

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Systems Engineering (35^{th}cycle)

Vulnerability-Tolerant Architectures for Resource-Constrained Devices

By

Gianluca Roascio

******

Supervisor(s):
Prof. Paolo Prinetto, Supervisor

Doctoral Examination Committee:
Prof. Alessandro Armando, Referee, Università degli Studi di Genova
Prof. Giorgio Di Natale, Referee, Université Grenoble Alpes
Prof. Alessandro Cilardo, Università degli Studi di Napoli Federico II
Prof. Gabriele Costa, Scuola IMT Alti Studi Lucca
Prof. Matteo Sonza Reorda, Politecnico di Torino

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Gianluca Roascio
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

A Pietro e Paolo, i quali incontrandomi come persona comune hanno scorto, tra i miei tanti vizi da mortale, le virtù per le quali oggi mi trovo a firmare questo lavoro.

Acknowledgements

Il percorso che si conclude con la stesura di questa tesi sarebbe stato impossibile senza il prezioso supporto delle tante persone che, per le ragioni più diverse, sono venute in contatto con me e con la mia attività. Un sentito ringraziamento va a tutti gli studenti che, grazie al loro lavoro di tesi, hanno portato mattoni nel loro complesso fondamentali per costruire l'edificio che metaforicamente rappresenta questa dissertazione. Un ringraziamento speciale va a Nicolò Maunero, insostituibile compagno di viaggio con il quale ho condiviso ogni giorno del mio percorso, con le sue gioie e le sue ansie, e al quale devo molti dei risultati qui raggiunti. Altri ringraziamenti speciali vanno a Gabriele Costa per l'indispensabile aiuto offerto per la formalizzazione della teoria qui presentata, ma anche per aver pazientemente seguito per anni il mio lavoro all'interno del progetto SPARTA, e per aver usato molto del suo tempo per donarmi consigli e buone pratiche su come diventare un ricercatore e un professionista migliore. Un grande ringraziamento va anche a Mahboobe Sadeghipour, per il singolare aiuto offerto nel non semplice lavoro di implementazione tecnica delle ideazioni, e per il suo ruolo di compagna fedele in questo ultimo tratto del duro percorso affrontato.

Per concludere, infinite grazie vanno al Professor Paolo Prinetto, l'uomo che in definitiva ha reso possibile tutto questo, comportandosi da vera guida professionale e spirituale oltre che scientifica. È stata la tua visione, Paolo, a non farmi mai dubitare delle scelte intraprese, e ad insegnarmi a tenere sempre gli occhi fissi sui grandi obiettivi anziché farsi distrarre dalle storture dei dettagli. Sono convinto che non ci sarà situazione, nella mia futura carriera, in cui non sia possibile far tesoro di ciò che mi hai trasmesso.

Abstract

Over the past three decades, advancement and spread of digital technology has enabled an unprecedented transformation of our society. We are gradually moving on to the almost complete digitization of all administrations, advanced industrial automation, mass distribution of personal devices, connection of homes to the Internet, and in very recent times, to the production of *smart things*, i.e., objects of our everyday lives equipped with a computing core, including wearable devices, means of transportation, household appliances, and even buildings themselves. Each of these systems produces and processes sensitive data, which may affect our business, our privacy, and even our safety, or controls processes critical to us as individuals, but also as a community.

Given this background, it is impossible not to scientifically question how to protect this huge amount of data and functions. While knowledge regarding the security of general-purpose devices is in a more advanced state, there is still a very open question on these special-purpose devices. Many of the solutions studied or implemented in the general-purpose domain cannot be supported by embedded systems for several reasons, including their specificity and the absence of base elements such as standard operating systems or architectures. Plus, for obvious low-consumption reasons, the resources available to these devices are typically single-thread CPUs, that do not exceed very few hundreds of MHz in frequencies, with RAM for code and data contained below few MB, and negligible or absent secondary storage.

For optimization reasons, the code executed by these devices is mostly written in low-level languages, such as C or Assembly, or more rarely C++. By allowing the programmer free use of memory, such languages open the way for a wide class of vulnerabilities, which allow an attacker to corrupt data and program execution. These attacks are commonly referred to as *binary attacks*. Among these, particularly

dangerous are those that redirects the execution flow to malicious instructions, injected or already present elsewhere in memory: this is what is usually referred to as *Arbitrary Code Execution* (ACE).

Ensuring *software integrity*, meant as immutability of data and code with respect to the original programmer intent, is the ideal limit solution for defeating binary attacks. With respect to ACE issues, this is declined into the so-called *Control-Flow Integrity* (CFI). By such a property, no branch within the program can reach destinations not originally allowed by its intended flow. CFI requirements rise the demand to design secure hardware architectures that, by natively supporting software integrity extensions, are naturally *resilient* to vulnerabilities in executed programs. In a nutshell, this is the concept of *Vulnerability-Tolerant Architectures*.

The present thesis first details and explains the considered security philosophy, together with fundamental concepts in the field. Then, an overview of two domains to be intersected is given, namely control-flow attacks and embedded systems. Then, the principles of a new theory for secure design policies that ensures both integrity and schedulability of the execution are presented, in the context of a framework named PROLEPSIS. Such a framework contains formal methods and demonstrations for modeling the issue and proving the validity of the protection strategy. Together with the theory, possible implementations using both legacy and soft processors are offered, with related experimental data. Also, methods for vulnerability discovery and binary instrumentation are presented, and a new strategy for devising security benchmarks.

Given the importance of the aforementioned security issues in the embedded domain, an Appendix is also provided with a research work conducted in the field of hardware security education and training, with results obtained from the implementation of platforms allowing hands-on experience for students and cybersecurity professionals.

Contents

List of Figures **x**

List of Tables **xiii**

Summary of Contents and Achieved Results **1**

List of Related Publications **5**

1 Introduction **8**

1.1 Fundamental Semantics of Cybersecurity 8

1.2 Vulnerabilities 13

1.3 The Role of Hardware in Cybersecurity 19

1.4 Zero-Trust Model and Security-by-Design 24

1.5 Vulnerability-Tolerant Architectures 27

2 Embedded Systems and Security **33**

2.1 The Embedded Systems Domain 33

2.2 Security Issues Overview 39

3 Control-Flow Attacks: State of Knowledge **45**

3.1 Binary Attacks and Memory Corruption 45

3.2 Code Injection and Code Reuse 51

3.3 Classical Defenses 59
3.4 Control-Flow Integrity 63
3.4.1 Software-based Control-Flow Integrity 71
3.4.2 Hardware-based Control-Flow Integrity 73
3.4.3 Abstraction Layer 79
3.4.4 Policy 81
3.4.5 Instrumentation 81
3.4.6 Granularity 83
3.4.7 Commercial Off-the-Shelf Solutions 85

4 Control-Flow Integrity for Embedded Devices 89
4.1 Attacker Perspectives 89
4.2 Issues and Challenges 93
4.3 State-of-the-Art Discussion 101

5 Fine-Grained, Real-Time, and Interrupt-Aware CFI: A New Formal Model 108
5.1 ToyRISC Language 111
5.1.1 Syntax 111
5.1.2 Program Semantics 113
5.1.3 Interrupt Semantics 115
5.2 CFI Definition 116
5.3 CFI Enforcement 119
5.3.1 Vulnerable Branch Detection 119
5.3.2 Reference Monitor 122
5.3.3 Program Instrumentation 129
5.3.4 Correctness 130
5.4 Technical Proofs 131

6 Design for Security Approaches and Experimental Assessment 136

6.1 External Monitor 138

6.2 Internal Monitor 148

6.3 Binary Analysis and Instrumentation 155

6.4 Security Evaluation 167

6.5 Performance Evaluation 174

7 Conclusions 188

A Hardware Security Education and Training 194

A.1 Introduction 194

A.2 Hardware-Based Capture-the-Flag Challenges 198

A.3 Cyber Ranges 207

A.4 The PAIDEUSIS Hybrid Cyber Range 214

A.5 HArMoNICS: Cyber Range for Intelligent Infrastructures 229

Bibliography 236

List of Figures

1.1 The CIA triad. 12

1.2 Characterization of attacks. 13

1.3 Characterization of vulnerabilities. 15

1.4 Layerized view of a computing system. 20

1.5 The triple role of hardware in cybersecurity. 23

1.6 The dependability threat chain. 28

1.7 The dependability threat chain declined to security context. 29

1.8 The dependability tree [31]. 29

2.1 Example of embedded SoC internal architecture. 37

2.2 Schematic diagram of a smart lock system application. 41

3.1 The call stack. 52

3.2 The call stack after a memory corruption targeting the return address. 53

3.3 Frame of the `find_file()` function. 53

3.4 Frame of the `find_file()` function after corruption. The return address now points to the corrupted area containing the shellcode. . 55

3.5 Consequences of a *Return-into-libc* attack into the stack. 56

3.6 Return-Oriented Programming. 58

3.7 The stack canary. 60

3.8 The shadow call stack. 61

3.9 A possible slice of a control-flow graph. 64

3.10 Example of an x86 indirect branch to be instrumented, with its destination. 65

3.11 Example of an instrumented x86 indirect branch and destination. . . 66

3.12 Categorization of control-flow integrity solutions. 70

3.13 Tracking of CFG in Sullivan *et al.* solution [137]. 77

4.1 RECFISH's trampoline-based instrumentation [121]. 102

4.2 View of the tasks running on TrackOS. The monitoring task is represented as overlapping TrackOS to represent the privileged mode it runs in [108]. 103

