annotating an application's source code for security is not very practical. This argument is also supported by [40]. In contrast, TrustFlow automates the instruction selection at compilation time freeing the user from any binding source code annotation. Also, unlike TrustFlow, the FIXER forward-edge policy matrix is coarse-grained. Indeed, indirect forward-edges can still target valid destination sites that are invalid for a specific call. In closing, both FIXER and TrustFlow have a negligible execution-time overhead. However, the flexibility proposed by FIXER is a great asset over TrustFlow. Indeed, the trusted memory size (shadow call stack) of FIXER can be optimized on purpose and accommodate new security threats.

Finally, TrustFlow can be compared with well-known fine-grained tagged memory policies such as HDFI [131] and LowRISC [132]. They use two tag levels (IL1 and IL0) to separate sensitive data (IL1) from regular data (IL0). To trace the data-flow, HDFI [131] extends the RISC-V processor memory with an additional one-bit tag field for every word. To enforce sensitive data isolation, HDFI introduces custom store and custom load instructions as TrustFlow. These instructions are supported by a modified processor logic such that; every regular write instruction defines a data and set the tag field to zero, every special write instruction defines a sensitive data and set the tag field bit to one and every special load instruction checks that the used sensitive data has a tag set to one. Thus, a regular write that modifies sensitive data also changes its tag. When this sensitive data is re-used by a custom load instruction, the processor detects a wrong tag and raises an exception. HDFI has a major advantage over TrustFlow in terms of memory limitation. Indeed, HDFI protection is not restricted by the size of the trusted memory. However, it induces one bit overhead per double memory words which also is not negligible.

The HDFI approach carries full operating system support which TrustFlow does not. However, in terms of safety, HDFI does not provide any recovery mechanism capable of restoring corrupted data. While a custom exception handler can be implemented to re-execute the faulty code section, it would allow an attacker to replay his attack until blocking a system. Also, it may induce a non-determinism source of perturbation over the critical device. Finally, HDFI, LowRISC, and TrustFlow have full toolchain support that facilitates the integration of security into systems.

4.7 Conclusion

This Chapter proposes an innovative practical defense against memory-based exploits on life-critical medical devices. According to the state-of-the-art, it seems that existing works are not the most appropriated for medical devices. Many solution overlooks **fine-grained security**, **safety**, and other **practical aspects** that cause their solutions to be neglected by life-critical device manufacturers. Many concepts often offer **incomplete defenses** and a poor balance between **execution-time overhead** and **accuracy**. On top of that, most hardware-based countermeasures lack **software support** so that they can be used with minimal human efforts. As long as developers have to rewrite some parts of the software assembly code by hand, security may never be integrated. Besides, binary instrumentation is neither **modular** nor **practical** in a software production environment. It sometimes breaks the C programming language **structures** and forces developers to perform **security tests** at the end of a system's development phase. Finally, among all existing memory safety protections, the state of a system after detecting an attack is never addressed. For critical devices, detecting an attack is only the first step, the second step is to move the system to a **safe** state that does not impact the users' **safety**. None of the current countermeasures exhibit innovative techniques to recover from memory attacks.

To address these gaps, TrustFlow, proposes a novel **practical** framework that simplifies the integration of memory safety in critical systems while keeping a sufficient tradeoff between **security**, **safety**, and **performance**. The contribution is composed of an LLVM based secure toolchain able to generate secure code for a RISC-V based secure environment that can prevent, detect, log, and self-heal critical devices from memory exploits. The whole hardware environment is composed of an extended RISC-V processor communicating with an isolated trusted memory that handles duplicated sensitive data.

According to the assessments, TrustFlow can ensure **complete** fine-grained control-flow integrity through fine-grained data-flow integrity with less than 1% **execution-time** overhead and a **trusted memory** with less than 100 entries. The contribution comes with strong compiler support, **compliant** with all **C standards**, making the integration of security **simple** and compatible with **incremental tests/compilation**. Finally, TrustFlow provides a novel data recovery process for corrupted data that maintains execution flow determinism. In closing, TrustFlow focuses on the first stage of spatial exploitation techniques. Two popular security threats such as temporal exploits and information leakage exploits still lack enough research. These two popular exploits constitute open issues and further TrustFlow improvements.

Final Note: TrustFlow is the result of several previous works achieved on hardware shadow stacks. Its origin goes back to Speculoos [158,159], a hardware shadow stack based on the OpenRISC architecture. Speculoos is a hardware shadow stack driven by a finite state machine, dedicated to low-end single task application. Unlike TrustFlow, Speculoos does not require any software support. For space reasons, Speculoos architecture is not detailed in this thesis. However, Speculoos has been published at the *2nd International Verification and Security Workshop, IVSW 2017* [158]. Moreover, further results of the

Speculoos architecture have also been published in the *IEEE Embedded Systems Letters* journal in 2018 [159]. TrustFlow leverages the limitation of Speculoos and results in finer protection and more advanced concepts than Speculoos. The TrustFlow contribution has been published in the *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* [157]. The original software support was based on the GCC [22] compiler, then, the work moved to the Clang/LLVM compiler. The trusted memory management has also been improved with the hardware garbage collector. The final TrustFlow framework has been accepted as a journal paper at the *ACM Transactions on Embedded Computing Systems* in May 2020 [160].

5 BackGuard

Summary of the Chapter

This chapter introduces BackGuard, the second approach of this thesis. BackGuard is a compiler-based toolchain that protects embedded system systems against memory-based attacks. More specifically, BackGuard is intended for tiny constrained embedded systems generally implemented in bare-metal. Unlike TrustFlow, BackGuard is only implemented at the software level. The latter does not require any specific invasive hardware support to protect embedded applications. BackGuard is implemented within the Clang/LLVM compiler and is tested in this chapter on the ARM ISA. The control flow integrity generated by the compiler relies on a bitmap, where each set bit indicates a valid pointer destination. The memory exploits protection is enhanced by in-memory guards that prevent spatial memory vulnerabilities to spread. The efficiency of the framework is benchmarked using an STM32 NUCLEO F446RE microcontroller that implements various well-known benchmark suites. The obtained results show that the control flow integrity solution incurs an execution-time overhead of 5% on average.

Contents

5.1 Motivation

With the emergence of the Internet of Things (IoT), there is an increasing trend of connecting tiny devices to the Internet. In healthcare, this trend has resulted in exponential growths of related security and safety incidences widely discussed along with this thesis [9, 12, 15, 17, 18].

Currently and for the coming years, the IoMT's market share is growing. This creates new business opportunities for manufacturers, start-ups, and independent innovators. However, the increase of connected devices directly increases the potential attack surfaces and data theft. If the Internet of Medical Things manufacturers continues to follow the trend of ignoring security issues in favor of adding fashionable features to their products, security incidents are likely to occur shortly.

The previous Chapter of this thesis introduced TrustFlow, a security-oriented processor with software support that ensures fine-grained memory safety in life-critical systems. To remind, TrustFlow is an architecture based countermeasure that protects the security-critical dataflow. Unfortunately, TrustFlow relies on invasive hardware changes to operate. Today's IoMT manufacturers or even startups do not necessarily have the financial capacities to produce silicone with specific support for memory safety. Also, sometimes, critical medical devices are developed on microcontrollers that do not allow architectural modifications [18]. To push is further, IoMT devices are often developed on low-end microcontrollers that do not support hardware-based memory safety support [15,17,18]. Indeed, these low-end microcontrollers are often highly constrained. They do not benefit from the same security support as traditional desktop systems. For instance, less than 50% of embedded system application enforces the Data Execution Prevention (DEP) protection and less than 5% enforces both the Address Space Layout Randomization (ASLR) and the Stack Smash Protector (SSP) [54, 161]. Usually, IoMT devices run on the metal relying on exotic architectures such as ARM, AVR, MIPS, RISC-V. Each low-end microcontroller comes with fixed and specific hardware features that often have no common dedicated support for memory safety. Yet, despite these constraints, medical devices still need to be protected from security threats.

This Chapter, therefore, takes a different approach to the TrustFlow approach outlined in Chapter 4. More precisely, this Chapter follows the second approach exposed in Chapter 2, section 2.3. This time, the thesis assumes that critical applications are developed on off-the-shelf fixed microcontroller architectures. Thus, security countermeasures against memory exploits can only be implemented at the software level. However, the objectives of this Chapter remain the same as TrustFlow, and those set out in the thesis approach in Chapter 2:

- **Security**: Like TrustFlow, the primary goal of this software-only approach is to prevent and harden memory-exploits development.

- **Performances**: This thesis targets industrial IoMT applications. As manufacturers optimize their application to the maximum, the countermeasures should not induce

prohibitive execution-time overhead. To remind [40], embedded system manufacturers may not accept more than 5% execution-time overhead.

- **Space overhead**: IoMT embedded applications are highly constrained. Embedded system platforms have little memory, of the order of a kilobyte. As manufacturers tend to reduce the cost and the space taken by an embedded application to its lowest, security protections should not induce prohibitive extra memory overhead.

- **Flexibility**: In comparison with TrustFlow, this time, the security must be the least intrusive. It should be easy to integrate within a practical embedded software development cycle, including incremental tests and targeting a wide range of embedded systems.

- **Compatibility and modularity**: The security should be compatible with third-party libraries and the existing C standards. Also, it should be compatible with incremental development and able software developers to work with separated modules.

- **Practicality**: To avoid security mistakes the security should be transparent for the developer. Embedded system developers do not always have advanced expertise in security. Thus, countermeasures should require minimum user interactions [40].

To address these challenges, this chapter proposes BackGuard, a flexible compiler toolchain that mitigates memory exploits. BackGuard reduces memory-based attack surfaces by hardening embedded applications with a bitmap-based backward-edge control-flow integrity protection. Besides, BackGuard complements the bitmap with an adaptation of the SSP called Random-Guard, and memory barriers called RO-Guards. The Random-Guards cluster vulnerable buffers from sensitive data such as function pointers. Conversely, RO-Guards clusters memory sections preventing spatial memory exploits to spread. Finally, BackGuard enforces DEP.

In this work, the efficiency of BackGuard is showcased on the ARM Cortex-M architecture. However, at the end of the chapter, several openings are given to allow BackGuard to be ported to other architectures at a lower cost. Finally, BackGuard induces the least possible modification of existing embedded systems development tools. In summary, the major contributions of this work are listed below:

- **Backguard**; a set of LLVM [23] compiler plugins that harden low-end microcontrollers against memory safety exploits. BackFlow is highly flexible, easy to use, deployable on several architectures requiring no internal changes of the LLVM compiler;

- **A security evaluation of BackGuard**; Both a quantitative and qualitative security evaluation of the framework are performed in this Chapter. This includes the average target reduction (AIR) measurements of the control flow integrity protection, the return-oriented programming gadgets reduction, and a qualitative basic exploitation test based on SecPump a security-critical wireless infusion pump [149].

- **Benchmark of Backguard**; The code-size overhead induced by BackGuard is measured on a real representative wireless infusion pump. Regarding the execution-time overhead, 145 measurements are performed on various benchmark programs such as Dhrystone [162], CoreMark [150], and BEEBS [163].

5.2 Approach

To address the BackGuard objectives, this chapter proposes the same top-down approach as TrustFlow. However, unlike the previous chapter on TrustFlow, the ability to integrate security features is much more restricted. While TrustFlow assumes that the security can be implemented on both the hardware and the software layer, BackGuard assumes that microcontrollers are provided as is by manufacturers and thus only the software layer can be modified. Like TrustFlow, Backguard tackles the memory safety issue from the initial software design-time to the execution on the hardware. At execution-time, the application is protected in a secure environment. This time, the secure environment is assured by both the software and the hardware. But, as the hardware is immutable, extra security features for memory safety can only be managed by the software. It should be mentioned that the software can efficiently use the available hardware to create a secure environment. The main purpose of the secure environment is to protect embedded software against spatial memory-safety exploits. This secure environment focuses on buffer overflow mitigation because they represent the top-rated threat [164] in the current low-end embedded system.

As TrustFlow, the secure environment assumes that the executed embedded application contains memory corruption bugs that can be turned into exploits. However, as BackGuard has no access to the hardware layer to provide extra security features, and to not induce excessive timing and size penalty, the protection offered by BackGuard is more relaxed than TrustFlow. BackGuard's secure environment focuses on the control-flow integrity which is less accurate than the fine-grained data-flow integrity provided by TrustFlow. Taking this into account, the hardware/software secure environment provided by BackGuard tracks the control flow integrity of a running application and raises the bar of exploit development. If an exploit attempt is detected by the secure environment the execution flow is interrupted.

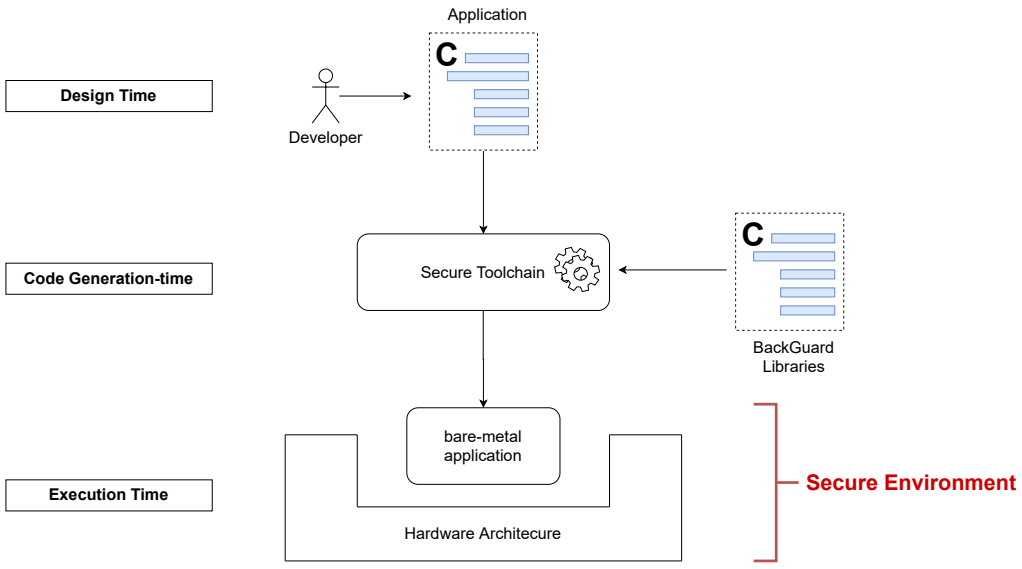

Figure 5.1: BackGuard top-down approach

The secure environment is entirely managed by the software. The latter configures the

fixed hardware efficiently and offers run-time control-flow integrity protection generated at compilation-time by a flexible toolchain. This toolchain is composed of both secure libraries and a set of compiler plugin. These libraries are linked to the final application to ensure the proper hardware configuration while the compiler hardens the final binary code against memory exploits. Following the approach, the BackGuard's software toolchain allows the developer to easily generate a secure code with minimal effort as the security is enforced by the compiler.

5.2.1 Protection concept

Typical control-flow exploits leverage in-memory control-flow information to hijack the execution flow. This control-flow information is either indirect backward-edges (function return addresses) or indirect forward-edges (function pointers). BackGuard focuses on indirect backward-edge control-flow integrity. Regarding indirect forward-edges, several practical compiler works have already taken the issue seriously and provided flexible defenses [92]. Also, indirect forward-edges such as C++ virtual methods tables pointers are less frequent in embedded system programming, and specifically in safety-critical embedded applications.

Protecting indirect backward-edges implies the protection of return addresses. To do so, one way is to duplicate the function return address into a shadow stack [158, 159]. Then, when a function returns, the validity of the return address is checked using the shadow stack. Another concept is to identify return addresses on an isolated bitmap. The bitmap is an array that identifies a valid return address destination using a single activated bit. Thus, at runtime, before each function return, the bitmap is checked to certify that the destination address is valid.

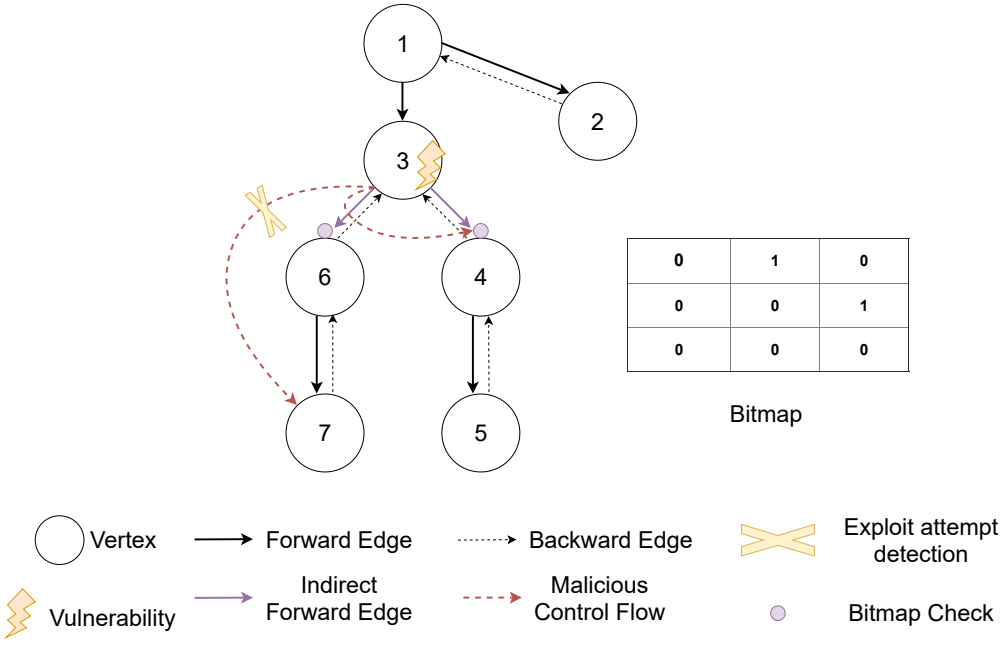

Figure 5.2: Control Flow Guard

Microsoft Control Flow Guard [165] is a compiler and operating system protection that uses a bitmap to prevent forward-edge hijacking. A simplified version of the concept is displayed in Figure 5.2. At program start, Control Flow Guard initializes a static bitmap where each set bit identifies a valid indirect forward-edge destination. When an indirect forward-edge is performed, Control Flow Guard intercepts the transition and checks that the edge destination corresponds to a valid bit set in the map. For instance, the indirect forward-edge from vertex 3 to 6 in Figure 5.2 triggers Control Flow Guard.