5.1 Forward Branch Policy (FBP) enforcement. 124

5.2 Backward Branch Policy (BBP) enforcement. 125

5.3 Interrupt handler policy enforcement. 127

6.1 The pin-to-pin connection of the CPU-FPGA interaction. 140

6.2 The structural hardware view of the FPGA-based CFI monitor. . . . 142

6.3 RISC-V base instruction formats showing immediate variants. . . . 148

6.4 Control-Flow Address Stack (CFAS) schematics. 150

6.5 Control-Flow Logic Verifier (CFLV) schematics. 152

6.6 Stages of the PROLEPSIS analysis and instrumentation tool. 158

6.7 Data listing conversion within PROLEPSIS instrumentation tool. . . 161

6.8 Forward and backward edges in JSON format. 166

A.1 Topics and typology distribution from a sample of 40 challenges marked as "hardware" in CTFTime.org (percentages are integer-rounded). 203

A.2 Taxonomy of cyber ranges [208]. 209

A.3 PAIDEUSIS internal organization. 216

A.4 PAIDEUSIS topology. 217

A.5 User interface of the Digital Hardware Emulation Range. 223

A.6 The Xilinx PYNQ-Z1 board. 224

A.7 The ChipWhisperer. 227

A.8 Screenshots from Jupyter Notebooks performing offline side-channel analysis. 228

List of Tables

4.1 Summary of analysed solutions and their coverage of the main issues. 107

5.1 ToyRISC Assembly language syntax. 111

5.2 Small step semantics of ToyRISC. 114

5.3 Formal semantics of reference monitor. 128

6.1 ARMv7-M ISA branch instructions, with corresponding ToyRISC abstraction. 139

6.2 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 1). 144

6.3 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 2). 145

6.4 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 3). 146

6.5 ARMv7-M instruction latencies for instructions used in the instrumentation [176]. 147

6.6 RV32IM ISA branch instructions, with corresponding ToyRISC abstraction. 149

6.7 CFI Extensions to RV32IM ISA. 153

6.8 RV32IM machine instruction translation for PIP and PEP to the internal PDP. 154

6.9 Security evaluation of PROLEPSIS using Em-RIPE's 105 attack forms, compared to common compiler-based protections. 174

6.10 Protection rates in the external monitor implementation measured over Embench-IoT. 178

6.11 Code size in the external monitor implementation measured over Embench-IoT, before and after the instrumentation process. 179

6.12 Runtime in the external monitor implementation measured over Embench-IoT, before and after the instrumentation process. 180

6.13 Lattice MachXO2 FPGA resource usage in the external monitor implementation. 181

6.14 State of the art comparison of CFI solutions for ARM microcontrollers. 181

6.15 Protection rates in the internal monitor implementation measured over Embench-IoT. 183

6.16 Code size in the internal monitor implementation measured over Embench-IoT, before and after the instrumentation process. 184

6.17 Runtime in the internal monitor implementation measured over Embench-IoT, before and after the instrumentation process. 186

6.18 Xilinx XC7A100T FPGA resource usage in the internal monitor implementation. 187

6.19 State of the art comparison of CFI solutions for soft cores. 187

Summary of Contents and Achieved Results

The research work described in this dissertation is composed of elements from a path that began almost four years ago, investigating cybersecurity from a *foundation perspective*: device security is no longer seen as an ancillary feature that can be added, but as a foundational element of systems, which must already be designed to respond to an external threat, that is always assumed to exist. Based on my research group's and my department's extensive experience and tradition on the design of hardware and computing architectures, the work has delved into the broad (but recent) strand combining *security* with *hardware*. This intermingling generates two different facets, which nevertheless tend toward a single end: the design of hardware that is *defended* against attacks that directly target its physical and electronic nature, and the design of hardware that is *defending*, i.e., by virtue of its security also enables the protection of the upper abstraction layers of computing systems, such as software and communications.

Hardware, like software, data, or communication infrastructure, must be designed, built, tested, used, and maintained while considering potential cyber attacks and their consequences. Indeed, hardware runs software and constitutes *the last line of defense*: if the hardware is compromised, any mechanism introduced to secure may be rendered ineffective. Unprotected hardware can be a weak point in the chain, providing at the same time an easy and powerful access point to the system functions and data.

The present research path, borrowing a well-established term in the culture of hardware design, investigates principles and methods of the design of *vulnerability-tolerant architectures*, i.e., platforms designed to ensure system security even in the presence of a vulnerability found, for example, in the software being executed.

The concept traces the already well-known concept of *fault tolerance*, whereby the presence of a hardware fault does not cause the system to fail. In this dissertation, the concept takes shape from a major software security problem for its prevalence and danger, namely the presence of *memory corruption vulnerabilities*, which enable the attacker to force malicious executions on the victim system, sometimes without even the need to inject code.

The target domain is *embedded systems*, i.e., the small electronic controllers that are hidden in the objects that surround our daily lives. Here, the effects of the dangerousness of such attacks are manifested to an even greater extent, as the hardware is often designed in such a way as to exclusively respond to functional needs, and with policies of maximum resource saving, leaving apart those features that could constitute a safeguard against attacks. Likewise, the code executed by such systems is largely written in languages, such as C or Assembly, which allow a deep optimization of resources, but at the same time open the way for the corruption vulnerabilities mentioned above. The goal is then to create sensitivity and knowledge about secure hardware design, particularly to address control flow corruption vulnerabilities: hardware architectures must assume such weaknesses as inescapably present in the software they will eventually run in their life cycle, and be able to block penetration attempts by construction.

The scientific contribution of this research work can be split in serveral parts. First, an important point of view on low-level computer security is introduced, with definitions and taxonomies that can be of special help in the hardware security field. Then, a formal theory for modeling and devising formal security solutions for embedded software is presented: the community can benefit from it in different ways, by just taking the reference formal model or by adopting, extending and implementing the identified security models. In fact, the core of the work is the presentation of PROLEPSIS, a theoretical and practical framework for devising vulnerability-tolerant architectures for embedded systems, that guarantee control-flow integrity which is simultaneously *fine-grained*, *real-time* and *interrupt-aware*. With respect to what has been elaborated in the control-flow integrity research field, PROLEPSIS advances the knowledge importing a model which is completely abstract from practical details, such as the used computer architecture or monitor technology, while offering a maximum coverage against any code redirection, and granting minimal overhead on runtime, memory size and hardware resource occupancy. The main achievement of PROLEPSIS is the idea of detecting the *minimum number*

of insecure branches, i.e., those truly at risk of being corrupted by an attack, and mandates checks only for those, avoiding support for a complete CFG verification, which would be too heavy for the reference domain. The framework operates no *branch regulation* on the protected code, leaving the nature of the program as untouched, but places at a lower level, inserting *instrumentation-based checks* that are transparent to the application and the users. Instrumentations are only needed to inform the monitor of the minimum information required, whereas actual checks are performed in parallel to the main execution. PROLEPSIS involves *no secret*, and does not mandate any *architectural support* by the platform on which it is applied.

Chapter 1 presents the general philosophy of cybersecurity in which the research work fits, giving the definitions of the basic principles accepted by the subsequent work. The concepts of *danger*, *threat*, *security*, *attack* and *vulnerability* are formalized and categorized. Next, the role of hardware in the cybersecurity landscape is analyzed, and seen as threefold: the concepts and coverage areas of *Hardware Security*, *Hardware-based Security* and *Hardware Trust* are defined. The *zero-trust model*, taken as fundamental to this study, is then introduced, according to which the external threat is to be considered certain and constant. In response to this condition, the concept of *security-by-design* is presented, as a philosophy according to which security is to be accounted from the earliest stages of the development of a computing system. Finally, the Chapter closes with a discussion of the concept of vulnerability-tolerant architecture.

In the remainder of the thesis, Chapter 2 provides background elements related to the computing domain this study is aimed at, i.e., embedded systems, giving their common characteristics and data on their use and deployment, and analyzing their main security implications. Chapter 3 goes into depth regarding the technicalities of a family of software security issues, i.e., *control-flow attacks*, which aim to use vulnerabilities present in software to redirect program flow to injected or present but unintended areas of code in order to force malicious execution. The classical defenses that are traditionally taken against these attacks are detailed, and the concept of *control-flow integrity* is introduced as a security property to ensure the prohibition of such redirections. All major proposals from the literature and the commercial sector are analyzed and categorized. In Chapter 4, the issues and the ways to merge control-flow integrity principles and the specificities of embedded systems are discussed, through the identification of critical and fundamental points for the reference domain and the design of vulnerability-tolerant systems that guarantee by-design

the above property. Chapter 5 introduces the PROLEPSIS framework, for the design of *control-flow-integral* architectures, consisting of algorithms, methodologies, software tools, and hardware design techniques. The Chapter introduces the formal theory behind the framework, and offers mathematical proofs to the theorems presented. In Chapter 6, the formal findings are corroborated by the real-world implementation of hardware security extensions, for two different architectures with internal and external security monitors with respect to the processor, respectively. In addition, tools for static analysis and empirical evaluation of the protection level are presented. All experimental data obtained from all implementations are offered, and the compliance of these with the previously presented theory is demonstrated. Finally, in Chapter 7, conclusions on the study are drawn.

A final corollary part of the dissertation describes the research work done on a topic which is discussed centrally in the thesis, i.e., on *Education and Training* of Hardware Security. The importance of the topic stems from our experience with the main topics covered in this thesis, and the particular relevance of secure hardware design practices on which to train the next generation of experts. The Appendix details and shows the results of a study carried out in order to face the challenge of integrating this topic within the practical training paths specialized on cybersecurity, also in the context of relevant projects in the Italian and European contexts.