Unfortunately, all valid destination targets are statically initialized by Control Flow Guard. This means that a hijacked forward-edge can target any wrong valid destination address set in the bitmap. For instance, in Figure 5.2, Control Flow Guard can detect an indirect malicious forward-edge from vertex 3 to 7, but not from vertex 3 to 4 when the expected branch was 3 to 6. Regarding the state of the art control-flow integrity definitions performed in Chapter 1, section 1.4.1, Control Flow Guard is coarse-grained and leaves room for advanced control-flow bending attacks [166]. However, Control Flow Guard still raises the bar for attackers.

BackGuard proposes bitmap-based control-flow integrity lighter than Microsoft Control Flow Guard that does not require a complex operating system. Indeed, BackGuard mostly targets single stack embedded system applications. It proposes more precise bitmap protection than [165] thanks to return address activation and deactivation in the bitmap. The overall bitmap protection concept proposed by BackGuard is displayed in Figure 5.3.

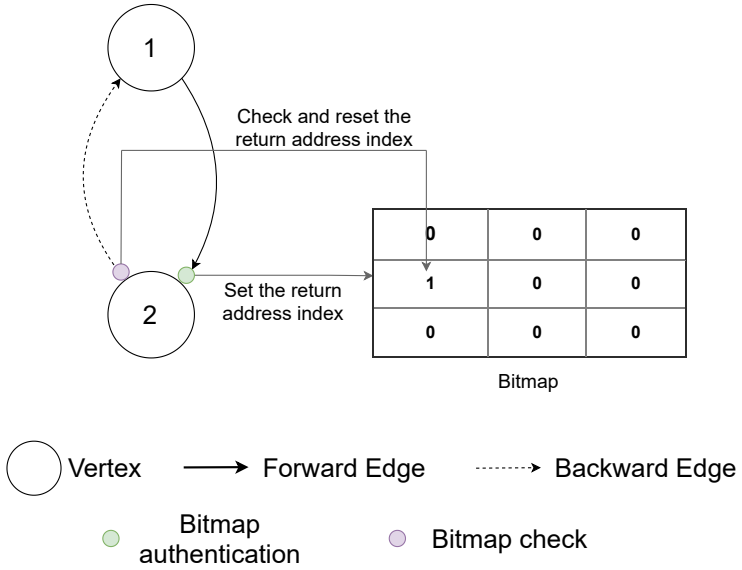

Figure 5.3: BackGuard bitmap concept

When a function is called, its return address is activated in the bitmap. The activation process takes the return address and performs an Address Translation (AT) that results in the activation of a corresponding bit in the bitmap. Then, when a function returns, its return address is translated to check whether the corresponding bit in the bitmap is valid. In the case of a match, the index in the bitmap is invalidated and the execution flow continues. Conversely, in case of mismatch, a control-flow exception is raised. To

summarize, a set bit in the map refers to a valid return pointer.

The advantage of a bitmap approach over a shadow stack is that its size is fixed at compile time. Each bit in the map identifies an executable address pointer which is generally 32 bits. As a result, the bitmap size is fixed to 3.125% of the total memory.

Finally, the bitmap approach maintains the application determinism. While this approach may induce an obvious execution-time overhead due to function activation, deactivation, and indirect backward-edge verification, the address translation is operated by fixed instructions that remain independent of the function that identifies in the bitmap. Consequently, it has no impact on the deterministic performances of an embedded application.

5.2.2 Security challenges

Regarding the security, the bitmap is a memory area that acts as a checker to validate the integrity of return addresses. Unfortunately, like other memory areas such as the stack, the BSS, and the data sections it can be jeopardized by the out-of-bound vulnerability. To illustrate these issues, Figure 5.4 proposes to display the memory map of the SecPump-BLE embedded application that is based on an ARM Cortex M3 microcontroller. It should be mentioned that this memory mapping is common to ARMv7 architectures.

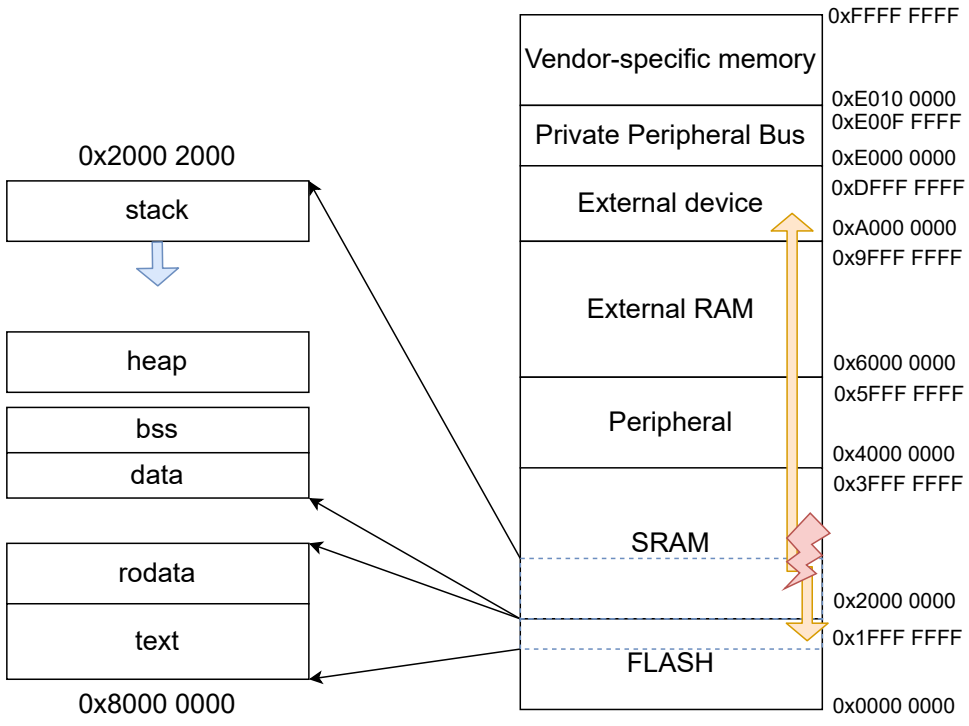

Figure 5.4: ARMv7 memory layout

In Figure 5.4, the processor chips a default memory map that addresses up to 4 Gbytes of memory. The embedded application code and the rodata are placed in the upper memory of the flash, while the stack, the heap, the BSS, and the data sections are located in the SRAM. More specifically, the data and BSS sections are copied from the flash

memory to the SRAM during the boot sequence of the microcontroller.

By default, the STM32s' stack is ascending downwards, and as the other sections located in the SRAM, the stack enforces read, write, and execute permissions.

The peripherals' memory mapping is located above the SRAM memory section of the microcontroller. These peripherals mostly include serial communications, GPIOs, and any external hardware peripherals. Of course, the peripherals are configured and accessed by embedded applications by writing and reading to their memory-mapped addresses.

Both the external RAM and external memory regions are usually dedicated to external memory and coprocessor. These memory regions are unused on the STM32 platform. The top memory regions such as the "Private Peripheral Bus" and "Vendor-specific memory" are present on every ARMv7-M processor. These regions, more specifically the "Private Peripheral Bus" region, handle the System Control Space (SCS) including the Nested Vector Interrupt Controller (NVIC), and the Memory Protection Unit (MPU), and the system reset.

Finally, it should be mentioned that the ARMv7-M architecture offers two execution modes. By default, an application runs at the highest privileges. This implies that the application can use all instructions and all processor resources. Conversely, the unprivileged mode has limited access to system registers and instructions. The unprivileged mode also restricts access to peripherals.

Following this brief discussion, it seems that a simple linear stack buffer overflow/underflow can take over an entire embedded application. First, a memory exploit can target a return address located in the stack and divert the execution flow of an application on a previously injected malicious code. Also, as the peripherals are located above the stack memory region. Their configuration, as well as the I/O of the application, can be tampered leading to undefined malicious behavior. Finally, as the flash memory enforces read, write, and execute by default, an attacker can even tamper the firmware and modify its integrity.

As a direct consequence, protecting an in-memory bitmap from a spatial vulnerability seems to be a difficult challenge. A naive technique to complicate spatial attacks is to place the bitmap in a hard-to-reach place in memory. For instance, by leveraging the read, write, execute property of the flash, the bitmap can be placed below the text section as displayed in Figure 5.5. Therefore, it would be harder to reach the bitmap using a simple out-of-bounds vulnerability located in the SRAM.

Unfortunately, just placing the bitmap section below the other sections is not enough. Certain out-of-bound attacks may allow an unlimited number of bytes to be injected in the memory space and reach the bitmap from underneath. An interesting improvement to protect the bitmap would be to use a Memory Protection Unit (MPU) combined with processor privileges execution modes. For instance, most ARM Cortex M0+/M3/M7 possess two privileges mode and an MPU. The MPU would only allow the privilege mode to write the bitmap and the embedded application would run at the least privilege level. Unfortunately, function calls and returns occur frequently. Thus, following the least privilege principle, each function call and return may generate an exception that escalates privilege

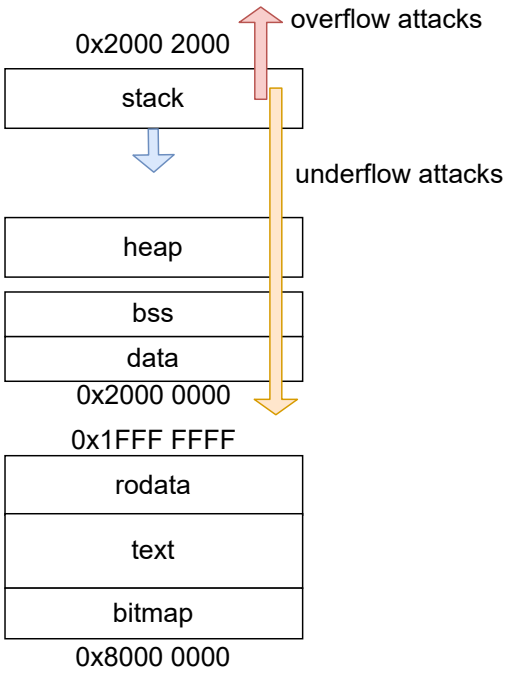

Figure 5.5: Bitmap memory position

and then access the bitmap. Regarding the execution-time cost, this type of protection may induce a high overhead and even double the execution-time overhead of tiny functions. Besides, by default embedded applications on ARM microcontrollers are running at the highest privileges. From a security point of view, it is not the best practice. However, dropping the privileges by default to protect the bitmap may introduce additional security and development constraints that may lead non-security experts to security mistakes. For instance, only privilege mode can access the peripherals, thus dropping the privileges after protecting the bitmap may require the embedded application developer to instrument each peripheral accesses such that the application escalates privileges before accessing the peripheral and dropping the privileges after accessing the peripherals.

Instead of separating privileges, BackGuard proposes a more relaxed policy that still protects the bitmap. This policy is based on read-only guards (RO-Guards). These guards, displayed in Figure 5.6, cluster the boundaries of the bitmap, the stack, and the unsafe global variables in the SRAM.

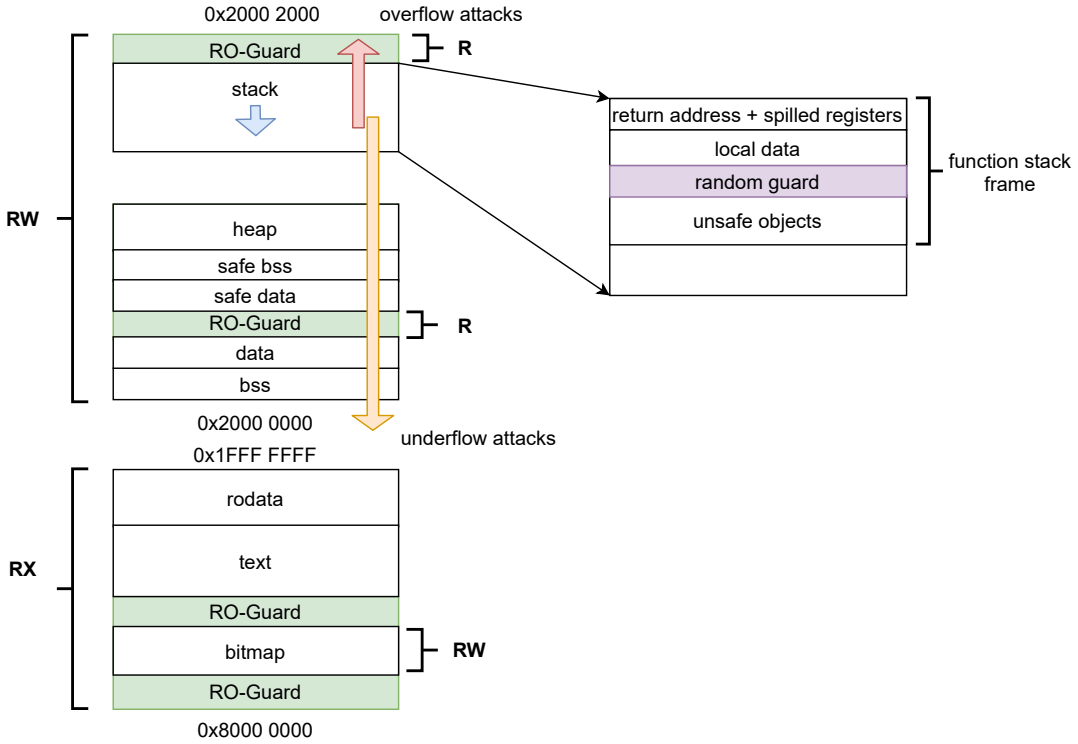

Figure 5.6: BackGuard protections

Besides the bitmap protection, BackGuard proposes two additional protections that make exploit development harder. The first is the Data Execution Prevention (DEP) policy applied to the SRAM sections. DEP is also combined with a Read eXecute (RX) policy for the code section and the rodata section located in the flash of the microcontroller. Of course, the bitmap section stays writable. Thanks to this DEP protection, an attacker may not be able to perform code-injection attacks anymore.

The second protection is called unsafe object clustering with Random Guards (RandGuard). This protection, displayed in Figure 5.6 is an adaptation of the Stack Smash Protector but for bare-metal embedded system application. The protection places RandGuards between unsafe memory objects, code-pointers, and data-pointers. These RandGuards cluster overflowable objects to prevent out-of-bounds attack propagation.

5.2.3 Implementation challenges

To summarize, BackGuard hardens low-end applications by enforcing both backward-edge control-flow integrity thanks to a dedicated bitmap where each set bit refers to a valid pointer destination. BackGuard also clusters spatial vulnerabilities with Rand-Guards and RO-Guards.

Such security features can only be implemented using a strong compiler infrastructure

and additional secure libraries that are linked to the final executable code. Overall, the BackGuard framework takes up the following implementation challenges:

Backward-edge control-flow integrity: The BackGuard compiler that generates the backward-edge protection should be aware when a return address is spilled in the memory. Once this is the case, BackGuard needs to generate the appropriate code that identifies the return address in the bitmap. Finally, when a function returns using a previously spilled return address, BackGuard should also generate the return address verification code.

Data Execution Prevention (DEP) and RO-Guards: During the boot sequence of the embedded application, an MPU should be configured such that DEP and code anti-tampering is enforced. Furthermore, the MPU needs to insert the RO-Guards to protect the bitmap, delimit the upper stack boundary, and the unsafe global variables memory space.

Rand-Guards: At execution-time, each function's stack should insert a Rand-Guard between vulnerable buffers and data variables. Although this protection is close to the regular stack canary, it needs a random number so that the value of the guard is different for each existing embedded application. Unfortunately, unlike desktop computers, most low-end microcontrollers don't have dedicated hardware support for Random Number Generation/Truly Random Number Generator [54]. Thus an appropriate method must be implemented for random number generation.

5.3 Implementation

BackGuard is implemented within the Clang/LLVM compiler infrastructure. It comes with secure libraries that are linked to the final executable. These libraries configure our STM32F446RE workbench MPU and generate a random number during the initialization sequence of the processor. The details of both the MPU and the random number generation is given further in the section.

The improved LLVM compiler provides bitmap protection. During code generation, the compiler identifies which function saves its return address in the memory. Then, it adds extra instruction to the application code so that vulnerable function activates and deactivates their return address corresponding bit in the bitmap. Regarding the Rand-Guards, the compiler analyses the stack frames data. It identifies vulnerable buffer that may overflow from other data. All the allocated stack frame objects are then sorted such that stack objects that may overflow are placed bellow the Rand-Guard and sensitive data such as function pointers are placed above the Rand-Guard. Finally, during code generation, the compiler changes the attribute section of overflowable global memory objects such that they cannot reach safe objects incase of vulnerability.