List of Related Publications

The research work carried out along the previous three years and dissected within this thesis has led, in order of increasing time, to the following publications, also mentioned in the remainder of the text at the passages where the specific work is examined:

1. Maunero, N., Prinetto, P., & Roascio, G. (2019). CFI: Control Flow Integrity or Control Flow Interruption? In *2019 IEEE East-West Design & Test Symposium (EWDTS)* (pp. 1–6). http://dx.doi.org/10.1109/EWDTS.2019.8884464

2. Maunero, N., Prinetto, P., Roascio, G., & Varriale, A. (2020). A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems. In *2020 IEEE 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)* (pp. 1–10). http://dx.doi.org/10.1109/DTIS48698.2020.9081314

3. Prinetto, P., & Roascio, G. (2020). Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy. In *ITASEC - Italian Conference on Cybersecurity* (pp. 177–189). Retrieved from http://ceur-ws.org/Vol-2597/paper-16.pdf

4. Prinetto, P., Roascio, G., & Varriale, A. (2020). Hardware-based Capture-The-Flag Challenges. In *2020 IEEE East-West Design & Test Symposium (EWDTS)* (pp. 1–8). http://dx.doi.org/10.1109/EWDTS50664.2020.9224932

5. Berra, G., Ferraro, G., Fornero, M., Maunero, N., Prinetto, P., & Roascio, G. (2021). PAIDEUSIS: A Remote Hybrid Cyber Range for Hardware, Network, and IoT Security Training. In *ITASEC - Italian Conference on Cybersecurity* (pp. 284–297). Retrieved from http://ceur-ws.org/Vol-2940/paper24.pdf

6. Forte, V., Maunero, N., Prinetto, P., & Roascio, G. (2021). PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity. In *2021 IEEE International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)* (pp. 1–6). http://dx.doi.org/10.1109/ICECCME52200.2021.9591080

7. Rajabalipanah, M., Roodsari, M. S., Jahanpeima, Z., Roascio, G., Prinetto, P., & Navabi, Z. (2021). AFTAB: A RISC-V Implementation with Configurable Gateways for Security. In *2021 IEEE East-West Design & Test Symposium (EWDTS)* (pp. 1–6). http://dx.doi.org/10.1109/EWDTS52692.2021.9580979

8. Eftekhari Moghadam, V., Prinetto, P., & Roascio, G. (2022). Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems. In *2022 IEEE European Test Symposium (ETS)* (pp. 1–4). http://dx.doi.org/10.1109/ETS54262.2022.9810441

9. Fanti, A., Perez, C. C., Denis-Courmont, R., Roascio, G., & Ekberg, J.-E. (2022). Towards Register Spilling Security using LLVM and ARM Pointer Authentication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1–1. http://dx.doi.org/10.1109/TCAD.2022.3197511

10. Roascio, G., Cerini, S. Y., & Prinetto, P. (2022). Remotizing and Virtualizing Chips and Circuits for Hardware-based Capture-the-Flag Challenges. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (pp. 477–485). http://dx.doi.org/10.1109/EuroSPW55150.2022.00057

11. Roascio, G. and Costa, G. and Baccelli, E. and Malina, L. and Matulevičius, R. and Momeu, M. and Morkevičius, N. and Russo, E. and Stojanović, B. and Tasidou, A. (2022). HArMoNICS: High-Assurance Microgrid Network Infrastructure Case Study. *IEEE Access*, 1–1. http://dx.doi.org/10.1109/ACCESS.2022.3218412

12. Roascio, G., Serra, G., & Eftekhari Moghadam, V. (2022). Em-RIPE: Runtime Intrusion Prevention Evaluator for ARM Microcontroller Systems. In *2022 International Conference on Electrical, Computer, Communications and*

Mechatronics Engineering (ICECCME) (pp. 1–6) http://dx.doi.org/10.1109/ICECCME55909.2022.9988527

Chapter 1

Introduction

This introductory Chapter is intended to provide the reader with the semantics of the main general concepts relating to cybersecurity, according to a formulation preliminary to the study of the specific topics under discussion in this thesis. The purpose is to contextualize the present work within a philosophical system of reference, which served as a guide for the subsequent research. What is here reported is an original content, that takes into account (but renews) what is universally accepted in the reference scientific field and presented in the literature. Although not all of the concepts defined here are actually functional for understanding the specific research work described in the following Chapters, it is nonetheless worth reporting all what has been elaborated by the writer and his school of thought, as in our humble opinion, this is intended to signify an important cue for innovation in the educational and research fields, in the more general context of computer security, and more particularly of hardware and embedded systems security. The meaning of the concepts here clarified is implied throughout the entire dissertation.

1.1 Fundamental Semantics of Cybersecurity

The Oxford English Dictionary, the leading historical dictionary of the English language published by Oxford University Press, in its 2021 edition gives the following definition as the first item in the entry *security*: "*objective condition of freedom from danger, or warranty against possible danger.*" This definition is very valuable,

as it immediately clarifies a dual nature of the concept: the absence of danger, or equivalently the guarantee that it will not materialize, even if present.

But what is a *danger*? The dictionary itself defines it as "*the possibility of something happening that is capable of causing injury or harm.*". Danger is thus an absolutely impersonal concept, determined by a force that - we know - can also be natural, such as a weather agent, or an object in a precarious condition, etc. This may be already sufficient to define many kinds of security: think of security in workplaces, or environmental security. However, the kind of security that is going to be treated in this thesis needs an additional element, i.e., the *willingness to benefit from a dangerous situation*. Such is the *threat*, defined by the same dictionary as the overt intention by someone to cause that injury or harm encapsulated in the definition of danger.

Thus, we have already concatenated three keywords for the purposes of our discussion: (i) threat, as the intention to actualize a danger, (ii) danger, as the possibility of suffering injury or harm, and (iii) security, as the absence of or guarantee against such danger. To close the circle, there is no left but defining what we mean by *injury or harm*. The definition of these concepts is in fact what creates the different declinations of security, and what gives the term *security* its own contextualized meaning. For this, we refer to a definition of security given by NIST, in one of its 2016 reports [1]: "*the freedom from those conditions that can cause loss of assets with unacceptable consequences.*". By superimposing this definition on the previous one, we find that injury can be corresponding to the deterioration or loss of an asset in such a way that its consequences cannot be accepted. As can be seen, the characterization is now all about what is considered an *asset*, and what are the consequences resulting from an injury to it, for which the security measure is evaluated.

The domain we refer to is that of information processing systems, more commonly referred to as *computers*. Such systems can be broadly divided into two major sub-domains: Information Technology (IT) and Operational Technology (OT). IT refers to anything related to computing technology, including storage, recovery, transmission, manipulation, and protection of data. OT can be instead defined as "*the hardware and software that detects or causes a change, through the direct monitoring and/or control of industrial equipment, assets, processes and events.*" [2]. In this context, as well as in another, an *asset* is an item to which stakeholders associate a value relative to what they consider as business or mission of the system. Therefore,

it may be a *tangible* item, such as a piece of hardware, a computing platform, a network device, but also *intangible*, such as information, software, copyright, intellectual property, image, or reputation. Assets may also be not directly internal to computing systems, but elements of the environment the system operates within: the environment itself, its ecosystem, its natural balance, etc., or human beings themselves, their health, their physical integrity, etc.

To better circumscribe what we are going to discuss, it is necessary to make an initial distinction here. When the concept of security refers to dangers to people or the environment, the correct word to use is *safety*. What is more properly called *computer security*, or *cybersecurity*, refers to dangers related to the technological assets of the systems themselves, including the examples given above. Technological assets can be *information* assets (such as files, keys, credentials, registries, databases, etc.) or *property* assets (such as offered services, functionalities, resources such as memory and CPU time, etc., but also physical components, since when operating within a system they define one of its properties).

But what is meant by injury or harm toward IT and OT systems? A harm can be defined as something that deteriorates or even nullifies the value of the system. Since the value of all assets within the system is decided - as we have said - according to what the stakeholder attributes to it, we can say that an injury or harm is constituted by an act that is contrary to what the stakeholder originally had in mind for a particular asset, and contrary to her/his will with respect to the purpose of that asset; or by an act that falls outside the use cases envisioned by the stakeholder or the specifications of the system, which represent the transcript of her/his direct or indirect intention. It comes naturally to think that the stakeholder desires that asset to operate in a certain way, and especially that its interaction with the outside world take place in a *regulated* way, i.e., corresponding to her/his intention and/or intended use cases. It is equally natural, therefore, to assume that the stakeholder would consider as dangerous an action that is outside of said rules for interaction, and would regard as a threat the intention to interact with the asset outside of those rules, or to force the asset to operate outside of those rules. As can be noticed, the word *rule* is central to what we are naturally associating with the concept of security. In fact, the rule is what establishes a boundary of legitimacy, between allowed and forbidden actions. Any system that implements a certain *security policy* for its assets materializes this boundary, and first places any entity likely to interact with the assets *outside* of it. The pass to cross that boundary is what we call *authorization*. We can therefore

conclude that injury or harm in this context is the consequence of an unauthorized action on an asset.

It is now necessary to characterize unauthorized actions on assets. If we try to imagine all the actions that may cause injury or harm to the system by affecting information processed or property owned by it, we end up reducing all these actions into three broad categories. The first category of actions are those carried out by entities that, while not being authorized, aim to take possession of a piece of information or a property. These actions can be well described with the verb *steal*. The second category of actions are those carried out by entities that, while not being authorized, nevertheless change a piece of information or a property. These actions are well described by the verb *corrupt*. The third category of actions is that of actions carried out to prevent that information or property from being used. These actions are well indicated by the verb *inhibit*. Provided that all unauthorized actions against an asset fall into these three categories, we can say that an asset is *secure* when it has the following three properties:

- it cannot be accessed by an unauthorized entity (*Confidentiality*);
- it cannot be modified by an unauthorized entity (*Integrity*);
- it cannot be made inaccessible to an entity that interacts with it in an authorized manner (*Availability*).

These three concepts form the *CIA triad*, and constitute the three pillars on which cybersecurity builds up (Figure 1.1). Referring back to the original definition, we can mean security of a system asset as its condition of freedom from the dangers of theft, corruption, and inhibition.