5.3.1 Compiler implementation strategy

The Clang/LLVM project is a very modular and well-documented compiler for native languages such as C/C++. The way LLVM is architected makes it easy to implement additional compiler passes. A global overview of the Clang/LLVM compiler infrastructure is displayed in Figure 5.7. One can see that the compiler is divided into several components that drive the compilation flow from C/C++ source codes to the final object-assembly files. In Figure 5.7, the compilation workflow starts with the Clang front-end. The latter translates native languages into a Clang Abstract Synthax Tree. The front-end ends by generating an Intermediate architecture-independent Representation (IR). The IR representation is strongly typed in a static single assignment form providing an infinite set of virtual registers. Briefly, each variable in the intermediate representation form is only assigned once and every variable is defined before it is used. As displayed in Figure 5.7, the Intermediate Representation can be independently optimized and instrumented by the middle-end with the optimization tool “opt”. An interesting feature provided by LLVM is that compiler passes working on the IR can be introduced in the compilation workflow on-the-fly. In other words, additional IR passes can be introduced as external plugins to the compiler without modifying its internals.

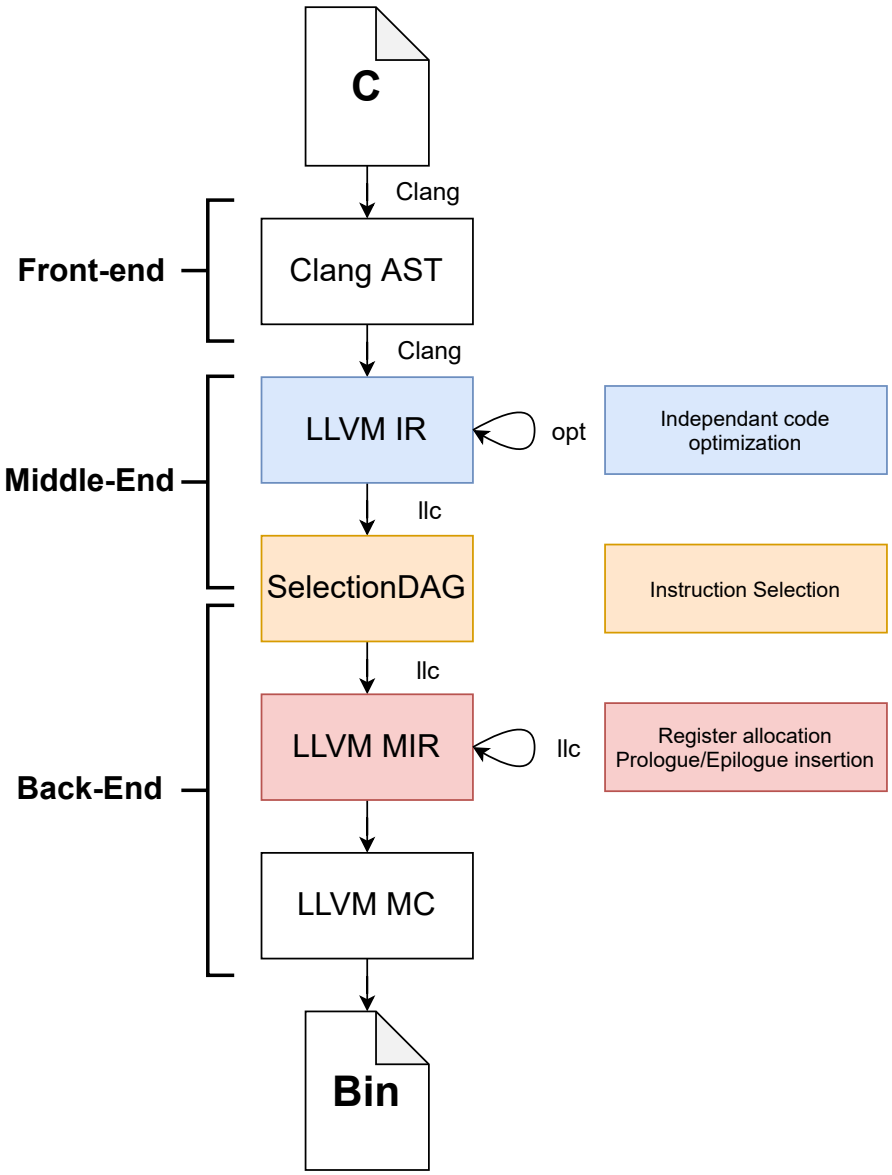

Figure 5.7: Clang/LLVM compilation workflow

After independent optimization, the IR code is passed to the backend that produces the architecture-dependent assembly code. The target-dependent code generation is driven by the "llc" tool. First, the IR is converter into Selection Dag Nodes for instruction selection. Then the DAGs are destroyed and translated to Machine IR (MIR). This representation enables specific target-dependent optimization, register allocation, prologue and epilogue insertion, and machine code instrumentation. Finally, the last step of the compiler emits the machine code and constructs the final object file.

Regarding the compiler infrastructure, it seems that the bitmap return address protection should be implemented in the backend. Indeed, the backend is responsible for target-dependent prologue and epilogue insertion including return address register spill. Thus, one way to protect the return address with the bitmap is to implement a target-dependent machine pass that adds extra instructions per function prologue and epilogue so that every spilled return address in the memory is registered and removed from the bitmap

on purpose. Many extensive works such as [167–169] have worked and implemented memory safety protection for low-end ARM-based microcontrollers in the backend.

However, backend modification imposes invasive changes in the compiler internals. Also, providing security features based on backend modifications breaks flexibility. It forces developers to use a specific version of the compiler and recompiles it from the sources. Indeed, invasive backend modifications only target specific architecture. This means that the portability is low and when changing architecture, the security feature may not be implemented. Conversely, it seems that working on the LLVM IR is much more flexible than providing security at the backend level. It allows the security to be developed as a compiler plugin that can be loaded within any compiler version. Also, it does not require developers to compile their compiler from the sources. Quite the opposite, the security can be loaded in the compilation workflow of the Clang/LLVM version available from officials repositories. However, it should be mentioned that implementing the security at the LLVM IR level raises challenges; Additional IR instructions for security may be removed in backend optimizations, breaking the security.

5.3.2 Additional passes

To ensure flexible security, the BackGuard toolchain introduces two additional Intermediate Representation (IR) passes in the Clang/LLVM compiler workflow. These compiler passes are developed as external compiler libraries and are loaded at compilation time in LLVM as a plugin. Working on the IR provides a high level of flexibility. First, external libraries do not require any invasive internal compiler modification. Second, the security provided by the external libraries can be easily deployed on exotic architectures maintained by the various LLVM backends. Finally, the security can be easily deactivated on purpose.

5.3.2.1 Bitmap hardening pass

The first BackGuard compilation pass enforces return address bit activation and deactivation in the bitmap. As this work focuses on ARM-based microcontrollers it is important to remind the procedure call standard. Function subroutines are called using both the "bl" and "blx" instruction. These "call" instructions pass the execution flow to a subroutine and place the caller return address pointer in the ARM link register (LR). Then, depending on the fact that whether the called function is a leaf function or not, the caller return address is spilled by the compiler.

Unfortunately, the information on whether a function saves the return address in the memory is only available late in the compiler pipeline during prologue/epilogue generation. To determine whether a function is saving its return address in the memory from the LLVM IR, the BackGuard bitmap pass iterates over every function's IR instruction to find function call. If the parsed function is not calling any function, it is considered as a leaf function. Consequently, the compiler backend may not spill its return address making the BackGuard pass skipping the function protection.

The second challenge is to get the return address value from the IR and then perform the address translation to enable the corresponding bit in the bitmap. Usually, the return address is handled in the Link Register which is defined in the compiler backend. However, LLVM provides a specific intrinsic (llvm.returnaddress) that can be used at the IR form to manipulated the return address of a function. This intrinsic function is inlined by the BackGuard compiler pass at the beginning of each function. The returning value of this intrinsic is the return address of the function that is then translated into a bit loaded in the bitmap by classical IR instructions. According to the LLVM documentation, intrinsics are transparent to optimization passes and consequently, they preserve the BackGuard bitmap protection during the whole compilation workflow.

5.3.2.2 Random guard pass

As previously explained, the random guard pass is very similar to the existing stack protector implemented in GCC. Our random guard pass is not the major innovation of this work but it contributes to hardening embedded applications against memory exploits.

The random guard value of BackGuard is based on a random number generated during the boot sequence of the microcontroller. The details of the boot sequence are highlighted in the next sub-section.

During compilation, the BackGuard's Random Guards pass analyses each function at the IR level and identifies stack frames object that may be vulnerable to overflow issues. The local stack frame objects are then sorted such that dangerous objects are placed bellow the Random Guard and safe objects above.

Regarding the global variables, the Random-Guard pass changes the section location of sensitive memory objects. As displayed in Figure 5.4, sensitive memory objects that are supposed to reside in the BSS or the data sections are moved into the safe BSS and the safe data sections.

5.3.3 Boot sequence

Both the bitmap protection and the random guard canary rely on memory protection (MPU) and random number generation (RNG). Concerning the protection of the bitmap, the Data Execution Prevention (DEP), and the code section protection, BackGuard links a secure library to the final embedded application. This library includes a set of functions that are called during the initialization sequence of the microcontroller. These functions configure the Memory Protection Unit of the microcontroller such that it protects the code section with RX, the bitmap section with RW, and the SRAM with RW permissions. BackGuard leverages the fact that the protected memory regions can overlap. For the ARMv7-M architecture, the region with the highest number has a priority over the other region with the lowest number (e.g region 4 protection takes precedence over region 2). Consequently, the configuration of the bitmap region takes precedence over the whole flash configuration. The configuration of the MPU is summarized in Table 5.1.

Table 5.1: MPU configuration

Priority	**Memory**	**Permissions**
1	Code	RX
2	Data (stack, BSS, Data, unsafe-regions)	RW
3	Stack-RO-Guard	R
4	Date-RO-Guard	R
5	Bitmap	RW
6	MPU-Configuration	R

R=Read; W=Write; X=Execute

The boot sequence of the microcontroller also includes the generation of the Random-Guard value. BackGuard leverages the property of SRAM's initial values to generate a random number. As a disclaimer, the goal of this study is not to evaluate the randomness of microcontroller-based SRAM initial values. Several works in the literature address this topic and provide very relevant results [170]. According to [171] and for the rest of the study we will consider that 20 bits of SRAM cells are enough to generate a random bit. Thus to generate a strong 256 bits random number, BackGuard bootloader reads 640 bytes of SRAM and passes them through an embedded BLAKE hash function [172]. As the 640 bytes of SRAM are random and the BLAKE hash function is a cryptographic pseudorandom function, the generated 256 bits value is random. The 32 bits value of the canary is then extracted from the generated key and placed in a read-only location in the flash.

5.4 Evaluation

This section assesses BackGuard protection. The whole BackGuard compiler extensions represent around 900 lines of C++ code. The external libraries of BackGuard include the custom bootloader that configures the memory protection unit (MPU), the bitmap, and the random guard value. It represents around 500 lines of C.

While BackGuard operates at the LLVM-IR and thus is architecture-independent, the evaluation of the countermeasure is performed on an off-the-shelf STM32F446-RE board based on the ARM architecture. More precisely, the security is assessed using SecPump-BLE [149] the STM32 security-oriented system presented in Chapter 3. Both quantitative and qualitative security assessments are performed on the pump. This evaluation reveals the average target reduction of BackGuard, the gadget reduction, and finally, the exploit development steps required by an attacker to bypass BackGuard.

While the size overhead of BackGuard is measured on the pump application, the execution-time overhead of the protection is measured with well-known benchmark suites dedicated to low-end ARM microcontrollers such as CoreMark [150], Dhrystone [162], and BEEBS [163]. Also, we implemented a backend variant of BackGuard and a shadow-stack variant of BackGuard to compare them with the IR variant detailed along with this Chapter. In the end, BackGuard is a compiler security framework offering several protection variants from middle-end to the backend. Finally, this section demonstrates that an acceptable level of security can be achieved on low-end microcontrollers with very few constraints and modifications in the development toolchains commonly used by embedded system developers.

5.4.1 Security evaluation

As previously explained, SecPump [149] is used to assess the security protection induced by BackGuard. SecPump is a functional wireless infusion pump system workbench tailored to security evaluation. The cyber-physical model targets on an off-the-shelf STM32 with a Bluetooth Low Energy extension board. It is highly representative of the next generation of IoT devices. One of the main advantages of the platform is that it models the classical closed-loop regulation of the glucose-insulin thanks to an integrated PID.

Quantitative Analysis: The entire embedded application consists of 18312 lines of C (including the external libraries). To evaluate the control-flow integrity accuracy provided by the BackGuard's bitmap, the Average Indirect target Reduction [69] (AIR) is measured on the secured application. To remind from Chapter 1, section 1.3.1, the AIR [69], gives a percentage of how much a protection enhances compliance of an application to its control-flow graph. The AIR is given by Formula 5.1.

$$AIR = \frac{1}{n}\sum_{j=0}^{n}(1 - \frac{T_j}{S}) \tag{5.1}$$

N is the number of indirect control-flow transfers in a program, S the total number of target addresses that can be reached by a branch (it is the size of the binary). T_j

represents the number of addresses that can be reached by a transfer j restricted by control-flow integrity protection. It seems that if the percentage given by this formula is close to 99% the control-flow solution can be considered as fine-grained [70]. Conversely, a very coarse-grained control-flow protection has an AIR close to 80% [70].

The bitmap protection enforces indirect backward-edge integrity. As a result, forward-edges are left unprotected. To measure the impact of unprotected forward-edges and the accuracy of protected backward-edges, we first measure the BAIR (Backward-edge Average Indirect target Reduction) provided by BackGuard which does not take into account the forward-edges. Then, we measured the AIR to determine the impact of unprotected forward-edges on the overall protection.

As a reminder, the bitmap protection is a little less accurate than a regular shadow call stack. Each function can only return to an activated return call-site in the bitmap. These activated return-sites are part of the call stack of an activated function. When returning, a function can thus return to a few valid locations in its call stack.

To measure the BAIR and the impact of the little inaccuracy of the protection, a specific graph analysis algorithm is developed. The latter evaluates the worst-case valid return call site per function and the best-case valid return call site per function. For SecPump, it results in both a worst-case and best-case direct acyclic graph. Both graphs are then processed with Formula 5.1 to determine the worst-case and best-case average indirect target reduction. The results are displayed in Table 5.2.

Table 5.2: AIR measurements

	Best-case	**Worst-case**
BAIR	99.99%	99.98%
AIR	83.33%	83.31%

According to the measurements, the protection enforces the consistency of the control-flow graph to 99% for function returns. In turn, the overall AIR is much less accurate due to the lack of forward-edges protection. Although indirect forward-edges remain exploitable, they are considered as a low-security risk. Indirect function calls using pointers are less used in the context of embedded systems programming. Combined with the fact that BackGuard isolates function pointers from vulnerable buffers using random guards, the likelihood of working indirect function call exploits is possible but harder.

Finally, using ROPgadget [94], 2261 return-oriented gadgets in the whole SecPump application are found. After applying the bitmap protection, we measured that in the worst-case only 14 gadgets were usable at the same time (99.4% gadget reduction).

Qualitative Analysis: To improve the quantitative security analysis, we propose a qualitative analysis of the protection by describing the exploit development workflow to thwart BackGuard. To validate this workflow, The security exploits exposed in section 3.4.1 are re-used and readapted.

Figure 5.8 displays the development steps of a functional exploit in the presence of

BackGuard.

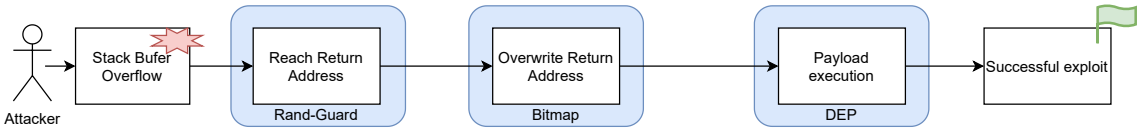

Figure 5.8: Exploit workflow

First, to reach any sensitive data, the attacker should pass the Rand-Guard protection. The Rand-Guard value can be leaked through a memory disclosure or with physical access. However, its value would only be valid for a single vulnerable application thanks to the property enforced by SRAM based Physical Unclonable Function (PUFs). Thus, to be efficient and to scale multiple targets, an exploit should leak the Rand-Guard any time it targets an application. Second, the attacker can either corrupt sensitive data such as function pointers or a function return address. Unfortunately, when targeting a return address the attacker has to replace it with a valid destination on the map or target a function pointer.

The other option is that the attacker tampers the bitmap. However, the latter is deeply buried in the flash and surrounded by RO-Guard that prevents spatial attacks. Of course, the bitmap is not unreachable, one way to corrupt it is to acquire a data pointer, make it point to the target index in the bitmap, use this pointer to write the bitmap and finally corrupt an in-memory return address to redirect the execution flow on the desired location.

Although this scenario is quite likely, it is tedious and requires several primitives. BackGuard considerably increases memory safety. Finally, as BackGuard enforces Data Execution Prevention (DEP), no malicious code can be executed in the stack. Thus, attackers should use advanced return-oriented programming gadgets to develop functional exploits.

5.4.2 Costs

5.4.2.1 Size overhead

The size overhead induced by BackGuard is measured in three different cases. First, the reference application is compiled with Clang/LLVM without the BackGuard protection options but with the “Os” optimization (size optimization). Second, the application is compiled with the bitmap protection only, and finally, the bitmap protection combined with the Rand-Guard. The vanilla application as a fixed code size of 29504 bytes and 32064 of total bytes in the flash. The application leaves 94% of the flash memory free.

Using the bitmap protection, any code pointers (executable address in the flash) is identified with a bit. As in-memory code-pointers are 32 bits the total size of the bitmap represents 3.1% of the flash memory. In our case, it represents 32768 additional bytes reserved in the flash. Furthermore, the additional code generated by the compiler to interact with the bitmap increases the total code application size up to 32264 bytes (9% increase). Overall the applied bitmap protection increases the application size up to

67592 bytes which is a 110% application code size overhead. However, it still leaves 88% of the entire microcontroller memory free. It is important to remind that the bitmap is fixed. Thus, although it induces a significant size overhead for a small application, its memory usage regarding the available memory of the microcontroller is very small (3.1%). Furthermore, the size of the bitmap is not linked to the size of the compiled application. Consequently, as long as the application does not use more than 96.9% of the microcontroller memory it should not be an issue.

Finally, the fully protected application occupies 81208 bytes of flash which represents a size overhead of 153%. Again, it leaves 85% of the microcontroller memory free. More specifically, we measured a code-size increase of 55%. This code-size increase includes the custom bootloader that initializes the random guard by reading the flash and passing the value to the BLAKE hash function, the MPU configuration, the bitmap checks, and the random guard insertion/checks. We believe that this 55% should decrease with a bigger application. Indeed, among the 55% increase, 84% represents the custom bootloader application that configures the MPU, reads the uninitiated SRAM, and hashes the random value using the BLAKE algorithm.

Figure 5.9 recapitulates the memory occupation of each compiled version of the protection. One can see that the more the application code increases, the more the occupation of the bitmap is reduced. This is because the bitmap is fixed and thus only the application code percentage is expected to grow. The reasoning remains the same for the memory occupation repartition of the full-protected version. Since the bitmap and the bootloader are fixed their memory sized percentage occupation tends to decrease in favor of an application code increase.

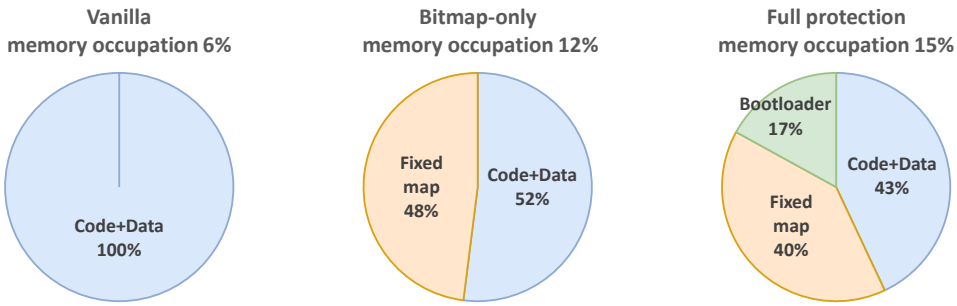

Figure 5.9: BackGuard memory occupation

5.4.2.2 Execution-time overhead

Two execution-time overhead is induced by BackGuard. First, during the initialization of the microcontroller, in addition to configuring all the peripherals, initializing the stack, and copying the data of the microcontroller in the SRAM, BackGuard should initialize the bitmap, the MPU, and the Rand-Guards with the BLAKE hash function. These additional operations increase the boot-time by 48%.

To assess the average execution-time overhead after boot-time, various embedded sys-

tem benchmark suites such as CoreMark [150], Dhrystone [162], and various embedded benchmarks from BEEBS [163] are compiled with BackGuard.

The average execution-time overhead induced by BackGuard is measured with optimization level 2. The CoreMark benchmark suite is configured to run up to 1000 iterations and the Dhrystone benchmark suite up to 50000 iterations.

To obtain accurate execution-time measurements we leveraged the STM32F446RE cycle counter. We developed a simple library that resets the cycle counter before performing the various benchmarks and displays its value at the end of the execution.

As mentioned earlier, several variants of BackGuard are added to the compiler. These variants include a backend implementation of BackGuard and a shadow stack variant of BackGuard. They allow them to be compared with the IR implementation of the original bitmap. The backend implementation of BackGuard is a machine pass that constructs the bitmap registration/un-registration after the prologue and epilogue insertion. This pass is completely compatible with the IT Block optimization pass performed by the ARM compiler backend (for more details about this machine bitmap, we refer the reader to Annex C). The shadow stack variant operates on the intermediate representation (IR).

The execution-time overhead results are displayed in Figure 5.10.

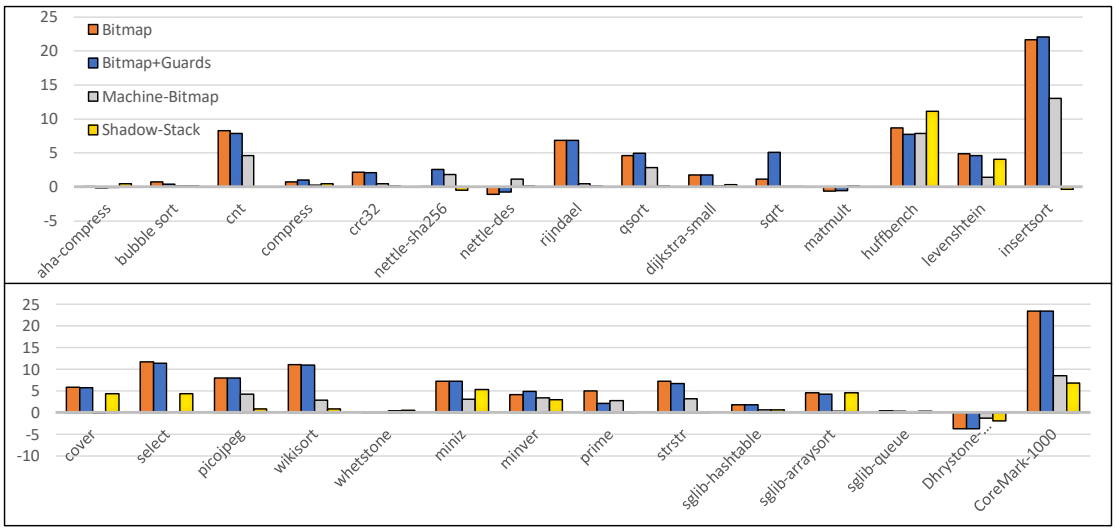

Figure 5.10: BackGuard benchmarks

The Y-axis represents the execution-time overhead percentage, the X-axis represents the benchmark suite

In Figure 5.10, the "Bitmap" represents the execution-time overhead induced by the BackGuard Intermediate Representation protection, the "Bitmap+Guards" represents the execution-time overhead induced by both the bitmap protection and the Rand-Guard insertion, the "Machine-Bitmap" represents the execution-time overhead measured by the backend bitmap pass and the "Shadow-Stack" displays the execution-time overhead measured induced by the shadow stack implementation.

On average we measured 5.06% execution-time overhead for the IR bitmap and 5.13% execution-time overhead for the shielded version. This execution-time overhead increase is

consistent with the fact that the combination of two protections such as the Rand-Guard insertion and the bitmap verification increases the size of the code. For the machine bitmap pass, we measured an average execution-time overhead of 2.15%, which is more than 50% times more optimized in comparison with the IR protection. This result is consistent, the back-end has more control over the generated code than in the intermediate representation. The machine pass can optimize to the maximum the instructions used by the compiler to interact with the bitmap. Finally, the shadow stack version of BackGuard induces less execution-time overhead. We measured 1.59% execution-time overhead induced by the shadow stack protection. The shadow stack protection is much less expensive than the bitmap protection in terms of execution cost because it adds far fewer protected per-function instructions than the bitmap. The only expensive interactions induced by the shadow stack are the push and pop instructions used to check the return addresses of protected functions. To conclude, the three approaches proposed in the course of this evaluation are consistent with our assumptions and in all cases result in around 5% execution-time overhead.

5.5 Discussion

This section outlines the current limitations of BackGuard and the research avenues for its improvement. As with any software defense, it is fair to state that BackGuard is not perfect protection. However, BackGuard still protects low-end microcontrollers from a wide range of trivial spatial memory attacks.

Regarding limitations, BackGuard is not well designed to support the famous *"setjmp/longjmp"* issue. Unlike TrustFlow which perfectly handles it, BackGuard handles it with a little bias. Indeed, the *"setjmp/longjmp"* functions are generally used to restore a program to a previous state. In the context of embedded system programming, *"setjmp/longjmp"* can be used in exception handling. The effect of the *"longjmp"* function is to restore the program and its stack frame to the *"setjmp"* point. Unfortunately, this jump breaks the classical call/return construction of a program. As a result, some functions called after the *"setjmp"* point may never return. For shadow call stack this might be a problem because the top of the shadow stack may not match the restored function's return address. For the bitmap, the restoration state might not be an issue, as all the whole call stack is activated on the map. Unfortunately, all functions called after the *"setjmp"* may never be deactivated from the map. This opens new valid target destination addresses for an attacker. One way to tackle this issue would be to deactivate the functions from which the stack frame is destroyed by the *"longjmp"*. While not impossible, this may require to parse the call stack trace of an application before the *"longjmp"* and after the *"longjmp"*. Then, the return address of each function freed by the *"longjmp"* should be identified and deactivated from the bitmap. Although this process might be costly, it may worth the price if an application rarely uses *"setjmp"* and *"longjmp"*.

Unlike TrustFlow, BackGuard is currently incompatible with recursion. A recursive function may call itself more than twice. As a result, after the first return, the function return pointer's corresponding bit in the bitmap is unset raising an exception at the second return. There are several ways to tackle this recursion issue. First, BackGuard can be provided with coding rules like TrustFlow that prevent the usage of recursion. Indeed, recursive function calls in low-end embedded systems may blow up the stack. Usually, the upper bound of recursive functions is difficult to predict with static analyzers and code verification tools. As a result, the usage of recursion in low-end embedded systems is inconsistent with safe development practices. To push it further, rule 16.2 of MISRA-C [32] for safety-critical embedded application bans the usage of recursion. Finally, recursive functions are computable by an iterative model [38]. In many cases, converting a recursive function into an iterative model can be achieved using a global stack completely compatible with BackGuard. Another solution to handle recursion is to use the shadow stack variant of BackGuard.

Currently, BackGuard is not implemented within multitasking real-time operating systems [29]. Of course, managing multiple tasks is hardly possible with a single bitmap. In a low-end real-time system, two separate tasks may share multiple functions and the scenario that two tasks may concurrently access the same function is plausible. Unfortunately, it

may result in a false-positive control-flow integrity violation.

One way the bitmap approach would be compatible with a multitasking system is to develop an application where every single task does not share any function. In this case, a single bitmap may handle a whole application. Another way would be to handle a single bitmap per task. Unfortunately, a single bitmap takes up to 3.1% of the whole memory. While this size overhead is acceptable for a single bare-metal system it may increase proportionally with the number of tasks to be secured.

It seems that a shadow stack approach could be a better approach than the bitmap for multitasking embedded applications. Following this principle, each process would handle a per task shadow stack that is switched and restored by the scheduler on purpose.

Since the BackGuard compiler extensions set acts on the intermediate representation (IR) of LLVM, its transformations are architecture-independent. We extensively test the implementation of BackGuard on the ARM instruction set using an STM32 as a system workbench. However, using the Clang/LLVM compiler version 9.0.1, we were able to implement BackGuard on the RISC-V, SPARC, MIPS, x86, Aarch64 architectures making the overall defense inherently scalable. Of course, the configuration of the MPU, and the generation of a random guard may depend on the target microcontroller. They represent the only architectural dependant part of the BackGuard framework that should be adapted.

5.6 Comparison with related work

This section discusses a coherent selection of well-known software-based memory safety protection dedicated to bare-metal embedded systems. Their effectiveness, approaches, and flexibility is compared with BackGuard.

RECFISH [70] is an ARM instrumentation-based framework that protects real-time embedded systems against memory safety issues. RECFISH instruments pre-compiled minimal real-time embedded systems to enforce control-flow integrity. It protects function backward-edges thanks to an RTOS-MPU protected shadow stack and indirect forward-edges thanks to immutable labels. While the backward-edge security policy of both RECFISH and BackGuard is broadly the same, BackGuard protects sensitive function pointers with Random-Guard, RECFISH enforces control-flow graph coherency with labels. Thus, RECFISH can be vulnerable to control-flow bending attacks [166] while BackGuard can be defeated by memory disclosures. Regarding the execution-time overhead, RECFISH induces 30% execution-time overhead on CoreMark 1000 while BackGuard induces 6.8% to 23.4% depending on the variant. In terms of flexibility, RECFISH applies to pre-compiled ARM binaries. This approach is better in security that BackGuard. It allows instrumenting pre-compiled/closed-source libraries. Unfortunately, binary patching is not always compatible with software-engineering aspects. BackGuard provides a compiler-oriented approach compatible with full modularity, portability, and compatibility with software tests. Besides, BackGuard operates on the LLVM-IR making it more flexible than RECFISH.

µArmor [54] is a Clang/LLVM compiler-based memory safety protection for ARM bare-metal embedded systems. µArmor enforces three protections to mitigate memory-based exploits: µESP, µSSP, and µScramble. µESP enforces non-executable permissions on non-code regions using the ARM MPU. µSSP is a stack smashing protection based on a random value generated by µRNG, an SRAM based PUF random number generator. As BackGuard, µSSP clusters vulnerable buffers from sensitive data per stack frames. However, unlike BackGuard µArmor does not specifically enforce control-flow integrity. In the case of stack memory disclosure, return address pointers are unprotected allowing attackers to hijack the execution flow. Finally, µScramble is a software diversity compiler pass that mimics fine-grained address space layout randomization protection. µScramble shuffles registers, function orders, and inserts dead-code. Unfortunately, the diversity of the code generated by µScramble is limited by the size of the microcontroller and code memory leaks. The former restricts the numbers of possible morphs that can be generated by the compiler while the latter can completely de-randomize the code memory section allowing on-the-fly memory explois [106]. Also, regarding the thesis's approach, software diversity is prohibited in safety-critical devices. Finally, BackGuard is more flexible than µArmor. µArmor requires to recompile the application code before being deployed on the hardware. Also, µArmor involves several invasive backend compiler modifications while BackGuard is more flexible by operating on the IR. Finally, regarding the performances, µArmor induces less than 1% execution-time overhead on average and less than 5% size

overhead which is better than BackGuard.

Silhouette [168] is a memory isolation protection that efficiently protects ARM embedded systems against control-flow hijacking attacks. Silhouette is based on the LLVM compiler and introduces native code generative passes to enforce memory safety. Silhouette protects backward-edges thanks to a protected shadow call stack. Besides, like RECFISH [70], Silhouette protects forward-edges thanks to immutable labels that force the application execution to comply with its control-flow graph. Finally, to protect the shadow stack Silhouette leverages the feature of the unprivileged STRT instructions of the ARM instruction set. Regardless of the processor execution mode, these specific instructions are treated as if they were executed in unprivileged mode. Thus, Silhouette leverage this feature to create two protection domains. On the one hand, the application is only using unprivileged stores and on the other hand privileged domain uses the regular ARM STR instructions. The silhouette's MPU is configured such that the shadow stack is readable and writeable in privileged mode but only readable in unprivileged mode. Hence it ensures that privileged code running using unprivileged instructions and unprivileged code cannot tamper the shadow stack. As a result, only code function prologue and epilogue uses regular STR instructions to write in the shadow stack. Silhouette provides better security protection than BackGuard. It ensures safe memory accesses thanks to the ARM STRT instruction feature. However, in its threat model silhouette does not protect the HAL library code. This code should be run at the highest privileges because it accesses the peripherals and the I/O. While this code is rendered unreachable from the application point of view, it may contain unprotected exploitable vulnerabilities that BackGuard considers. On average Silhouette induces less than 5% execution-time overhead and a geometric mean of 21.4% and 25.5% code size overhead on CoreMark-Pro and BEEBS. However, to maintain the STRT memory access feature, Silhouette relies on heavy modification of the LLVM ARM backend. While it provides safer security than BackGuard its flexibility and portability are more impacted.

EPOXY [167] is an LLVM compiler-based embedded system protection that mixes memory safety and least privilege execution. EPOXY extends the CPI and CPS project such that any embedded application maintains two concurrent stacks. One stack is dedicated to unsafe objects such as buffer, the other stack is dedicated to sensitive code and data pointers. EPOXY [162] clusters the unsafe stack with a guard to detect any overflow attempt. At boot-time EPOXY also configure the ARM microcontroller to enforce DEP and code anti-tampering. EPOXY drops the privileges of the running application to prevent any malicious write to either modify the MPU configuration, sensitive IO, or peripherals. At compilation time EPOXY identifies which store instruction is allowed to write the sensitive IO/peripherals. Before each privileged store, EPOXY raises the application privileges such that the store can be performed. The privileges are dropped immediately after. EPOXY induces 1.6% execution-time overhead which is better than BackGuard. Unfortunately, as the other previous proposition, EPOXY is less flexible than BackGuard.