Still going back to the original definitions, a *threat* can be meant as the intention to steal, corrupt, or inhibit an asset in order to harm the system, while a *danger* is the possibility for an asset to be stolen, corrupted or inhibited. What materializes the concept of danger in our domain is a *weakness*. A weakness is a particular class of defect that an asset may have with respect to its realization. Falling into practical case, a weakness could be, for example, a program's mishandling of an input area in memory, such that it allows bytes to be written beyond their intended terms [3]. When this generic class of defect is instantiated within an actual asset, such as the specific version of a software, then it is called *vulnerability*. Referring to the case of

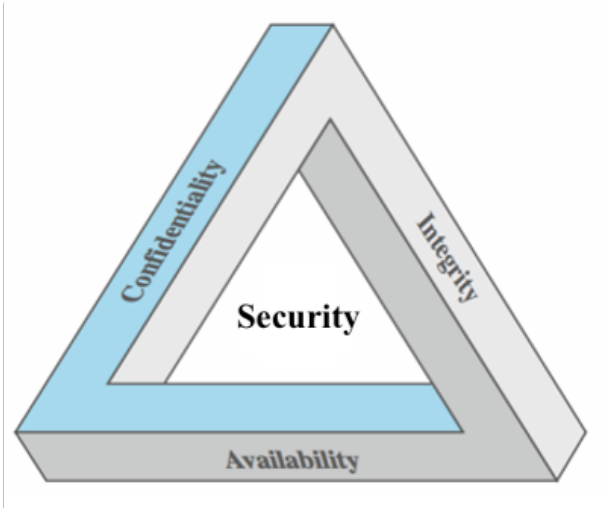


Figure 1.1 The CIA triad.

the previously mentioned weakness, for example, a vulnerability of that class has been found in the software QEMU version 3 in 2019 [4].

If the vulnerability materializes the concept of danger for a system, it means that it may lead to theft, corruption or inhibition of an asset. In other words, in the presence of a threat, the vulnerability is the gateway for unauthorized actions. Thus, we can say that the vulnerability is not such if it cannot be exploited by an external entity to perform an unauthorized action. An *exploit* is exactly the mean or method by which the vulnerability is exploited to one's advantage to perform malicious actions on an asset. An *attack* (or *cyber attack*) is finally the act by which a threat is actualized, i.e., the practical and deliberate act of taking advantage of a vulnerability through an exploit to harm a system asset. The Internet Security Glossary published by the Internet Engineering Task Force (IETF) in May 2000 defined *attack* as follows: "*Any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information itself.*". This definition is perfectly adherent to what has been achieved in the present discussion: in fact, all these malicious actions can fall within the three macrocategories identified above, as well as we can superimpose the *targets* of an attack according to this definition (*information system resources* and *information itself*) with the information and property assets we have described above. Figure 1.2 schematizes the central concepts with respect to the definition of attack.

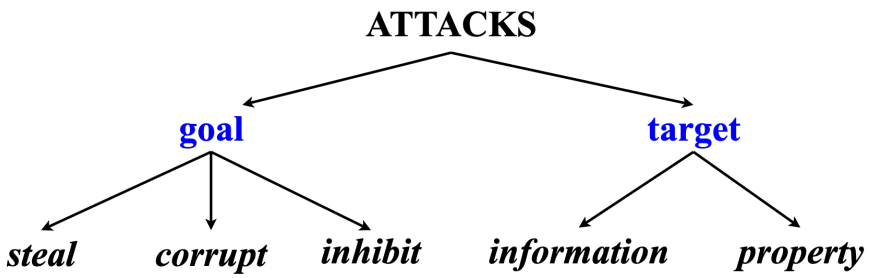


Figure 1.2 Characterization of attacks.

1.2 Vulnerabilities

As said, a vulnerability is an instance of a weakness within a system, such that it constitutes a danger for an asset. It represents the mean by which a threat can materialize into an attack. Thus, one can reason about the logical fact that without a vulnerability, an attack is not possible. Classifying an attack, which certainly includes finding a goal and a target, is a non-trivial task, as the ways an attacker can choose to exploit a vulnerability are potentially infinite, and even the same vulnerability can present multiple exploits, e.g., depending on the attacker's goal and target. Very virtuous examples for attack modeling include, for example, the ATT&CK database[1] maintained by MITRE Corporation. In any case, if it is true that an attack cannot exist without a vulnerability, then vulnerability classification can certainly be a starting point to offer a good key on attacks. Indeed, the dynamic nature of the attack is opposed by the static and determined nature of the vulnerability. For example, we can say that a vulnerability unequivocally resides within a *domain*: in fact, it can be identified in a hardware component, but also within software, or it can even transcend the technological domain and affect the organizational aspects of data processing, and even human aspects. Domain is therefore one of the dimensions through which to classify a vulnerability.

Another important distinction that can be made between vulnerabilities is that concerning their origin. Vulnerability is in fact the instance of a weakness, which - as we have seen - models a design or implementation defect with respect to an expected or assumed behavior. Given today's complex processes for the realization of a computational system, a full-bodied etiology can be developed to model the derivation of vulnerabilities within a system, and in the following we will actually

[1] https://attack.mitre.org/

try to give hints on how to motivate some vulnerability categories. However, an inescapable element, preceding all etiological considerations that can be made, relates with the *nature* of the vulnerability itself, whether it is the result of human error or misunderstanding during the creation of a system (*unintentional*), or the result of a conscious will to make the system being created weak, e.g., in order to later exploit that vulnerability or sell knowledge of its presence to others (*intentional*). This distinction is fundamental and is another of the dimensions by which to classify a vulnerability.

Figure 1.3 graphically shows what we have concluded so far, with further branches under some of the identified categories, which we will take care to explain below. As can be seen, vulnerabilities of an unintentional nature are further divided into two subcategories: *bugs* and *flaws*. Intentional vulnerabilities, on the other hand, are referred to as *backdoors*. Regarding the three domains listed so far (*technological*, *organizational*, and *human*), further subdivisions are presented for the technological domain: the vulnerability may indeed reside at the *software* level, at the *hardware* level, but also at the level of *communications* between different components or systems, and even at the level of *specifications*.

This offered taxonomy cannot be considered as a complete one for vulnerabilities, as many other dimensions can be added to classify them. It is of interest to this discussion giving the reader a glimpse of these two dimensions only, in order to emphasize the philosophical results achieved by the research work that produced this thesis. In general, the references that this thesis accepts for the collection and grouping of all known weaknesses and vulnerabilities are, respectively, the *Common Weakness Enumeration* (CWE) database[2], maintained by MITRE Corporation, and the *Common Vulnerabilities and Exposures* (CVE)[3], maintained by the same company.

Let us now inspect the items presented in Figure 1.3. For simplicity, we will use the word *product* along this stretch of discussion to refer to an artifact that constitutes a component of a computer system or the system itself.

A *bug* is an unintentional error made by an operator along the product lifecycle that leads, within its characteristics, to an unexpected and undesirable result with respect to what was specified for that product by the stakeholders involved in its

[2] https://cwe.mitre.org/
[3] https://cve.mitre.org/

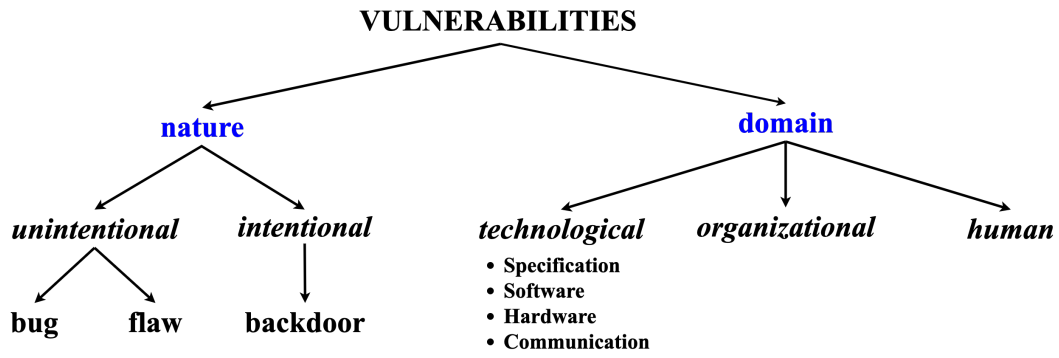


Figure 1.3 Characterization of vulnerabilities.

production. Take the example of a program subroutine, `get_value_from_array()`, which accepts as parameters: an address in memory, the amount of data stored at that address, and a given index. Suppose that, according to the specification, the function shall print the value of the array stored at that index, while it shall print an invalid value (e.g., -1) if the index is not contained within the bounds of the array. Suppose that the programmer's implementation in C language of `get_value_from_array()` is as shown in Listing 1.1. It can be seen that the code is affected by a serious vulnerability, which allows data to be read even outside the array bounds. In fact, the programmer takes care to check that the passed index is strictly less than the length of the data, but does not check whether this is a positive number. For pointer arithmetics, a negative index is perfectly allowed in the C language. This creates a vulnerability that is an instance of CWE-127, called *Buffer Under-read* [5].

```
void get_value_from_array(int *array, int len, int index)
{
  if(index < len) {
    value = array[index];
  }
  else {
    value = -1;
  }

  printf("Value is: %d\n", value);
}
```

Listing 1.1 An example of bug.

Bug causes can reside in several sources, including involved people, procedures, and tools. They can be determined by the inexperience of developers, designers, or test engineers in relation to design/implementation errors, insufficient/incorrect validation and verification (V&V), or insufficient/incorrect coverage of performed tests.