5.7 Conclusion

This chapter proposed a **flexible** defense against memory-based exploits on low-end embedded systems. Low-end microcontrollers are becoming increasingly popular with the rise of the Internet of Things. They are currently integrated into exotic embedded applications ranging from connected bulbs to real-time life-critical devices. Also, low-end microcontrollers are often used as companion processors to relieve main application processors from specific tasks. Unfortunately, unlike powerful processors, the security features provided by low-end microcontrollers are less. Low-end microcontrollers do not benefit from well-known memory safety protection such as the Address Space Layout Randomization (ASLR), the Data Execution Prevention (DEP), and the Stack Smash Protector (SSP). On top of that, these microcontrollers are highly restricted in terms of execution-time, code size, and power consumption which does not ease the integration of security protections. As a result, even the most trivial linear buffer overflow vulnerability can easily be turned into a successful remote code execution exploit.

To improve memory safety, BackGuard proposes a set of compiler plugins that simplifies the integration of memory safety in low-end bare-metal microcontrollers. BackGuard aims at keeping a sufficient trade-off between **security**, **performance**, and **code-size** overhead. Besides, unlike other existing works, BackGuard protection is **flexible**. Indeed, BackGuard extends the Clang/LLVM compiler infrastructure with external plugins that can be loaded within the compilation workflow without requiring any internal invasive compiler modification. As a result, BackGuard is **simple** to integrate into real-world embedded software development processes. Also, BackGuard is **compatible** with Clang/LLVM versions available from official repositories, it does not require any Clang/LLVM recompilation from sources. Finally, the BackGuard plugins operate on the LLVM Intermediate Representation. This feature makes it multi-architecture compatible.

According to the assessments, BackGuard can ensure a reasonable backward-edge control-flow integrity protection with 5% execution-time overhead on average. The main backward-edge control-flow integrity protection provided by BackGuard relies on an updated bitmap where each set bit represents a valid control-flow destination. This backward-edge protection provides an accurate average indirect target reduction close to 99.99% and requires only 3.1% of the available microcontroller memory. Moreover, BackGuard hardens memory exploits development thanks to RO-Guards and Rand-Guards. These protections prevent spatial memory exploits to reach sensitive microcontroller memory regions. Finally, BackGuard efficiently configures a Memory Protection Unit to enforce Data Execution Prevention. At this time, BackGuard does not mitigate against information leakage and temporal memory safety.

Final Note: BackGuard is originally an extension from BackFlow, a compiler-based memory safety protection for low-end ARM microcontrollers. Unlike BackGuard, BackFlow was a compiler back-end extension specific to the ARM Instruction Set Architecture. Two passes of BackFlow were dedicated to building dynamic bitmap protection, and another two passes were dedicated to building static bitmap protection. BackFlow was

accompanied by a set of diversification passes dedicated to embedded system applications. BackFlow has been published as "BackFlow: Backward Edge Control Flow Integrity Enforcement for low-End ARM Microcontrollers" at the IEEE DATE conference 2020. Also, the efficiency of BackFlow has been demonstrated as "BackFlow: Backward Edge Control Flow Integrity Enforcement for low End-ARM Real-Time Systems" at the University Booth Demonstration at IEEE DATE 2020. Finally, the results regarding BackGuard, the improvement of BackFlow has not been published yet. A final BackGuard optimization not included in this thesis will be evaluated before publication to the ACM Transactions on Embedded Computing Systems journal upon submission of this manuscript to the reviewers.

Conclusion

Violating the control-flow integrity of an application using software exploits is at least as broad as the existing protections. During this thesis, I particularly explored the memory safety issue and its impact on wireless critical medical devices (IoMT). Interestingly, despite the numerous efforts spent at mitigating such attacks over the last 30 years, medical devices do not possess even the most basic memory safety defenses.

By reviewing the memory safety defenses state-of-the-art and the context of the Internet of Things (IoT), I exhibit that only a few concepts are suitable for critical and constrained IoMT devices. First, these devices are part of the IoT trend. They are pressured by **time-to-market** and **costs** that often take precedence over the integration of **security**. Besides, it seems that manufacturers of wireless medical devices are **lacking security experts** and **connectivity** opens new **threats** that were not present beforehand. On top of that, medical devices are **constrained** by their **size**, **real-time performances**, and **safety requirements** making the integration of security challenging.

With the awareness of this context, we identified several criteria that should be taken into account when designing practical defenses against memory attacks in critical and constrained embedded systems. In a nutshell, these criteria highlight that memory safety in these systems should be **robust**, **practical**, **safe**, **deterministic** inducing low **performance** degradation and **size overhead**.

From the identified criteria, we developed a complete case study of efficient memory safety integration on critical cyber-physical systems. As a first contribution, we proposed SecPump, a life-critical wireless infusion pump system workbench tailored for security assessments. The aims of SecPump are manifold. Indeed, it allows us to conduct the approach of this thesis as well as to validate it. But also, the model aims to be open to the community allowing security research on cyber-physical systems in various areas that cover both hardware and software.

Then, to protect critical and constrained cyber-physical systems from memory violation attacks, we proposed two top-down approaches that led to several contributions such as TrustFlow and BackGuard. Both approaches tackle the memory safety issue from the software design-time to its run-time on the hardware.

Regarding the thesis approach, TrustFlow provides both a processor and a complete software toolchain that can be used to develop secure cyber-physical applications. TrustFlow proposes an efficient hardware-based secure environment that tracks the integrity of sensitive data in real-time. More precisely, TrustFlow extends the RISC-V Instruction Set Assembly with additional secure instructions and trusted memory. These secure

instructions are used to manipulate sensitive data and protect them using the trusted memory. The latter acts as an enhanced shadow stack able to protect both backward-edges and forward edges while remaining fully compatible with "*setjmp*" and "*longjmp*". Besides, TrustFlow leverage the trusted memory to detect memory violations, log them, and heal the corrupted data in hardware without interrupting the application execution-flow. In other words, TrustFlow is engineered to continue to function correctly even if under memory-based attacks. The results demonstrated that the TrustFlow environment induces a negligible execution time overhead with a reasonable trusted memory depth. At design-time, TrustFlow comes with a practical compiler toolchain. This compiler generates a secure code that enforces instruction separation between sensitive data that use the trusted memory and non-sensitive data that remain in the regular memory.

Unlike TrustFlow, BackGuard assumes that hardware is immutable and memory safety can only be implemented at the software level. Thus, BackGuard only provides a secure software toolchain that can be included in the software development cycle of cyber-physical systems. The aim of BackGuard is to demonstrate that memory defenses are feasible in embedded applications even if the hardware does not enforce specific support. Besides, BackGuard aims at being highly flexible targeting a wide range of Instruction Set Assembly. To do so, BackGuard leverage the LLVM Intermediate Representation to construct a bitmap-based backward-edge control-flow integrity protection coupled with in-memory guards. At compilation-time BackGuard separates sensitive memory objects from unsensitive one using random guards. Besides, BackGuard protects in-memory return addresses using an isolated bitmap. With BackGuard, we proved that practical and efficient memory defenses are feasible for off-the-shelf microcontrollers without relying on invasive hardware modifications. In comparison with TrustFlow, BackGuard is more flexible. As it only relies on the software it provides less security accuracy than TrusFlow. Our measurements showed that BackGuard induces around 5% execution-time overhead but requires at least 5% of a microcontroller memory.

As a result, both approaches seem adapted to secure constrained and critical IoMT devices. They induce low-performance degradation (both execution-time and size), they are robust, deterministic, and practical to use and implement by non-security experts.

Perspectives

This thesis opens new prospects, particularly in the security of constrained and critical embedded systems. Regarding microprocessors, the results of the TrustFlow's trusted memory are encouraging. They demonstrate that a reasonable trusted memory can provide fine-grained control-flow integrity without degrading the execution-time performances. As a direct consequence, we believe that the TrustFlow concept can be implemented in the next generation of low-end microcontrollers dedicated to the IoT. Of course, further research would be required to integrate TrustFlow with a multi-tasking system. In particular, the TrustFlow trusted memory should be optimized to accommodate context switches. We believe that some research in Direct Memory Access (DMA) should be done to quickly save and restore the trusted memory when context switches occur. Also, it would be interesting to study the feasibility of coupling TrustFlow with software isolation mechanisms. Indeed, in the context of a multi-tasking system, it is better to prevent untrusted tasks from using the trusted memory. At the software level, this thesis shows that the implementation of security no longer requires detailed knowledge of the hardware by the developer. For instance, the TrustFlow compiler automates code generation preventing any security implementation mistakes. Of course, the TrustFlow compiler is a proof of concept and may require much more testing and research to refine the definition of critical data and the selection of according secure instructions. Beyond TrustFlow, the use of open architectures such as RISC-V or open-RISC opens new possibilities in all fields of innovation. These open architectures allow customizing processors and System on Chip for specific applications' requirements. Also, fitted with compilers and operating system extensions, these architectures allow designing highly efficient systems.

BackGuard also opens new perspectives regarding the software security of low-end embedded systems. BackGuard demonstrated that memory safety can be integrated within embedded applications even if a microcontroller does not have dedicated hardware memory safety primitives. Although full software security support is less efficient than a combination of both hardware and software, BackGuard stills harden memory exploits, improving the overall security. Besides, BackGuard aims at keeping a sufficient trade-off between security, performance, and code-size overhead. The latter induces around 5% execution-time overhead which is suitable for most existing embedded systems. In closing, BackGuard leverages the LLVM intermediate representation to generate memory safety protected code. This makes BackGuard multi-architecture compliant and flexible. Further research directions plan to study the existing architecture that can be covered by BackGuard.

Finally, the two approaches proposed in this thesis are compatible with each other.

We believe than a hybrid solution between TrustFlow and BackGuard would allow fine-grained memory safety protection while keeping an excellent tradeoff between security, performances, and space overhead. For instance, it would be interesting to study how BackGuard can be coupled with TrustFlow to reduce the trusted memory footprint of an application. On the other hand, the bitmap protection proposed by BackGuard is something that can be implemented at the hardware level. Indeed, the bitmap could be integrated instead of the trusted memory currently implemented in TrustFlow.

At the end of the day, the IoT promises to evolve over the next few years. Future devices may require less energy, less memory consumption, and higher processing capabilities. Simultaneously, new attacks will emerge, opening new research opportunities and in particular for solutions that aim at efficiently combining the hardware along with the software. This reinforces the fact that open architectures have a bright future with many research opportunities.

Bibliography

Journal papers:

C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer, "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," *IEEE Embedded Systems Letters*, 2018.

C. Bresch, D. Hély, S. Chollet, and R. Lysecky, "SecPump: A Connected Open Source Infusion Pump for Security Research Purposes," *IEEE Embedded Systems Letters*, 2020.

C. Bresch, D. Hély, R. Lysecky, S. Chollet, I. Parissis "TrustFlow-X: A Practical Framework for Fine-Grained Control-Flow Integrity in Critical Systems," *ACM Transactions on Embedded Computing Systems (TECS), 2020.*

International Conferences:

C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hély, "A red team blue team approach towards a secure processor design with hardware shadow stack," *2017 2nd International Verification and Security Workshop, IVSW 2017.*

C. Bresch, S. Chollet, and D. Hély, "Towards an inherently secure run-time environment for medical devices," *Proceedings - 2018 IEEE International Congress on Internet of Things, ICIOT 2018 - Part of the 2018 IEEE World Congress on Services.*

C. Bresch, D. Hély, S. Chollet, and I. Parissis, "TrustFlow: A Trusted Memory Support for Data Flow Integrity," *in 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2019.

C. Bresch, D. Hély, R. Lysecky, "BackFlow: Backward Edge Control-Flow Integrity Enforcement for Low-End ARM Microcontrollers," *Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition, DATE 2020.*

International Demonstrations & Workshops:

C. Bresch, "Return-Oriented Programming Attacks on Embedded Linux," *European Forum for Electronic and Components and Systems, Bruxelles, 2018.*

C. Bresch, "Control-Flow Hijacking Attacks on SecPump, a Cyber-Physical System," *The International Cybersecurity Forum, Lille, 2019.*

C. Bresch, D. Hély, R. Lysecky, "BackFlow: Backward Edge Control-Flow Integrity Enforcement forLow End-ARM Real-Time Systems," *University Booth Demonstration at Design, Automation and Test in Europe Conference and Exhibition, DATE 2020.*

Z. Kazemi, C. Bresch, M Fazeli, D. Hély, "A Systematic Approach for Hardware Security Assessment of Secured IoT Applications," *TRUDEVICE Workshop at Design, Automation and Test in Europe Conference and Exhibition, DATE 2020*

Invention Statements:

Invention statement of BackGuard and TrustFlow to Grenoble INP as part of the Linksium out-of-lab challenge. A study of technology transfer or company creation is in progress with the Linksium Grenoble.

Article in submission progress:

C. Bresch, D. Hély, R. Lysecky, "BackGuard: Control-flow Integrity and Binary Hardening for Low-End Microcontrollers," *ACM Transactions on Embedded Computing Systems (TECS), 2020,* (upcoming submission).

E. Aerabi, C Bresch, D Hély, A Papadimitriou, M Fazeli, "CONFISCA: an SIMD-based CONcurrent FI and SCA countermeasure with switchable performance and security modes," (upcoming submission)

Thesis References:

[1] C. Louis, "IoT Market Predicted To Double By 2021, Reaching $520B," *Forbes*, 2018. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/08/16/iot-market-predicted-to-double-by-2021-reaching-520b/#5769f651f948

[2] M. Chiang and T. Zhang, "Fog and IoT: An Overview of Research Opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[3] C. Bresch, S. Chollet, and D. Hély, "Towards an inherently secure run-time environment for medical devices," *Proceedings - 2018 IEEE International Congress on Internet of Things, ICIOT 2018 - Part of the 2018 IEEE World Congress on Services*, pp. 140–147, 2018.

[4] FDA, "How to Determine if Your Product is a Medical Device." [Online]. Available: https://www.fda.gov/medical-devices/classify-your-medical-device/how-determine-if-your-product-medical-device

[5] C. Bisdikian, "An overview of the Bluetooth wireless technology," *IEEE Communications Magazine*, vol. 39, no. 12, pp. 86–94, 2001.

[6] "Faits et chiffres sur le diabète." [Online]. Available: https://www.who.int/diabetes/infographic/fr/

[7] Medtronic, "Minimed Pump 512." [Online]. Available: https://www.medtronicdiabetes.com/download-library/minimed-512-712

[8] A. Melmer, T. Züger, D. M. Lewis, S. Leibrand, C. Stettler, and M. Laimer, "Glycaemic control in individuals with type 1 diabetes using an open source artificial pancreas system (openaps)," *Diabetes, Obesity and Metabolism*, vol. 21, no. 10, pp. 2333–2337, 2019.

[9] H. N. Lily, "Medical Devices Are the Next Security Nightmare," *Wired*, 2017. [Online]. Available: https://www.wired.com/2017/03/medical-devices-next-security-nightmare/

[10] Ventures, "Cybersecurity jobs report," *Herjavec Group*, 2017. [Online]. Available: https://www.herjavecgroup.com/wp-content/uploads/2018/11/HG-and-CV-Cybersecurity-Jobs-Report-2018.pdf

[11] Z. Kim, "Hacker can send fatal dose to hospital drug pumps," *Wired*, vol. 9, 2015. [Online]. Available: https://www.wired.com/2015/06/hackers-can-send-fatal-doses-hospital-drug-pumps/

[12] L. Newman, "A New Pacemaker Hack Puts Malware Directly on the Device," *Wired*, 2018. [Online]. Available: https://www.wired.com/story/pacemaker-hack-malware-black-hat/

[13] H. N. Lily, "These Hackers Made an App That Kills to Prove a Point," *Wired*, 2019. [Online]. Available: https://www.wired.com/story/medtronic-insulin-pump-hack-app/

[14] Z. Kim, "Video Shows a Terrifying Drug Infusion Pump Hack in Action," *Wired*, 2015. [Online]. Available: https://www.wired.com/2015/08/video-shows-terrifying-drug-infusion-pump-hack-action/

[15] "CVE-2017-2853." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-2853

[16] M. Labs, "2017 Threats Predictions," 2016. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-threats-predictions-2017.pdf

[17] "Hospira Multiple Products Buffer Overflow Vulnerability," 2016. [Online]. Available: https://www.us-cert.gov/ics/advisories/ICSA-15-337-02

[18] G. Scott, "Remote Code Execution on the Smiths Medical Medfusion 4000." [Online]. Available: https://github.com/sgayou/medfusion-4000-research/blob/master/doc/README.md

[19] C. Stephen, "IEEE Spectrum - The 2017 Top Programming Languages," 2017. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[20] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.

[21] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[22] A. Griffith, *GCC: The Complete Reference*. McGraw-Hill Osborne Media, 2002.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization, CGO*, no. c, pp. 75–86, 2004.

[24] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.

[25] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[26] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[27] Microsoft, "Threadx real-time operating system," 2018.

[28] ARM, "Mbed OS." [Online]. Available: https://www.mbed.com/en/platform/mbed-os/

[29] R. Barry *et al.*, "Freertos," *Internet, Oct*, 2008.

[30] W. River, "VxWorks." [Online]. Available: https://www.windriver.com/products/vxworks/

[31] L. Rierson, *A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[32] R. Bagnara, A. Bagnara, and P. M. Hill, "The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software," in *International Static Analysis Symposium*. Springer, 2018, pp. 5–23.

[33] K. Arnold, J. Gosling, D. Holmes, and D. Holmes, *The Java programming language*. Addison-wesley Reading, 2000, vol. 2.

[34] G. Rossum, *Python reference manual*. CWI (Centre for Mathematics and Computer Science), 1995.

[35] L. Ablon and A. Bogart, *Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits*. Rand Corporation, 2017.

[36] Mitre, "2019 CWE Top 25 Most Dangerous Software Errors," 2019. [Online]. Available: https://cwe.mitre.org/top25/archive/2019/2019{_}cwe{_}top25.html

[37] "CVE-2017-6862," 2017. [Online]. Available: https://cve.mitre.org

[38] G. J. Holzmann, "The Power of Ten-Rules for Developing Safety Critical Code," *Software Technology: 10 Years of Innovation in IEEE Computer*, pp. 188–201, 2018.

[39] M. Harris, "Open-source medical devices: When code can kill or cure| the economist," *The Economist*, 2012. [Online]. Available: https://www.economist.com/technology-quarterly/2012/06/02/when-code-can-kill-or-cure

[40] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[41] "IEC 62304," Tech. Rep., 2006. [Online]. Available: https://www.iso.org/fr/standard/38421.html

[42] "ISO 14971," Tech. Rep., 2019. [Online]. Available: https://www.iso.org/fr/standard/72704.html

[43] "ISO 13485," Tech. Rep., 2016. [Online]. Available: https://www.iso.org/fr/standard/59752.html

[44] FDA, "Electronic signatures final rule, 21 cfr part 11," *Federal Register, March*, 2000.

[45] C. Ferdinand, R. Heckmann, and B. Franzen, "Static memory and timing analysis of embedded systems code," in *Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March*, 2007, pp. 07–04.

[46] G. J. Holzmann, "The Power of Ten–Rules for Developing Safety Critical Code," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[47] A. One, "Smashing The Stack For Fun And Profit," *Phrack*, vol. 49, 1996.

[48] M. Lad, X. Zhao, B. Zhang, D. Massey, and L. Zhang, "Analysis of bgp update surge during slammer worm attack," in *International Workshop on Distributed Computing*. Springer, 2003, pp. 66–79.

[49] H. Orman, "The morris worm: A fifteen-year perspective," *IEEE Security & Privacy*, vol. 1, no. 5, pp. 35–43, 2003.

[50] H. Shacham, M. Page, B. Pfaff, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[51] P. Wagle and C. Cowa, "Stackguard: Simple stack smash protection for gcc," *Proceedings of the GCC Developers Summit*, pp. 243–255, 2003.

[52] J. Edge, "Strong stack protection for gcc," *Linux Weekly News*, 2014.

[53] 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*, 2010, pp. 559–572.

[54] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," *Proceedings - 4th IEEE European Symposium on Security & Privacy, EURO S and P 2019*, pp. 31–46, 2019.

[55] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Good motive but bad design: Why ARM MPU has become an outcast in embedded systems," *arXiv preprint arXiv:1908.03638*.

[56] A. Francillon and C. Castelluccia, "Code Injection Attacks on Harvard-Architecture Devices," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 15–26.

[57] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, 2011, pp. 121–141.

[58] Nergal, "The advanced return-into-lib(c) exploits (PaX case study)," *Phrack*, vol. 58. [Online]. Available: http://phrack.org/issues/58/4.html{#}article

[59] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[60] T. Bletsch, X. Jiang, and V. Freeh, "Jump-Oriented Programming : A New Class of Code-Reuse Attack ," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2010, pp. 30–40.

[61] 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.

[62] F. Lindner, "Cisco IOS Router Exploitation," *BlackHat USA*, 2009.

[63] A. Francillon and C. Castelluccia, "Tinyrng: A cryptographic random number generator for wireless sensors network nodes," in *2007 5th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops.* IEEE, 2007, pp. 1–7.

[64] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, R. K. Iyer, and W. M. Street, "Non-Control-Data Attacks Are Realistic Threats," in *USENIX Security Symposium*, vol. 5, 2005, pp. 177–191.

[65] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming : On the Expressiveness of Non-Control Data Attacks," *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 969–986, 2016.

[66] I. Evans, F. Long, and H. Shrobe, "Control Jujutsu : On the Weaknesses of Fine-Grained Control Flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[67] B. Sun, C. Xu, and S. Zhu, "The power of data-oriented attacks: Bypassing memory mitigation using data-only exploitation," *Black Hat Asia*, 2017.

[68] M. Abadi, M. Budiu, and U. Erlingsson, "Control-Flow Integrity Principles, Implementations, Applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, p. 4, 2009.

[69] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[70] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 133, no. 11, pp. 1–11, 2019.

[71] R. de Clercq and I. Verbauwhede, "A survey of Hardware-based Control Flow Integrity (CFI)," *arXiv preprint arXiv:1706.07257*, vol. 1, pp. 1–27, 2017. [Online]. Available: http://arxiv.org/abs/1706.07257

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

[73] A. Edwards, H. Vo, and A. Srivastava, "Vulcan binary transformation in a distributed environment," 2001.

[74] J. L. Henning, "SPEC CPU2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[75] Intel, "Control-flow Enforcement Technology Specification," Tech. Rep., 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[76] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-r. Sadeghi, and Y. Jin, "Strategy Without Tactics : Policy-Agnostic Hardware-Enhanced Control-Flow Integrity," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[77] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones." in *NDSS*, vol. 26, 2012, pp. 27–40.

[78] N. Christoulakis, G. Christou, and E. Athanasopoulos, "HCFI : Hardware-enforced Control-Flow Integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.

[79] M. Alam, D. B. Roy, S. Bhattacharya, and V. Govindan, "SmashClean : A Hardware level mitigation to stack smashing attacks in OpenRISC," *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, no. November, 2016.

[80] M. Kayaalp, S. Member, M. Ozsoy, and S. Member, "Efficiently Securing Systems from Code Reuse Attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, 2014.

[81] T. N. Vijaykumar, C. E. Brodley, A. Jalote, W. Lafayette, and B. A. Kuperman, "SmashGuard : A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1271–1285, 2006.

[82] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[83] S. Das, W. Zhang, and Y. Liu, "A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 11, pp. 3193 – 3207, 2016.

[84] W. He, S. Das, W. Zhang, and Y. Liu, "No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries," *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 23, 2017.

[85] Y. Wilander, John; Nikiforakis, Nick; Youan and E. Al., "RIPE: runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 41–50.

[86] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-FLow ATtestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 743–754. [Online]. Available: http://arxiv.org/abs/1605.07763

[87] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[88] B. Wijnen, E. Hunt, G. Anzalone, and Pearce JM, "Open-Source Syringe Pump Library," *PLoS ONE*, vol. 9, no. 9, p. e107216, 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0107216

[89] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware," *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 24.

[90] A. Waterman and K. Asanovic, "The risc-v instruction set manual; volume ii: Privileged architecture, sifive inc. and cs division, eecs department," *University of California, Berkeley*, vol. 1, 2017.

[91] "Pulpino. An Open-Source Microcontroller System based on RISC-V." [Online]. Available: https://github.com/pulp-platform/pulpino

[92] C. Tice, T. Roeder, P. Collingbourne, L. Lozano, S. Checkoway, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[93] C. Zhang, T. Wei, Z. Chen, and L. Duan, "Practical Control Flow Integrity & Randomization for Binary Executables," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[94] J. Salwan, "ROPgadget." [Online]. Available: https://github.com/JonathanSalwan/ROPgadget

[95] M. Polychronakis and A. D. Keromytis, "ROP Payload Detection Using Speculative Code Execution," in *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 2011, pp. 58–65.

[96] X. Wang and J. Backer, "SIGDROP : Signature-based ROP Detection using Hardware Performance Counters," *arXiv preprint arXiv:1609.02667*, 2016.

[97] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," Tech. Rep., 2012.

[98] G. Hunt and D. Brubacher, "Detours: Binary interception of win 32 functions," in *3rd usenix windows nt symposium*, 1999.

[99] I. Fratrić, "ROPGuard : Runtime Prevention of Return-Oriented Programming Attacks," Tech. Rep., 2012.

[100] Y. Cheng, Z. Zongwei, and M. Yu, "ROPecker : A Generic and Practical Approach For Defending Against ROP Attack," in *NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium*, 2014, pp. 1–14.

[101] M. Kayaalp, T. Schmitt, and J. Nomani, "SCRAP : Architecture for Signature-Based Protection from Code Reuse Attacks," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 258–269.

[102] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazi, and D. Boneh, "Hacking Blind," in *In 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 227–242.

[103] L. Liu and D. Zha, "Launching Return-Oriented Programming Attacks against Randomized Relocatable Executables," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications.*, 2011, pp. 37–44.

[104] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK : Automated Software Diversity," in *In 2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–294.

[105] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the Software Stack Using Randomized NOP Insertion," *Springer, New York, NY*, pp. 151–173, 2013.

[106] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.

[107] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[108] C. Kil and J. Jun, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[109] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.

[110] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, and W. C. Rd, "Binary Stirring : Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[111] J. Gionta and W. Enck, "HideM : Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities Categories and Subject Descriptors," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 325–336.

[112] R. D. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, and P. Maene, "SOFIA : Software and Control Flow Integrity Architecture," *Computers & Security*, vol. 68, pp. 16–35, 2017.

[113] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, "Cryptographically Enforced Control Flow Integrity," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 941–951.

[114] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, 2003, pp. 91–104.

[115] S. Bhatkar and R. Sekar, "Data Space Randomization," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Berlin, Heidelberg, 2008, pp. 1–22.

[116] S. MCPEAK and S. P. NECULA, George C. RAHUL, "CIL: Intermediate language and tools for C program analysis and transformation." in *Conference on Compiler Construction*, 2002, pp. 213–228.

[117] H. Liljestrand, T. Nyman, K. Wang, and C. C. Perez, "PAC it up : Towards Pointer Integrity using ARM Pointer Authentication ," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.

[118] Qualcomm Security, "Pointer Authentication on ARMv8 . 3," 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[119] R. 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," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, pp. 4–44, 2017. [Online]. Available: http://tosc.iacr.org/index.php/ToSC/article/view/583

[120] O. Oleksenko and D. Kuvaiskii, "Intel MPX Explained," *arXiv preprint arXiv:1702.00719*, 2017.

[121] S. Nagarakatte, M. Martin, and S. A. Zdancewic, "Watchdog : Hardware for Safe and Secure Manual Memory Management and Full Memory Safety," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, no. June, 2012, pp. 189–200.

[122] S. Nagarakatte and M. M. K. Martin, "WatchdogLite : Hardware-Accelerated Compiler-Based Pointer Checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 175.

[123] S. Nagarakatte and M. Martin, "SoftBound : Highly Compatible and Complete Spatial Memory Safety for C," *ACM Sigplan Notices*, vol. 44, no. January, pp. 245–258, 2009.

[124] J. Devietti, C. Blundell, M. Martin, and S. A. Zdancewic, "HardBound : Architectural Support for Spatial Safety of the C Programming Language," in *ACM SIGARCH Computer Architecture News.*, 2008, pp. 103–114.

[125] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T : A RISC-V Processor with Light Weight Security Extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, 2017, no. June, pp. 1–8.

[126] R. Wahbe, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," *Proceedings of the fourteenth ACM symposium on Operating systems principles*, no. 203-216, 1993.

[127] V. Kuznetsov, L. Szekeres, and M. et al Payer, "Code-Pointer Integrity," *11th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 147–163, 2014.

[128] I. Evans and S. Fingeret, "Missing the Point(er): On the Effectiveness of Code Pointer Integrity," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 781–796.

[129] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," *OSDI 2006 - 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006.

[130] P. Akritidis, C. Cadar, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *2008 IEEE Symposium on Security and Privacy*. IEEE, 2008, pp. 263–277.

[131] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI : Hardware-Assisted Data-flow Isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 1–17.

[132] W. Song, A. Bradbury, and R. Mullins, "Towards general purpose tagged memory," in *Proceedings of the RISC-V Workshop*, 2015.

[133] E. Bosman and H. Bos, "Framing signals - A return to portable shellcode," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 243–258, 2014.

[134] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, "Cyclone: A type-safe dialect of C," *C/C++ Users Journal*, vol. 23, no. 1, pp. 112–139, 2005.

[135] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, 2005.

[136] K. Bernsmed, M. G. Jaatun, and P. H. Meland, "Safety critical software and security-how low can you go?" in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE, 2018, pp. 1–6.

[137] A. Kuehn and M. Mueller, "Analyzing bug bounty programs: An institutional perspective on the economics of software vulnerabilities," 2014.

[138] J. Radcliffe, "Hacking medical devices for fun and insulin: Breaking the human scada system," in *Black Hat Conference presentation slides*, vol. 2011, 2011.

[139] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky, "Formal methods based development of a pca infusion pump reference model: Generic infusion pump (gip) project," in *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007)*. IEEE, 2007, pp. 23–33.

[140] M. E. Fisher, "A Semiclosed-Loop Algorithm for the Control of Blood Glucose Levels in Diabetics," *IEEE Transactions on Biomedical Engineering*, vol. 38, no. 1, pp. 57–61, 1991.

[141] A. Roy and R. S. Parker, "Mathematical Models of the Metabolic System in Health and in Diabetes: Dynamic Modeling of Exercise Effects on Plasma Glucose and Insulin Levels," *Journal of diabetes science and technology (Online)*, no. 3, p. 338.

[142] R. N. Bergman, "Lilly lecture 1989. Toward physiological understanding of glucose tolerance. Minimal-model approach," *Diabetes*, vol. 38, no. 12, pp. 1512–1527, 1989.

[143] E. Friis-jensen, "Modeling and Simulation of Glucose-Insulin Metabolism," Ph.D. dissertation, 2007.

[144] D. Care and S. S. Suppl, "2. Classification and Diagnosis of Diabetes: Standards of Medical Care in Diabetes-2020," *Diabetes care*, vol. 43, no. January, pp. S14–S31, 2020.

[145] SiFive, "SiFive RISC-V port." [Online]. Available: https://github.com/sifive/

[146] E. ANSSI, "Ebios-expression des besoins et identification des objectifs de sécurité," 2016.

[147] Z. Kazemi, A. Papadimitriou, D. Hely, M. Fazcli, and V. Beroulle, "Hardware security evaluation platform for mcu-based connected devices: Application to healthcare IoT," *2018 IEEE 3rd International Verification and Security Workshop, IVSW 2018*, pp. 87–92, 2018.

[148] E. Aerabi, M. Bohlouli, M. A. Livany, M. Fazeli, A. Papadimitriou, and D. Hely, "Design Space Exploration for Ultra-Low Energy and Secure IoT MCUs," *IACR Cryptology ePrint Archive*, vol. 1, no. 1, 2020.

[149] C. Bresch, D. Hely, S. Chollet, and R. Lysecky, "SecPump: A Connected Open Source Infusion Pump for Security Research Purposes," *IEEE Embedded Systems Letters*, vol. 0663, no. c, pp. 1–1, 2020.

[150] S. Gal-on and M. Levy, "Exploring CoreMark™ - A Benchmark Maximizing Simplicity and Efficacy," *The Embedded Microprocessor Benchmark Consortium (EEMBC)*, 2012. [Online]. Available: www.eembc.org

[151] M. Muench, J. Stijohann, and F. Kargl, "What You Corrupt Is Not What You Crash : Challenges in Fuzzing Embedded Devices," *NDSS*, 2018.

[152] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[153] K. Pagiamtzis, S. Member, A. Sheikholeslami, and S. Member, "Content-Addressable Memory (CAM) Circuits and Architectures : A Tutorial and Survey," *IEEE journal of solid-state circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[154] A. Limaye and T. Adegbija, "HERMIT : A Benchmark Suite for the Internet of Medical Things," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 4212–4222, 2018.

[155] ARM, "Mbed TLS," 2015. [Online]. Available: https://tls.mbed.org/

[156] A. De, A. Basu, S. Ghosh, and T. Jaeger, "FIXER: Flow Integrity Extensions for Embedded RISC-V," *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019*, pp. 348–353, 2019.

[157] C. Bresch, D. Hély, S. Chollet, and I. Parissis, "TrustFlow : A Trusted Memory Support for Data Flow Integrity," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019.

[158] 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," *2017 2nd International Verification and Security Workshop, IVSW 2017*, pp. 57–62, 2017.

[159] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer, "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 87–90, 2018.

[160] "TrustFlow-X: A Practical Framework for Fine-Grained Control Flow Integrity in Critical Systems," *ACM Transactions on Embedded Computing Systems*, 2020.

[161] J. Wetzels and A. Abbasi, "Ghost in the machine: Challenges in embedded binary security," *Usenix Enigma*, 2017.

[162] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[163] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," *arXiv preprint arXiv:1308.5174*, 2013. [Online]. Available: http://arxiv.org/abs/1308.5174

[164] O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. I. Sidorov, and A. A. Timorin, "Industrial control systems vulnerabilities statistics," *Kaspersky Lab, Report*, 2016.

[165] Microsoft. Control Flow Guard. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[166] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," pp. 161–176, 2015.

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

[168] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient Protected Shadow Stacks on Embedded Systems," 2019. [Online]. Available: http://arxiv.org/abs/1910.12157

[169] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2020.

[170] J.-L. Zhang, G. Qu, Y.-Q. Lv, and Q. Zhou, "A survey on silicon pufs and recent advances in ring oscillator pufs," *Journal of computer science and technology*, vol. 29, no. 4, pp. 664–678, 2014.

[171] D. Holcomb, "Randomness in integrated circuits with applications in device identification and random number generation," 2007.

[172] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[173] R. Earnshaw, "Procedure call standard for the arm architecture," *ARM Limited, October*, 2003.