A *flaw* is instead a non-primary feature that is inserted by an operator into a product, and that does not constitute any inconsistency with what is specified for that product by the stakeholders involved in its production. It results from a misconception by the operator, that inserted it without taking into account its potential dangerousness. Let us take the example of a program that, according to specifications, shall perform an attempt execution to open a connection to a database. In case of failure, the program shall show the user a proof of what happened, and possibly provide indications for solving the problem. Suppose an operator's implementation of this functionality is as shown in Listing 1.2, which shows a few lines of Java code. As can be seen, the program catches the exception and prints two messages: one shows the message contained within the exception, the other aims to help the user to possibly solve the problem. The vulnerability consists of the latter print, as the program is exposing some information that in principle does not pertain to the user, and that could be maliciously exploited to conduct a subsequent attack toward the file stored at the indicated path. We are dealing with a vulnerability that is an instance of CWE-209, called *Generation of Error Message Containing Sensitive Information* [6].

```
try {
 openDbConnection();
}
catch (Exception e) {
 System.err.println("Caught exception: "+ e.getMessage());
 System.err.println("Check credentials in config file at "
    + MYSQL_CONFIG_LOCATION);
}
```

Listing 1.2 An example of flaw.

Unlike bugs, flaw causes rather lie in the lack of security culture by designers, developers, or test engineers, even when the level of experience of such operators is well advanced with respect to other metrics (e.g., performance optimization, ease

of use). Whereas a bug is caused by insufficient coverage of use cases during the various stages of production, a flaw is determined instead by poor coverage of *potentially malicious* use cases. In other words, a bug is simply originated by an inconsistency error between what is expected for that product and what that product actually features, while a flaw is precisely caused by a failure in correctly assessing security during production, even when it meets all the specifications that have been given. It is therefore important to point out this difference, and to introduce it within a coherent taxonomic system that adheres to reality.

Bugs and flaws are the two alternatives to classify unintentional vulnerabilities. On the other hand, when an operator consciously introduces into a product a feature, be it present or not present within its specification, in order to guarantee itself or others the ability to interact with assets contained in that product, in a dissimilar manner to what the stakeholders involved in the production intended for the end user, we are dealing with a *backdoor*, or alternatively a *trapdoor*. What distinguishes between a bug/flaw and a backdoor is just the intentionality of the operator who introduced it: the distinction does not pertain to the form of that vulnerability. A backdoor is *always* a vulnerability, even when the operator did not introduced it with malicious intent. Let us take the example of a function that, given a pair of credentials (username and password), shall return a boolean that certifies the validity of those credentials and, subsequently, allows a login to the system. Suppose an operator's implementation of this function is as shown in Listing 1.3, written in Java language. As can be seen, data are handled in accordance with some basic security rules: for example, the collection containing the list of accounts does not present plaintext passwords, but the comparison is done after a secure digest of the input. However, in the last check, the return of a positive value is upon two OR-ed conditions: either the password hash is equivalent to the one stored at that user's entry, or the plaintext password equals the string "letmein". The latter condition precisely represents a vulnerability which is instance of CWE-912, called *Hidden Functionality* [7].

It is meaningless to make assessments about the etiology of a backdoor, since the root cause is the sole intent of the operator who placed it within the product. It cannot be excluded that originally the operator may have inserted this backdoor, e.g., for more agile debugging during functional verification phases of the program, or even to ensure easier maintenance in anticipation of a future problem that, to be solved, requires privileged access to the system, outside the usage patterns designed for the

end user. However, this in no way changes either the extent of the vulnerability or that of the consequences of an attack exploiting it: the potential harm to the system is not affected by the malice with which the backdoor has been inserted.

```
public static boolean authenticate(String username, String
    password)
{
    if(!accounts.containsUser(username))
        return false;
    Credentials cred = accounts.getCredentials(username);
    String hash = Crypto.secureDigest(password);
    if(cred.getPassword().equals(hash) || password.equals(
       "letmein"))
        return true;
    else
        return false;
}
```

Listing 1.3 An example of backdoor.

Moving to discuss the *domain* in which vulnerabilities may reside, we find three categories (*technological*, *organizational*, and *human*), the first of which has four subcategories that, in broad sense, may encapsulate all levels of abstraction detectable on a computer system. Vulnerabilities present at the *specification* level typically arise from inconsistencies, incompleteness, or ambiguities present in the various levels of documentation that accompany the creation of a product. As for *software* vulnerabilities, some notable examples have already been introduced through Listings 1.1, 1.2, and 1.3. At the *communication* level, vulnerabilities can be found, e.g., with respect to a faulty implementation of a communication protocol, even when the protocol model is proven to be secure. Other examples may be the non-application of an encryption scheme, which leaves communications totally unencrypted, or alternatively the application of a weak encryption scheme, that can be broken without the need for intensive computation [8]. Protocols making use of numbers that are not perfectly random [9], or making use of insufficiently long keys, may still be mentioned. Finally, hardware vulnerabilities will be discussed extensively in the next Section.

Less related to the collective imagination are the vulnerabilities that we can classify as belonging to the organizational or human domains. Residing in another

domain than the technological one, these vulnerabilities pose a danger to assets that may not even be purely technological, but which we could more appropriately call as *meta-technological*.

Organizational vulnerabilities arise from the inadequacy of organizational aspects in terms of structures, infrastructures, or strategies adopted by organizations that make use of IT and OT systems. An attack that exploits these vulnerabilities can impact not only on technological assets, but also on entire defense infrastructures, on incident response procedures, on attack detection, remediation and recovery systems, or on the partial or total resilience of the organization itself. In this domain, problems arise from the lack or inadequacy of the security strategies adopted, with particular reference to the robustness of the teams in charge of managing cybersecurity risks and incidents, best practices published by public or private entities, and employed tools and technologies.

Finally, human vulnerabilities are those related to the presence of the human factor within data treatment processes, and are those that most elude any kind of taxonomy, clustering, or methodology for detection and remediation. Although their impact may be thought as lower than that of vulnerabilities in the previous two domains, about 4 out of 10 attacks are thought to be caused by the human factor [10]. Phishing and social engineering attacks [11] are the most classic examples of attacks exploiting these vulnerabilities, such as human guilt, panic, ignorance, lust, greed, and compassion to harm an asset guarded or supervised by the victim person.

1.3 The Role of Hardware in Cybersecurity

Figure 1.3 gives us an important view of the abstraction layers that belong to the technology domain. Indeed, it is no mystery to note that IT/OT systems are *layerized* systems, i.e., in which there are hierarchically organized technology subdomains, whereby one layer interacts with the others in two substantial ways: by using the services offered by the layer below, and in turn offering services to the layer above. Apart from the specification layer, which is prior (or somehow parallel) to the physical realization of systems, we can refine Figure 1.3 by stacking the technology layers according to their hierarchy (Figure 1.4). At the top of the stack, we find the communications layer, through which information is transmitted and received to be processed in a distributed manner. Below that, we find the software layer,

which is a *mare magnum* of instances that can be further refined by distinguishing, at least, a sublayer more pertinent to the applications running on that system, and another specific to the management software of the system itself, which is directly responsible for the use of the resources that system relies on. This is because it resides immediately above the base layer for all systems, i.e., the hardware, the physical substrate by which they actually exist in reality. The word *hardware* in itself encompasses a vast set of concepts, with some of the most disparate meanings associated with it depending on the context, the actors involved, the end users, the vendors, or the manufacturers. For the purposes of this discussion, by hardware we will mean the entire set of electronic components that are part of a discrete IT/OT system [12].

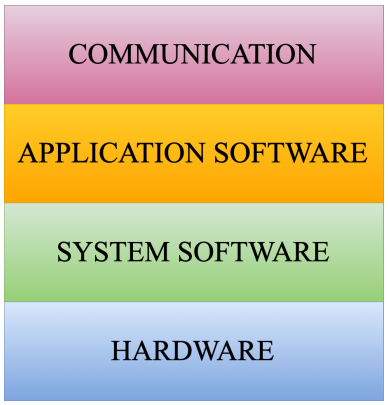


Figure 1.4 Layerized view of a computing system.

The layered view of systems takes on profound significance when coupled with the presence of vulnerabilities. The layered design of systems views each layer as independent of the others, interfaced only through a clear protocol of services required and offered, without going into the internals of each. Nonetheless, it is not difficult to imagine that when a vulnerability present within one layer is exploited by an attack to target one asset and damage the system, the others cannot be totally unaffected. Also because the stack representation is not accidental, and the layers have been placed vertically and not horizontally not just for convenience. When we say that the lower layer supports the upper above, we also mean an *encapsulation* of the upper one within the lower one: it is the lower layer that provides the upper one with the resources to exist, and such a provision is not like a *blank transfer*, but according to the way systems are architected, it is more like a *supervised concession*.

Thus, when an attack exploits a vulnerability present in one of the layers, there are basically two scenarios that can be encountered:

- The underlying layer is prompt and resilient, and can "take fuel away" from the infected part of the layer above, isolating the attack and making it harmless to the system;
- The underlying layer is weak and unresponsive, and the infection can spread to it as well, endangering the remaining parts of the system as well.

Thus, it could be said that, beyond the defenses that can be discreetly installed in each layer and its components, the lower layers can also act as an *immune system* for the upper ones, if designed or equipped in a certain way. Again looking at Figure 1.4, we find that hardware plays a crucial role in all of this. As constituting the backbone of the system, its last layer, it is logical to conclude that it represents, by construction, *the last line of defense*. This role can be read according to two different keys:

- If the hardware is weak and penetrable, then it can even make useless the defenses that have been placed at the overlying layers;
- If the hardware is strong and resilient, then it can constitute the final fortification on which the security of the overlying layers can be based;

Referring to this last point, the concept of *root of trust* has been developed in previous decades around hardware. The term is traditionally used to refer to one or more hardware components of a system that, by definition, are assumed to be secure, as the designer relies on them without being able to verify their trustworthiness [13]. For example, for a system software developer, a Trusted Platform Module (TPM) or a Subscriber Identity Module (SIM) are roots of trust on which to base security features. For the TPM designer, in turn, such security relies on the secure storage of keys in memory, or on the quality of random number generation. Thus, a secure core is identified that, from the user's point of view, behaves as a trusted component on which not to instill doubt. To describe the concept of root of trust with a metaphor, one might think of a *black veil* covering a given number of components. Actors legitimately interacting with it trust it and investigate no further; actors having malicious intentions are blind to its potential vulnerabilities, and find no entry point for their attacks.