List of Figures

List of Tables

A Annex 1 : TrustFlow pipeline

A.1 Trusted memory integration

The functioning of the TrustFlow extension is explained in Chapter 4, section 4.3.1. This appendix gives little more details about the TrustFlow extension, and the changes performed within the Rocket Chip processor generator [152]. All TrustFlow extensions are implemented within a Rocket core (Tile), an in-order RISC-V CPU. A Tile, such as the one displayed in Figure A.1 consist of several components such as the L1 instructions and data caches, a Rocket core (processor pipeline), an FPU, and a coprocessor interface such as RoCC.

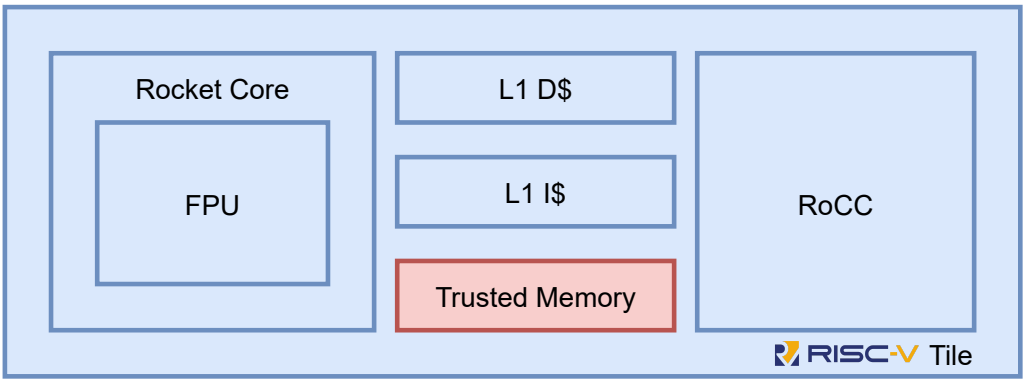

Figure A.1: Rocket Tile

The trusted memory support of TrusFlow interfaces with the Rocket core pipeline. To remind, the latter is a content-addressable memory that performs input address search against a table of stored address/data vectors. Regarding memory safety, this trusted memory can be compared to an improved shadow call stack by mostly protecting sensitive control-flow data. The trusted memory is driven by two custom instructions added to the RISC-V Instruction Set Assembly (ISA). These two instructions such as "load word secure" (*lws*) and "store word secure" (*sws*) inherit from the classical load and store instructions of the RISC-V instruction set. However, these instructions trigger the TrustFlow extension so that it stores and checks the sensitive data. An overview of the extended Rocket core is displayed in Figure A.2. The diagram is simplified, it does not include all the signals and components of the pipeline, but only the relevant information for the understanding of the TrusFlow extension.

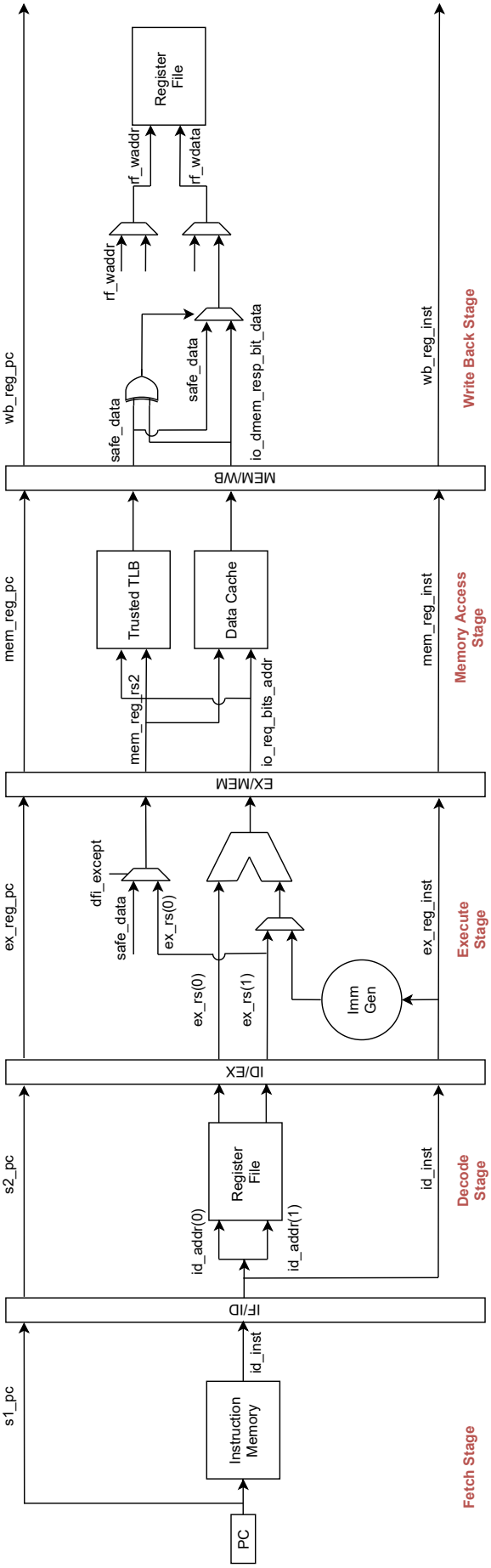

Figure A.2: TrustFlow pipeline

TrustFlow particularly extends the memory access stage and the write-back stage of the Rocket core pipeline. During the memory stage, the data cache is accessed by a memory access instruction and the result is then forwarded to the write-back stage which updates the register file.

During the memory access stage, the TrustFlow extension decodes the custom instructions handled in “mem_reg_inst”. If custom instructions appear to be valid at this stage of the pipeline, the TrustFlow extension retrieves the address of the data that should be stored or loaded in regular memory from the output of the ALU (“io_req_bits_addr”). Furthermore, in the case of the custom store instruction, the data to duplicate in the trusted memory is handled in “mem_reg_rs2”. Both “mem_reg_inst” and “io_req_bits_addr” triggers the TrustFlow trusted TLB controller, which is detailed further in this annex.

During the write-back stage, the TrustFlow extension operates in consistency with the custom instruction. Concerning custom loads ("*lws*"), the output of the trusted memory (“safe_data”) fetched from the memory access stage is compared with the output of the data cache (“io_dmem_resp_bit_data”). In case of discrepancy, a data-flow violation is detected. In our proof of concept, TrustFlow proposes two ways to deal with this violation. The first one generates a hardware fault stopping the execution flow of the exploited application. The other way is that the violation triggers the self-healing mechanism of TrustFlow that replaces the corrupted data by the healthy data and logs the violation in a hidden register.

Regarding custom stores ("*sws*"), the TrustFlow extension waits until the custom store instruction is valid at the end of the pipeline to perform the final commit of the sensitive data in the trusted memory. Indeed, if the custom store is not valid at the end of the pipeline, it means that the data storage in the main memory has not been carried out correctly. Thus the trusted memory is synchronized according to this result.

A.2 Trusted memory controller

The TrustFlow's trusted memory extension is driven by a controller that follows the state machine displayed in Figure A.3. Both sws_req and lws_req events are "*sws*" and "*lws*" instruction requests performed during the memory access stage. Besides, both sws_ valid and lws_valid events are valid "*sws*" and "*lws*" instructions in the write-back stage of the pipeline. It turns out that the transitions of the finite state machine take place according to four different events spread over two stages (memory access and write-back stage).

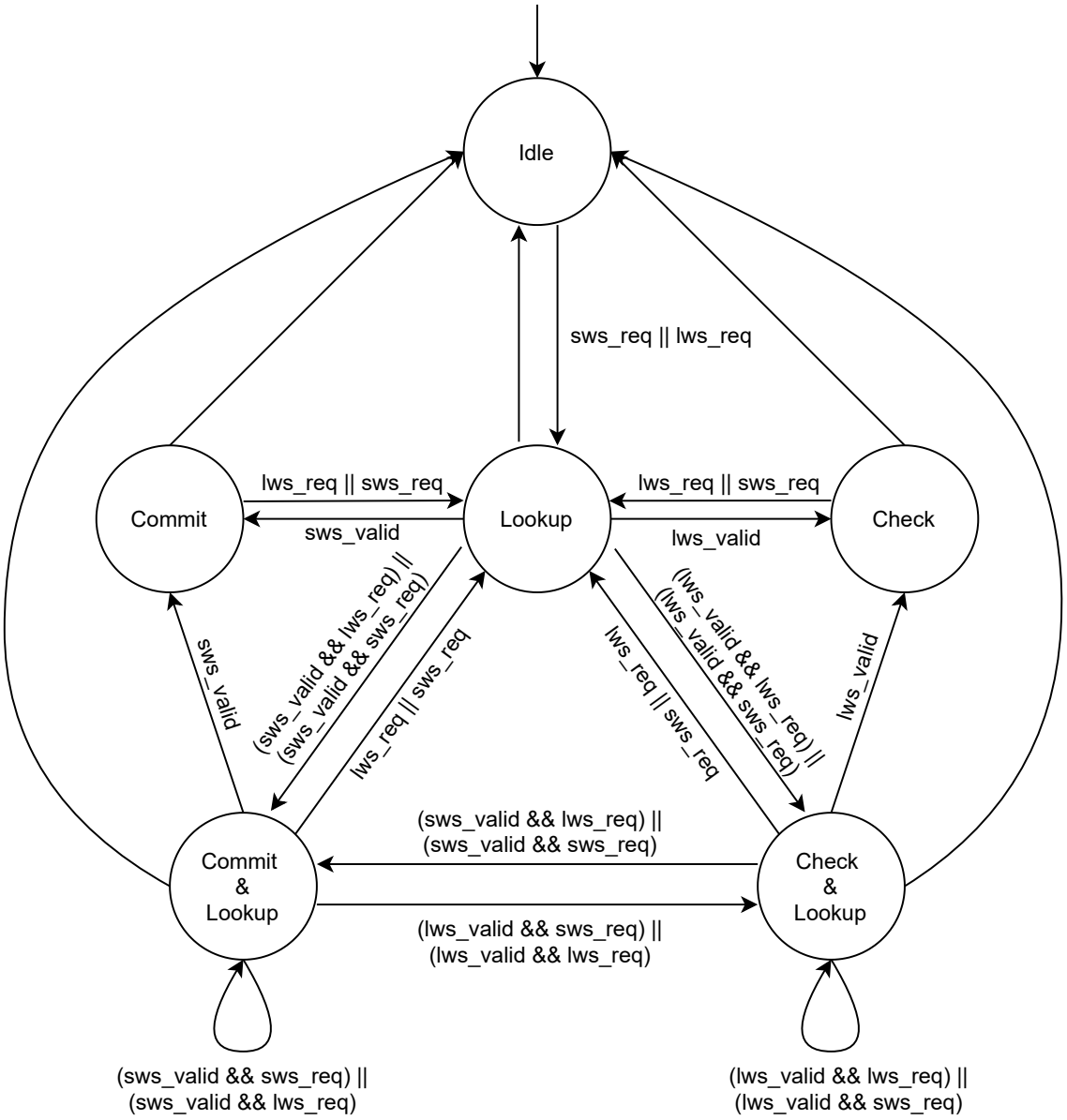

Figure A.3: Trusted memory controller state-machine

The finite state machine in Figure A.3 handles six states. The operations are performed on the transition.

- **Idle**: The controller waits for a valid custom instruction to be decoded in the pipeline.
- **Lookup**: TrustFlow performs a lookup table in the trusted memory.

  - In the case of "*sws*" instruction, the lookup table determines the index in the TLB where the sensitive data/address vector should be stored. If the data/address vector is already in the trusted memory the lookup table returns its index. If the data/address vector is not in the trusted memory the lookup table returns a free entry.
  - In the case of "*lws*" instruction, the matching data is forwarded to the output of the trusted memory.

- **Commit**: TrustFlow commits a data in the trusted memory at the index determined by the lookup table operation.
- **Check**: TrustFlow performs a check between the output of the trusted memory and the output of the regular memory. In case of discrepancy, TrustFlow either stops the executed program or heals the corrupted data.
- **Commit & Lookup**: This state is reached when two custom instructions follow each other in the pipeline. In such case TrustFlow performs both a **Commit** and a **Lookup** operation.
- **Check & Lookup**: This state is reached when two custom instructions follow each other in the pipeline. In such case TrustFlow performs enables a **Check** and a **Lookup** operation.

A.3 Data restoration

Introducing two custom instructions and violated data replacement in the Rocket core requires to adjust the datapath accordingly. As for regular instructions, the custom instructions are subject to data hazards, stalls, and forwarding issues. For instance, when an instruction reads a register following a load that defines the same register. Figure A.4 displays a non-handled data hazard.

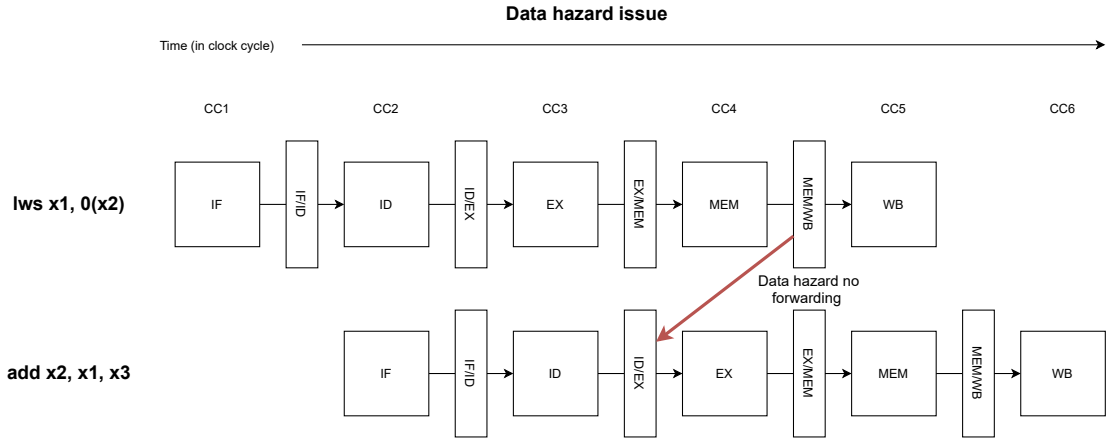

Figure A.4: Data hazard example

In TrustFlow and like in regular RISC pipeline, the hazard detection unit introduces a stall to delay the instruction that follows the custom load. Then, the forwarding unit forwards the loaded memory data back in the pipeline. Figure A.5 displays the stall and forward case handled by TrustFlow.

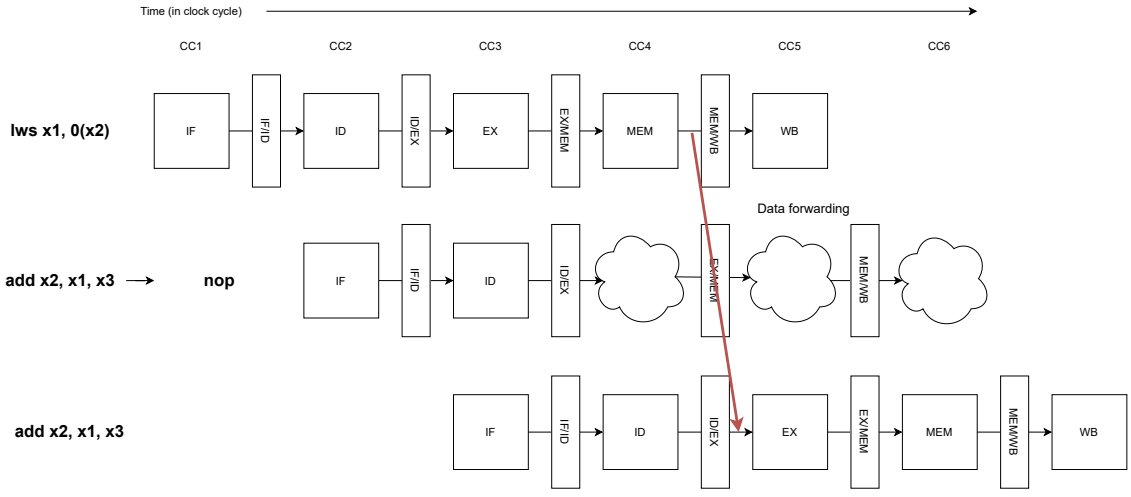

Figure A.5: Data hazard resolution

Of course, both Figure A.4 and Figure A.5 display a specific example of a data hazard happening in the TrustFlow pipeline. We currently modified the forwarding unit and the hazard detection unit to handle custom instructions. Also, when a data violation is detected within the pipeline, the modified datapath takes care of forwarding the healed data. More simply the modified forwarding unit of the TrustFlow pipeline implements the following conditions for the load hazards:

MEM/WB forwarding 1:
if (MEM/WB.Rd // if MEM/WB writes a register
and not (EX/MEM.Rd // and if MEM/WB is not writing the same register as EX/MEM (e.g Rs1 or Rs2)
and ((EX/MEM.Rd == ID/EX.Rs1) or (EX/MEM.Rd == ID/EX.Rs2)))
and ((MEM/WB.Rd == ID/EX.Rs1) or (MEM/WB.Rd == ID/EX.Rs2))) // and if ID/EX uses an operand defined by MEM/WB
    // if data healthy
    if not (dfi_except) Forward MEM/WB
    // if data violation
    if (dfi_except) Forward safe_data

MEM/WB forwarding 2:
// Case especially for stores that follows loads
if (MEM/WB.Rd // if MEM/WB writes a register
and ((MEM/WB.Rd == EX/MEM.Rs1) or (MEM/WB.Rd == EX/MEM.Rs2))) // and if EX/MEM uses an operand defined by MEM/WB
    // if data healthy
    if not (dfi_except) Forward MEM/WB
    // if data violation
    if (dfi_except) Forward safe_data

B

Annexe 2 : RISC-V prologue and epilogue insertion

This annex details how the RISC-V LLVM compiler backend generates a secure prologue and a secure epilogue. The prologue and epilogue inserter pass arrives late in the target code generator of the compiler. According to Figure B.1 that displays the various passes executed in order by the back-end, the prologue and epilogue code insertion happens after the register allocation pass. Indeed, to determine the registers that should be spilled in a stack frame, the prologue and epilogue code insertion pass should be aware of the registers used by a callee. Thus, the prologue and epilogue inserter pass is a machine function pass that operates at the function level. The insertion of the prologue and epilogue mostly involves stack unwinding, finalizing the function layout, saving the callee-saved registers, and then emitting the prologue and epilogue code.