Traditionally, the role of root of trust had been assumed assumed by the hardware in its entirety: the hardware used to be regarded as something unquestionable, to be taken as granted, and security instead had to be implemented in the layers above it. However, the realization of the hardware can violate this trust assumption. As with software, or communication infrastructures, hardware must be designed, built, tested, used, maintained, and dismissed, and throughout all this life cycle, the probability that any operator introduces a vulnerability is very high.

For example, specific hardware extensions for running cryptographic algorithms used to be regarded by system designers as black boxes for the attacker: only algorithm-level inputs and outputs could be observed, and only from those could cryptanalysis attacks possibly be mounted. However, when researchers have begun to talk about side-channel attacks [14], this assumption has lost its value. The attacker has begun to be able to exploit not only a mathematical vulnerability in the algorithm's specification, but also information about the time taken, the power absorbed, and the electromagnetic emissions to unlock the secrets contained in that cypto-component. Another example is about processor architecture. The model through which the hardware of a processor shows itself to the overlying software is the Instruction Set Architecture (ISA). The ISA can be seen as the main service that is offered by a processor to the program running on it. The ISA used to be considered the trust boundary for the software designer, until researchers discovered microarchitectural flaws within the most widely used processor families, such as Spectre [15], Meltdown [16], or Foreshadow [17].

Both of these cases demonstrate how the presence of vulnerabilities in hardware throws off the black-box model we had become accustomed to. *Hardware Security*, intended as a branch of the Computer Security science, breaks the black veil that traditionally shrouded hardware, revealing its previously ignored or underestimated nature, and grants the concept of root of trust a new, more concise meaning, tied to a set of specific components and not extended to the entire electronic component of systems.

Specifically, given the above considerations, hardware can serve the function of root of trust when (i) it has no vulnerabilities, (ii) by that, it is able to grant protection to other technological subdomains, and (iii) it comes from a trusted and authentic source, for which the actual trust relationship is created with the other components of the system, and they base their security on it. This triad encapsulates

the actual role of hardware in cybersecurity [12], and provides the key to identify three sub-branches within the specific subject (Figure 1.5):

- the study related to vulnerabilities and attacks in the hardware domain (*Hardware Security*, properly said);
- the study related to the use of hardware to mitigate vulnerabilities in other technology subdomains (*Hardware-Based Security*);
- the study related to the authenticity of hardware, to create trust in secure use (*Hardware Trust*)

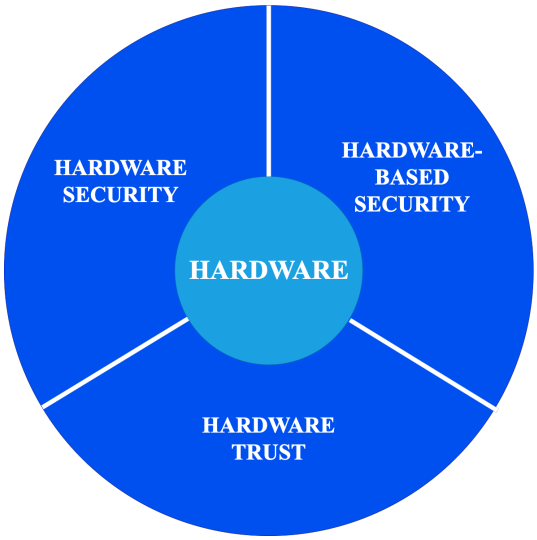


Figure 1.5 The triple role of hardware in cybersecurity.

Hardware Security thus becomes the name of the science related to all the actions needed to (i) identify hardware vulnerabilities, (ii) analyse their effects, (iii) prevent their exploitations by mitigating, reducing, and (ideally) nullify risks induced by their presence, (iv) develop and implement protections and remediation solutions, and (v) possibly avoid them by proper remediations during the design and production phases. Note that this definition is in no way constrained on *where* or *when* what described above can be done. The fact that vulnerabilities are located in the hardware and that hardware attacks try to exploit them, does not necessarily mean that defences must be implemented at the hardware level. This would be extremely limiting, especially if considering that most vulnerabilities are discovered once the hardware

is already operating in the field, without the possibility of being patched. Therefore, any technique aimed to counter hardware attacks falls under the scope of Hardware Security, even if mitigations are applied at upper layers.

Given the above considerations about the role of hardware within the science of Computer Security, it is legit to wonder how these find their place within training programs involving universities and teaching courses for the future class of cybersecurity experts. A considerable amount of the academic work done by the author along the past three years has been dedicated to this topic, and has led to specific results. In order not to interrupt the discussion at this point, the reader is pleased to refer to the Appendix of this thesis, where the above activities are presented in more detail.

1.4 Zero-Trust Model and Security-by-Design

On December 8, 1987, the U.S. President Ronald Reagan and the Soviet Union leader Mikhail Gorbachev signed an important agreement on nuclear disarmament, the *Intermediate-Range Nuclear Forces* (INF) Treaty, which was applied the following year. This episode is reported as an anecdote because on that occasion the U.S. President used an expression derived from a famous Russian proverb: "Доверяй, но проверяй" (tr. "Doveryay, no proveryay"), meaning "Trust, but verify". This is not the first time this anecdote has been reported in the scientific literature related to cybersecurity, and the reason is explained in the following.

In Section 1.1, we said that an attack is what actualizes a threat, meant as the intention to cause injury to a system asset. This puts attack and threat in a causal relationship with each other: an attack occurs as a consequence of a threat, and there is no attack unless there is a deliberate threat first. In fact, as can be seen from the definitions we have given, security is a subject that deals with problems arising from the will to create harm by an agent. At this point, a fork opens up, and considerations become more strategic than technical. In fact, one can think of modeling threats as intermittent conditions, eventual, and confined to a certain circle of well-identified agents. This presupposes a very thorough threat analysis, which must identify attack vectors in time and space. By the very nature of the threats it is going to model, such an analysis needs continuous updating. On the contrary, not as a matter of simplicity, but rather of improved efficiency and more comprehensive modeling, one could think

of the threat condition as *constant*, continuous, and not necessarily coming from some agents, but from *all possible agents*, including all operators who play some role throughout the life cycle of the product.

In other words, returning to the proverb quoted by Regan, trust is never built on the basis of a zero-threat assessment with respect to discrete times and external agents, but always on the basis of verifying the absence of a dangerous condition, which, as we have said, in security pertains rather to the product itself and the presence of vulnerabilities, than to what lies outside it. This is the so-called *zero-trust model*, first introduced in 2010 in a famous report edited by John Kindervag [18]. The philosophy behind this concept can be briefly summarized this way: *Assume that the threat you are trying to defend against is always real, and potentially coming from any agent*, or also: *Act in such a way as to consider an attack as always imminent*. Such is the threat model regarded as scientifically valid by this discussion. The concept has also been embraced by leading global cybersecurity institutions, such as the UK's National Cyber Security Centre (NCSC), which in 2019 recommended zero-trust model in its guidelines for device security [19]. The same happened in the United States the previous year, with the release of the *Special Publication 800-207: Zero-Trust Architecture* [20], authored by the National Institute of Standards and Technology (NIST) and participated in by the National Cybersecurity Center of Excellence (NCCoE).

If one assumes this model as true, i.e., there is no no circumstance under which an operating product can be considered secure *a priori*, then subsequent considerations arise, touching all aspects of the product life cycle, including the early stages of design. In fact, it is illogical that a trusted designer, considering that the artifact to be created will necessarily face a threat, does not implement the necessary precautions. This may result in two further reasonings:

- Dangerous conditions should be reduced as much as possible by adopting design patterns that disveal and eliminate vulnerabilities in the transitions between stages of production, including the use of validation and verification (V&V) techniques employing formal or semi-formal methods [21];
- Even when dangerous conditions are identified but unavoidable, the artifact should be architected in such a way that it can *resist* and continue to operate, possibly protecting assets of particular value behind a further internal border (this point will find more clarification in the next Section).

Such are the principles behind what is known as *Design for Security*, equivalently called *Security by Design* [22]. In a nutshell, this design methodology consists of recognizing security as one of the requirements during all steps of a product's lifecycle, from the very early stages of specification to complete dismission. In fact, many flaws identified within systems arise from not considering security at all as a goal to be achieved, or from considering it as an ancillary property, possibly to be added later, only after the goals set on other metrics have been achieved (e.g., performance, usability, portability, maintainability). The Design for Security methodology places security as a target metric to be achieved, equally with the others, and forces developers to "make security travel" at the same speed as the others, including through specific formalities along the design process. The logical consideration behind the methodology is that if the security of the artifact is not addressed correctly from the outset, this impacts its marketability and usability, just as it does with consumption or execution speed, for example. An artifact that becomes part of a system and does not properly contribute to the protection of its assets, effectively puts it out of the market, even when its technical features might appeal possible customers. With supreme respect, one might think of Design for Security today as the very *Kantian categorical imperative* for IT/OT sector operators. Paraphrasing the great German philosopher [23], the rule of Security by Design might be summed up this way:

Act in such a way that you treat security never merely as a means to an end, but always at the same time as an end.

The methodology has several variations and facets: for example, it results in the assumption that any interaction with the artifact is potentially malicious. In the context of software design, it may involve the adoption of standards for creating secure code, such as MISRA [24] or CERT [25]. Another very important approach consists in avoiding to base security on design secrecy (*Security through Obscurity*). Although this often acts as a deterrent to a subset of potential attackers, if one assumes a ubiquitous threat, this is logically not enough. In this regard, it is relevant to cite the so-called *Linus' Law*, formulated by Eric Raymond in his 1999 book "The Cathedral and the Bazaar" [26], which asserts that "*given enough eyeballs, all bugs are shallow.*". In other words, the larger a product's observer base, the more quickly problems are characterized and exploits become obvious. Imagining that we can calculate the limit of this expression for the number of people tending to infinity, the

event of vulnerability discovery becomes certain, and thus, if the threat is ubiquitous, the attack as well.

Another important concept derived from this methodology is that which results in the specification of maintaining the lowest possible level of privilege for each actor interacting with the artifact. This is the *Principle of Least Privilege*, already theorized in 1974 the by American computer scientist Jerry Saltzer, who wrote "*Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.*" [27]. Examples of this concept are found today not only in the software domain, but also in the hardware domain (e.g., think of *protection rings* in the Intel x86 architecture [28]).

To date, the Design for Security methodology is also adopted at the corporate level by major global industry players. One example is the publication of the Microsoft Security Development Lifecycle[4], already more than a decade ago. International standards and legislation have acknowledged the concept as well, providing concrete steps for creating and testing secure systems. Some examples are the ETSI TS 103 645 standard [29], which specifies standards for IoT device security, and the ISO/IEC 27000-series [30].

1.5 Vulnerability-Tolerant Architectures

In Section 1.1, when we introduced the concept of *threat* and diversified the assets subject to threats with regard to IT/OT systems, we distinguished between *safety* and proper *security*, or computer security. The study of the former addresses the protection of assets related to the health and integrity of people and evnironment, while the latter pertains to the protection of technological assets, such as information and properties contained in the computing systems themselves. Both safety and security are actually part of the broader concept of *dependability* [31], which has been known in the scientific literature for decades. We can define dependability as the ability of a system to be relied upon to perform exclusively and correctly the system task(s) under defined operational and environmental conditions over a defined period of time, or at a given instant of time. Classically, a systematic exposition of the concepts internal to dependability consists of the identification of three fundamental

[4] https://www.microsoft.com/en-us/securityengineering/sdl

elements: (i) the *attributes*, i.e., how to assess the dependability of a system, (ii) the *threats*, i.e., what can affect dependability, and (iii) the *means*, i.e., what can increase dependability.

Attributes represent the expected properties of the system, according to which the quality of services offered by that system is evaluated. These may include, for example, *reliability*, meant as the continuity of proper service, or *maintainability*, as the ability to offer easy remedy to an adverse condition, but also safety and security, with the latter possibly broken down into its basic components: confidentiality, integrity, availability.

Threats are that something by which the dependability of a system can drop. Although a precise meaning has been given so far to the term *threat*, inherently to security, this is broader in the context of dependability, and involves the presence of the so-called *threat chain* (Figure 1.6). It consists of three key elements:

- *Fault*: a defect present in the system;
- *Error*: a discrepancy between an intended behavior and the actual behavior of a system. Errors occur when some part of the system enters an unexpected state as a result of a fault activation;
- *Failure*: the evidence that the system behaved differently than expected as a result of an error, resulting in an undesirable condition with respect to the dependability attribute considered.

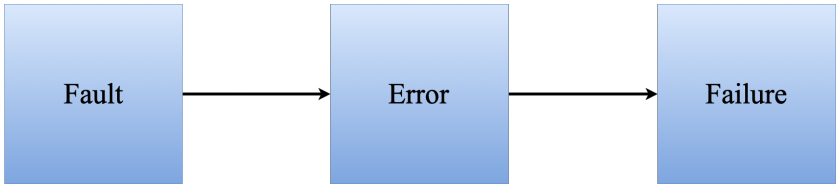


Figure 1.6 The dependability threat chain.

The time consequentiality of this chain, i.e., a fault causing an error leading to a failure, is not necessarily caused by the sole presence of a fault. Although a system may contain a fault, if its inputs and internal state do not cause its activation, the above chain is not generated.

If we wanted to decline these three elements within the context of security, we could identify the *vulnerability* with the fault, the *attack* with the error, and the *harm*

that is caused to the asset affected by the attack with the failure (Figure 1.7). In the specific case, the activation of the fault is only subject to the will of an external malicious agent, which is what we have indicated so far by the term *threat*.

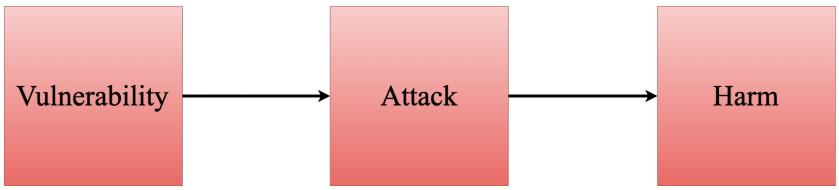


Figure 1.7 The dependability threat chain declined to security context.

The last fundamental element of dependability is means, which are nothing but the products of the related sciences, i.e., all those methods for analysis, identification, detection, prevention, remediation, patching of faults (vulnerabilities) for preventing, mitigating, defeating, making ineffective errors (attacks), so to avoid failures (harms). The means address the root of the threat chain, i.e., the fault, and are divided into *procurement means*, including *fault prevention* and *fault tolerance*, and *assessment means*, including *fault removal* and *fault forecasting*. The summary diagram of what has been presented on dependability is depicted in Figure 1.8.

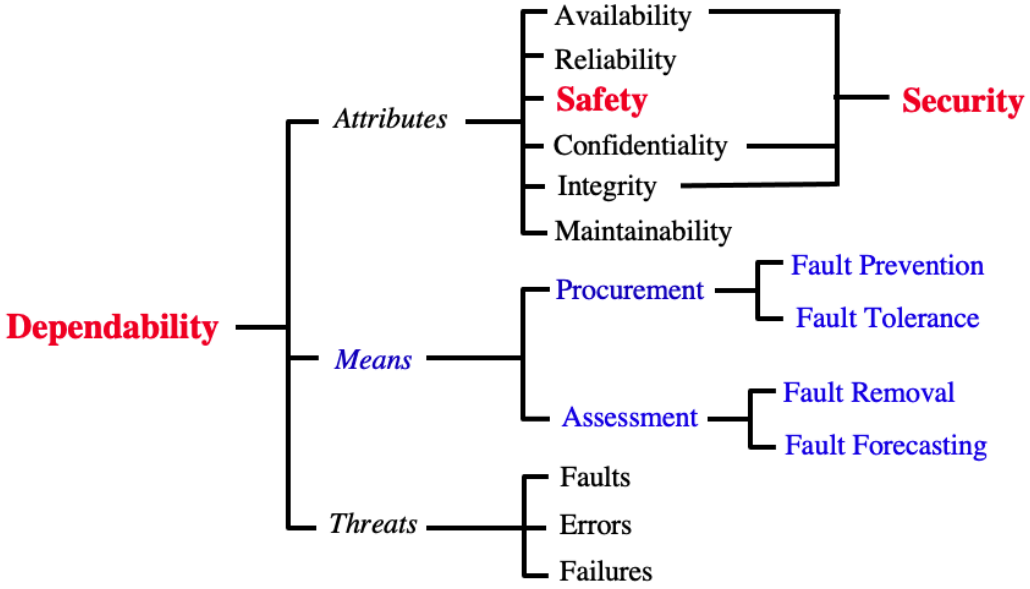


Figure 1.8 The dependability tree [31].

Importing the whole concept of dependability is not strictly necessary for the purposes of this discussion, but it is rather useful in declining into the narrow domain

of security one particular mean: *fault tolerance*. In fact, while it is not that difficult to imagine a security transposition of the concepts of prevention, forecasting, or removal, it is less evident how tolerance can be meant, so that its investigation is the main matter of the present dissertation. By the way, all four of these means completely fall within the scope of Security by Design, discussed in the previous Section.

The concept of fault tolerance has wide popularity in the safety community since long time [32]. It can be summarized as the property of a system to continue to operate even in the event that one of its faults is triggered and leads to an error. The concept is also known as *graceful degradation*: it is clear that the activation of a fault always results in some loss for the system, but the discussion all revolves around *how much* this loss impacts the activity of the system itself, or at least how much this impacts what is considered an asset by the specific safety policy. To give some practical examples, a fault tolerance solution for hardware may require that damaged parts be removed and replaced with new parts without disrupting system activity. This technique is also known as *hot swapping* [33]. In the communications domain, some data formats can be designed to degrade in a way that prevents failure. HTML[5], for example, is designed in such a way as to be forward-compatible, allowing Web browsers to ignore unsupported features of a document without making the page itself unviewable. Other examples may be networks that continue to transmit data despite failures in links and nodes. And even transcending the computer domain, we can also find it in civil engineering, where buildings and infrastructures can be designed not to be destroyed in earthquakes, floods, or other natural events of catastrophic nature.

Usually, a common principle in fault tolerance techniques, especially in the safe design of electronic components that must operate in high-risk areas such as air or space transportation, is the so-called *redundancy*, which is the presence of multiple identical instances of a certain subsystem to perform a single function [34]. While under normal conditions this redundancy is absolutely useless, it gains significance when a fault activates. Think for example of a transient fault, caused by an ion beam inducing a bit flip, or of a persistent fault, due to aging of a physical component. In such cases, the faulty subsystem among those present is immediately identified and isolated, since redundant architectures are usually accompanied by a decision logic

[5] https://html.spec.whatwg.org/

also called *voter*, that compares the results of the different redundant portions and continuously checks their consistency.