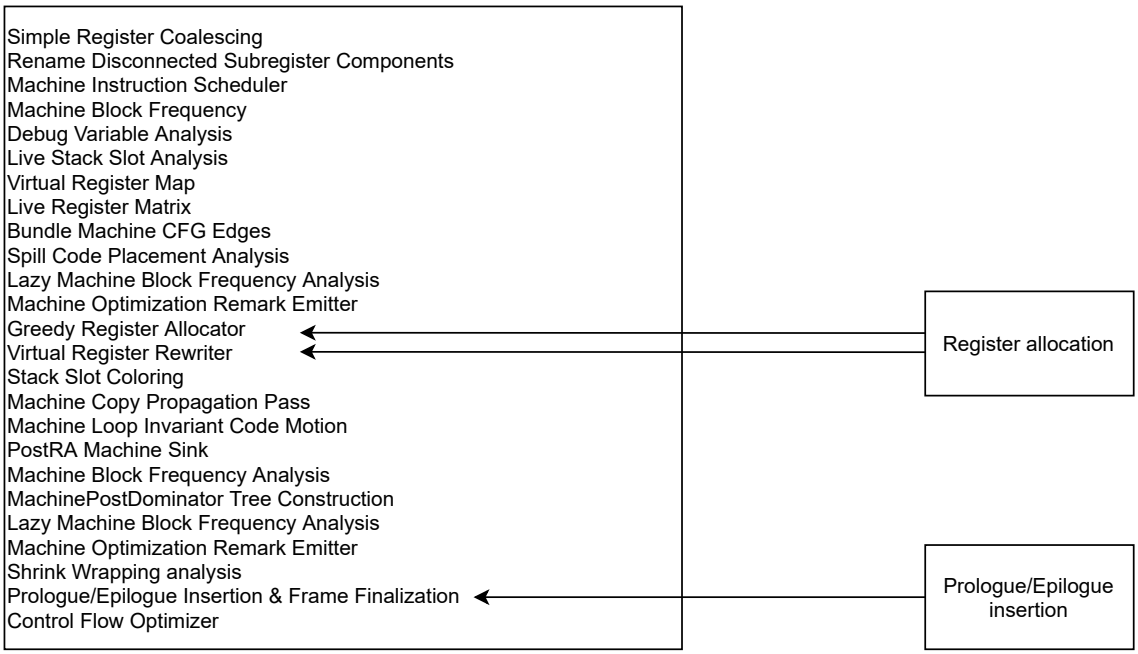

Figure B.1: LLVM RISC-V back-end pipeline

The prologue and epilogue insertion pass is located in the target-dependent code generator of the back-end. More specifically, the "PrologEpiloginserter.cpp" file is responsible for handling the Prologue and Epilogue Insertion (PEI). PEI operates on "Machine Functions". As displayed in Figure B.1, it calls the "SpillCalleeSavedRegs" function that determines which registers in the callee should be saved. It also assigns a stack slot for any

called registers. Finally, the function that inserts the code for the callee-saved registers used in a function is achieved by "insertCSRSaves". This specific function either call the "spillCalleeSavedRegisters" function (if implemented in the back-end) or the "storeRegToStackSlot" to spill the caller registers. The "storeRegToStackSlot" is a regular function that is usually used in various passes of the back-end to insert load and/or store instruction to spill a register in a stack frame. This function is not specifically dedicated to the PEI.

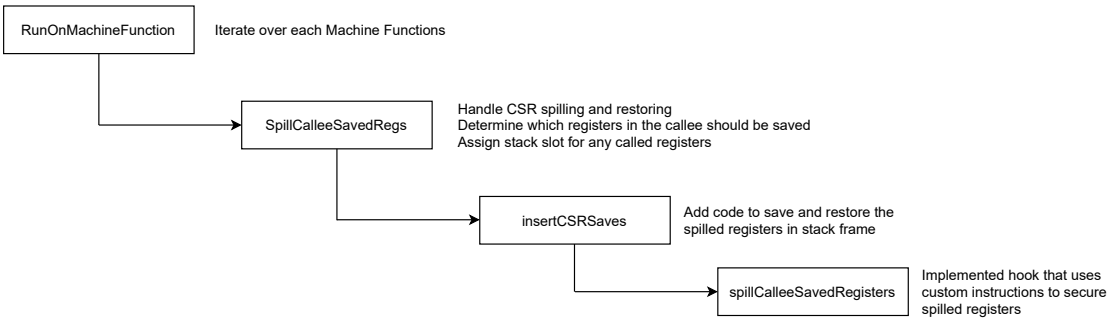

Figure B.2: LLVM PEI pass

The version of LLVM used during the thesis was still a testing version not pushed on the official stream. At that time, the RISC-V back-end used the current "storeRegToStackSlot" function to store spilled registers in a stack frame and "loadRegFromStackSlot" to restore the spilled registers from the memory. One strategy would have been to modify this function so that it uses custom instructions to secure the memory loads. Unfortunately, this function is not only used by the prologue and epilogue inserted in the back-end. Modifying it could induce hazards in code generation. For this reason, we extended the RISC-V backend so that it uses the predefined "spillCalleeSavedRegisters" and "restoreCalleeSavedRegisters" functions. These two functions are only used for prologue and epilogue generation and, depending on the security level (explained in section 4.6), secure the spilled registers with custom instructions.

Figure B.3 displays the RISC-V assembly code generated by the TrustFlow LLVM Back-end according to security options.

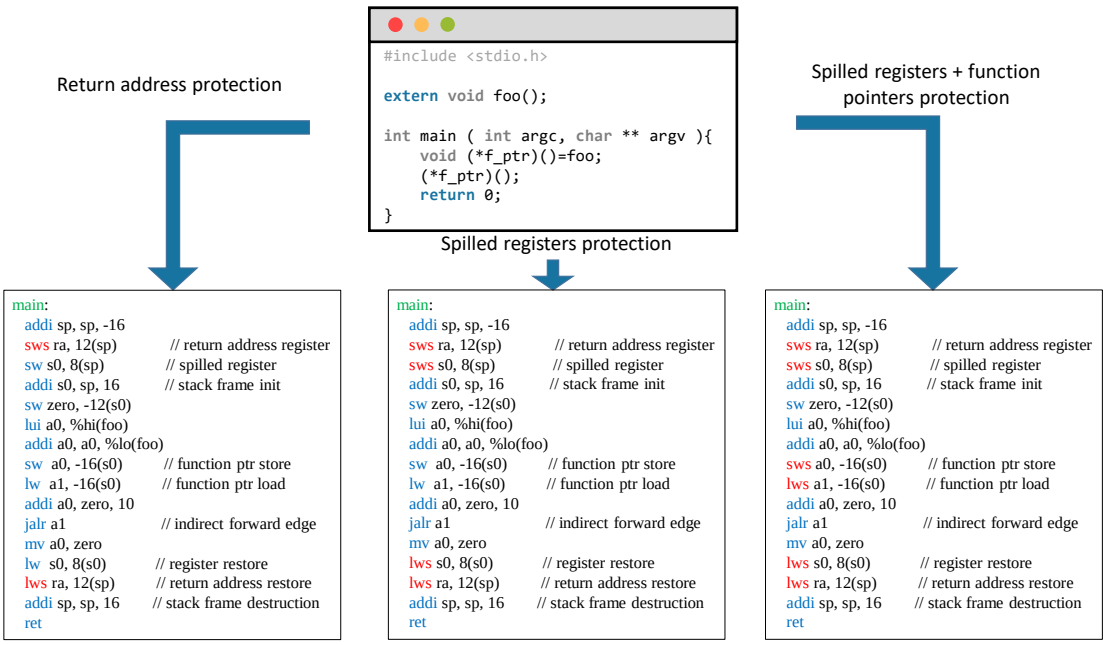

Figure B.3: TrustFlow generated Code

C Annexe 3 : LLVM ARM backend bitmap pass

This annex details the implementation of the ARM machine bitmap pass which is part of the BackGuard compiler Framework. In the study regarding BackGuard, this pass is specifically developed to observe the impact on the performances between flexible security operated on the LLVM IR and target-specific security implementation. The details of the BackGuard compiler extension at the LLVM IR level are provided in Chapter 5, section 5.3.2. The bitmap protection variant implemented within the ARM Backend allows the BackGuard framework to benefit from a better execution-time overhead than the LLVM IR bitmap protection. In reference to section 5.4.2.2, the obtained results reveal that the execution time overhead induced by the LLVM machine bitmap pass is 50% lower than the LLVM IR protection. This is explained by the fact that the LLVM machine bitmap pass can perform architecture dependant optimizations by directly working on the assembly before code emission.

To better understand the functioning of the LLVM backend bitmap pass, this annex requires several essential points regarding the ARM Cortex-M Instruction Set Assembly (ISA) and its calling convention. The ARM Cortex-M microprocessors follow a Reduced Instruction Set Computing (RISC) design. They are 32-bit processors that provide sixteen general-purpose registers (from r0 to r15) and 32-bit aligned instructions. Besides, these microprocessors provide a sub-instruction set called the Thumb ISA. The Thumb instruction set only uses a subset of the general registers (r0-r7), and instructions are 16 bits long. Most of the current Cortex-M processors use the Thumb 2 instruction set, which is a mix between 32-bit and 16-bit instructions. The main registers involved in the ARM calling convention are summarized in Table C.1 bellow.

Table C.1: ARM Core Registers

Purpose in the procedure call standard	**Register**	**Special**
Program counter	R15	PC
Link Register	R14	LR
Stack Pointer	R13	SP
Variable Registers	R4-R7	
Scratch Registers	R0-R3	

The procedure call standard [173] for the ARM architecture details how subroutines are called, how they return, and finally how the registers in Table C.1 are involved. Both “bl” (branch and link) and “blx” (branch and link exchange) instructions are used to call

a subroutine. These instructions pass the execution flow to a subroutine and put the caller return address pointer in the link register (LR). In a program, each function has a prologue and an epilogue. Function prologue and epilogue have the effect of setting/destroying a new stack frame and spilling/restoring the caller registers. To ensure a function return, we denote two types of instruction patterns that can be generated by the compiler. If the return address is previously spilled from the link register to a stack frame during the prologue, the ARM compiler uses a multiple load instruction to both places the return address in the program counter register and restore the spilled registers. On the contrary, if a return address handled in the link register is never spilled, the ARM compiler uses a branch exchange ("bx") instruction to branch to the return destination target.

Following the calling convention of the ARM instruction set, it follows that every function that saves the link register in a stack frame should be secured using the bitmap. On the other hand, functions that keep return addresses in the link register are not vulnerable to memory attacks. In order to protect functions that spill the link register, one challenge is to modify the way they are generated by the compiler. During compilation, and especially in the back-end, the compiler is aware of which functions spill its link register. These functions are called leaf function (e.g functions that make no calls). From this analysis, the compiler can generate a specific prologue and epilogue that registers/checks the link register using the bitmap.

The machine bitmap pass modifies the ARM calling convention. More precisely the LLVM ARM backend prologue and epilogue generation passes are modified and a custom machine passes in introduced within the compilation workflow. The resulting code generated by the compiler is displayed in Figure C.1.

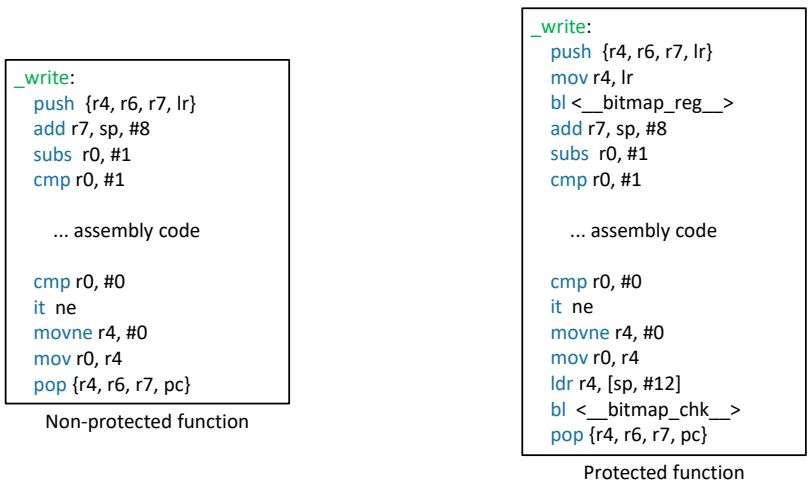

Figure C.1: BackGuard Machine pass generated Code

The "___bitmap_reg___" routine is a couple of instructions that set the link register in the bitmap using the r4 register as an argument. Likewise, the "___bitmap_chk___" routine is used to check the integrity of the spilled link register using the bitmap. Both jump tables are constructed by the compiler backend during code generation. We decided to use the variable register r4 as an argument for the assembly tables. Indeed, at the beginning/end of each function, the r4 register is never live. Consequently, this register can be used to perform operations without inducing any hazards in the following

instruction flow of the function. Naturally, the r4 register can be used by a called function. The latter is thus both saved and restored by the modified compiler prologue and epilogue. Choosing to use the r4 saves the spill of several registers.

During the function epilogue, the compiler machine passes leverage the LLVM stack frame info to retrieve the position of the saved link register in the local stack frame. Then, the compiler generates an instruction to load the return address from the stack frame to the r4 register. Once again this register is used by the jump table to verify and de-register the return address.

The machine bitmap pass is implemented immediately after prologue and epilogue insertion and before the second instruction scheduling pass. More specifically, the machine bitmap pass is implemented just before the ITBlockPass. This pass is an optimization that leverages the IT instruction to specify assembly condition code. As some function returns are conditional the ITBlockPass includes these return instructions within an IT block. We leverage this feature to include the bitmap checks within IT blocks so that bitmap checks are conditional if returns are conditional as well.

As a final note, we also implemented a "Machine bitmap pass" that directly includes the tables' instructions within functions' prologue and epilogue. This implementation required heavy modifications to the backend, and unfortunately, we came to the conclusion that this had the effect of considerably increasing the size of an application for very little gain in execution time.

Approches, Stratégies, et Implémentations de Protections Mémoire dans les Systèmes Embarqués Critiques et Contraints.

Approaches, Strategies, and Implementations of Memory Safety Defenses in Critical and Constrained Embedded Systems.

Résumé

Cette thèse traite de la problématique des corruptions de mémoire dans les dispositifs médicaux vitaux. Au cours des dernières années, plusieurs vulnérabilités telles que les exploits de mémoire ont été identifiées dans divers dispositifs connectés de l'Internet des objets médicaux (IoMT). Dans le pire des cas, ces vulnérabilités permettent à un attaquant de forcer à distance une application à exécuter des actions malveillantes. Si de nombreuses contre-mesures contre les exploits logiciels ont été proposées jusqu'à présent, seules quelques-unes d'entre elles semblent convenir aux dispositifs médicaux. En effet, ces dispositifs sont contraints de par leur taille, leurs performances en temps réel et les exigences de sûreté de fonctionnement, ce qui rend l'intégration de la sécurité difficile. Pour répondre à ce problème, la thèse propose deux approches. Toutes deux abordent la question de la sécurité de la mémoire depuis la conception du logiciel jusqu'à son exécution sur le matériel. Une première approche suppose que les défenses peuvent être mises en œuvre à la fois dans le matériel et dans le logiciel. Cette approche aboutit à TrustFlow, une structure composée d'un compilateur capable de générer un code sécurisé pour un processeur modifié. Ce processeur peut prévenir, détecter, enregistrer et auto-guérir les applications critiques victimes d'une attaque mémoire. La seconde approche considère que le matériel est immuable. Selon cette contrainte, les défenses ne reposent que sur le logiciel. Cette seconde approche aboutit à BackGuard, un compilateur modifié qui renforce efficacement les applications embarquées tout en assurant l'intégrité du flot d'exécution.

Mots-clés : IoMT, sécurité, corruption de mémoire, intégrité du flot d'exécution.

Abstract

This thesis deals with the memory safety issue in life-critical medical devices. Over the last few years, several vulnerabilities such as memory exploits have been identified in various Internet of Medical Things (IoMT) devices. In the worst case, such vulnerabilities allow an attacker to remotely force an application to execute malicious actions. While many countermeasures against software exploits have been proposed so far, only a few of them seem to be suitable for medical devices. Indeed, these devices are constrained by their size, real-time performances, and safety requirements making the integration of security challenging. To address this issue, the thesis proposes two approaches. Both address the memory safety issue from the software design-time to its run-time on the hardware. A first approach assumes that memory defenses can be implemented both in hardware and software. This approach results in TrustFlow, a framework composed of a compiler able to generate secure code for an extended processor that can prevent, detect, log, and self-heal critical applications from memory attacks. The second approach considers that hardware is immutable. Following this constraint, defenses only rely upon software. This second approach results in BackGuard a modified compiler that efficiently hardens embedded applications while ensuring control-flow integrity.

Keywords : IoMT, security, memory corruption, control-flow integrity