Provided that (i) fault tolerance is one of the means by which to increase dependability levels of a system, (ii) security is one of the attributes included in dependability, and (iii) in the specific context of security, the fault is represented by a vulnerability, then one can imagine the concept of ***vulnerability-tolerant architecture***: a system that, despite including a vulnerability, continues to operate while respecting the valued attribute, i.e., the security of those assets that are the object of the adopted policy. There is a vulnerability, there is an attack that exploits it, but nevertheless, no harm is done to the assets that are guarded by the policy. In the whole area of dependability, including security, the expected outcome is to avoid not the error (and therefore, the attack), but rather its consequences, i.e., the failure, the harm that is done to one of the assets.

For example, referring back to what said in Section 1.3 about hardware-based security and what was said in Section 1.4 about unavoidable dangerous conditions that must be resisted, an instance where vulnerability tolerance techniques can find application is that of *hardware support for software security at the architectural level*. The assumption is that a set of vulnerabilities present in the software cannot be avoided, but still, the system needs to ensure that certain assets are protected. As discussed, hardware can act as a security enabler for software: therefore, hardware-based security and vulnerability tolerance can join when there is a need to build *secure hardware running insecure software*.

The two basic concepts underlying the design of vulnerability tolerant architectures are *isolation* and *attestation* [35]. By isolation, we mean those mechanisms that are added to the architecture to enforce access control policies, such as hardware extensions to validate the interactions between various pieces of software and the associated data flow. A Protected Model Architecture (PMA) [36] provides a hardware guarantee that a piece of software executes undisturbed with respect to possible external influences, or conversely limits the effects of potentially tainted software on its environment. By attestation, on the other hand, we mean those mechanisms for proving to a third party that the system is in a particular state, usually to distinguish whether this is legitimate or illegitimate [37]. Attestation can be *local* and *remote* depending on where the request to provide state information comes from.

Among the most important practical translations of these two basic concepts in vulnerability-tolerant architectures are:

- *Sealed storage* [38], i.e., the process of wrapping code and/or data with certain configuration, process or status values. Sealed data can be unsealed only under a correct configuration (e.g., program counter value, nonce, secret key, etc.). Dynamic root of trust in combination with a late launch may guarantee that even if the processor starts from an unknown state, it can enter a fixed known piece of code and known state. This typically requires special instructions to enter and exit the protected partition;
- *Memory protection*, i.e., the protection of data when it travels between the processor unit and the on-chip or off-chip memory. It protects against bus snooping or side-channel attacks, or more active fault injection attacks [39];
- *Control-flow integrity* [40], i.e., a security mechanism to prevent malware attacks from redirecting the flow of execution of a program. In hardware, the pre-computed control flow of the program is compared at runtime with the actual one;
- *Information flow analysis*, i.e., a security mechanism to follow the flow of sensitive data while it travels through the different components, from memory to cache over multiple busses, into register files, and to processing units and back. This is important in the context of micro-architectural and physical side-channel attacks [41] [16] [15].

Among these, *Control-Flow Integrity* (CFI) is the topic on which a substantial (though not unique) part of the work of these three years has been focused, and which has provided the knowledge necessary to achieve the scientific results discussed in this thesis. In particular, the research focused on the study of **vulnerability-tolerant architectures for ensuring control-flow integrity in the domain of so-called resource-constrained devices**, more commonly known as *embedded systems*.

Chapter 2

Embedded Systems and Security

As stated at the end of the previous Chapter, this thesis is the result of the study of security within in a particular domain of computers, i.e., the embedded systems. The present Chapter is intended to provide the reader with the necessary background on the characteristics of these devices, and the security issues associated with them.

2.1 The Embedded Systems Domain

Since the ancient times, the history of human thought has always been characterized by many basic oppositions: immanent and transcendent, physical and logical, finite and infinite. Among these, there is a long-standing one about the nature of reality, and in particular about the distinction between what can be *thought*, and what can be *sensed*. Already since Anaxagoras, there has been a distinction between the concepts known in philosophy as *noumenon* and *phenomenon*. A centuries-long debate has pitted philosophers against each other as to the actual location of the noumenon, i.e., whether within our intellect or actually in physical reality. However, the interaction between humans and reality is mediated by the phenomenon, as a sensory representation of our surroundings. Until a few decades, there was no doubt that the phenomenon is the form taken by reality from its interaction with our 5 senses, namely sight, touch, hearing, and even smell and taste. In recent times, however, we are no longer the only beings who can sense reality, since a formidable companion that we ourselves have created travels alongside us today: the digital technology. In spite of our 5 senses, it has only one way of getting in touch with its

surroundings: by reducing all of them to a string of ones and zeroes chained together. Because of the today's central and inescapable role that digital technology plays in our lives, we can safely say that *data is the new phenomenon*, the real key to today's understanding of reality, without making the holders of Philosophy courses at the most prestigious universities jump in their chairs. As of today, in 2023, *everything is data*.

In this everything, of course, we are included as human beings, and we hardly manage to think of an aspect of our lives that has not gone through the numerical quantization dictated by the digital innovation. The amount of money at our disposal is data. Our medical records and our hospital history, the medications we have taken, and the vaccines we have had are data. Our work history, and therefore our tax and contribution history, is data. Our passions, our hobbies, the people we love and what we do with them and share with the rest of humanity is data. These four examples alone represent a major portion of our lives, and all four share a common phenomenology: they are data, stored in some digital archive around the world, and processed and exchanged daily by information processing systems, namely computers.

All four of these examples are already part of a *macroscopic* dimension of data, which pertains to which we personally interact with machines and delegate to them the automatic handling of much information about our lives. However, there is also a *microscopic* dimension of data, which instead pertains to situations in which we do not even perceive the presence of a computer. This dimension certainly has a more recent history than the previous one, and can be better understood if we analyze the typical day of a modern man. Every morning, he wakes up, gets ready, and gets into his car to drive to work. The navigator shows him the route to take in such a way that he avoids most of the city traffic. Once he arrives near his place of work, he glances at the roadside information to see where there is a parking lot with still free space. Once he is parked, he locks his car, and at the end of his day, if he ever forgets which parking lot his car is in, his phone reminds him, indicating its location. When he arrives home, the gate to his house opens automatically when his car approaches.

While all these steps may seem normal to a person of our time, people who lived even until the 1980s or 1990s would have squint their eyes. This is because along all these passages, a very large number of computers have interacted with each other, creating and exchanging data, which modern man is not aware of, but which are

contained in his car, in the buildings where he goes, and even in the walls of his house. Through this depiction, we intended to make the reader understand what is meant by *embedded systems*, i.e., small computers worth a few dollars, hidden inside most of the artifacts that surround us, and responsible for the largest slice of daily global data production, although in their microscopic dimension. In 2021, the number of globally connected embedded devices reached 11.3 billion [42], compared to 6.2 billion personal devices (the union of PCs, tablets and mobile phones) [43].

But what is an embedded system in itself? It can be defined as a computational system (equipped with CPU, memory, and peripherals) that lives integrated within a larger system, and is intended to control one or more of its parts or functionalities, or even simply to record and monitor its activity. What mainly distinguishes it from a traditional system, such as a PC, tablet or smartphone, is its characteristic of being *special-purpose*: the applications running on embedded systems are already determined at the time of its creation, as well as its location and operational environment.

Since its mission is narrower than a general-purpose computer, an embedded system only features what is necessary to run its task, and the hardware platform that hosts it often has constraints: for instance, a CPU that runs more slowly to save battery power; a little memory so it can be manufactured more cheaply; a tight subset of peripherals; etc. To fix ideas, processors in such systems have speeds between 50 and 150 MHz on average. CPUs usually belong to the RISC families, i.e., with a small, non-complex instruction set. They are pipelined processors, but often having a single execution unit, without supporting parallelism at hardware level. In more specific cases, they may have several computational extensions, such as those for Digital Signal Processing (DSP) or floating-point calculations, or more rarely, specific extensions such as for cryptographic acceleration. The reference architecture is generally the Harvard architecture, i.e., code memory is physically separated from data memory. In particular, code memory is mostly a ROM (Read-Only Memory), and uses NOR Flash or EPROM technologies [44]. In fact, if the application is already determined at the time of creation, there is hardly need to load from a drive the program to be executed, so that this software dimension is much more often referred to by the term *firmware*. Plus, the size of memories hardly exceeds a few MB. Similar sizes are those of the main data memory, implemented with Static RAM (SRAM) technology, as it does not need to be excessively large. Almost absent is

the concept of cache memory in this domain, and the reason are the speed and the proximity of these small RAMs to their processors.

With the advancement of technology scaling, to date, these systems are hardly implemented starting from discrete components on a PCB. Rather, we speak of *Systems on Chip* (SoC), where within the same silicon segment all components are integrated, even when from different intellectual property holders. An example of how an embedded SoC can be architected is depicted in Figure 2.1. We find a central bus that carries information between the processor, memories, and peripherals. Among the most common peripherals, we usually find an interface for directly handling general input and output lines (GPIO), but also standard interfaces for serial byte communication, such as UART, SPI, and even I2C. Widely used in the automotive sector is the CAN protocol. Then, among the peripherals in these systems are analog-to-digital converters (ADCs), which receive an analog input from, for example, a sensor. Also common are digital-to-analog converters (DACs), used instead in cases where a signal must be generated for actuation. In more modern systems, the presence of at least one peripheral for radio frequency communication is found, for example, to enable the system to communicate on the Personal Area Network (PAN) or Local Area Network (LAN), or more rarely on a Wide Area Network (WAN). Systems of this type, also called *microcontroller systems* or simply *microcontrollers*, have superseded the previous design paradigm, that wanted instead embedded systems directly implemented on an *ad-hoc* piece of silicon containing the required functionalities. This solution goes by the name of *Application-Specific Integrated Circuit* (ASIC), and is certainly more valuable from a performance point of view, but it incurs significantly higher costs concerning the individual design of silicon pieces. On the other hand, the microcontroller industry largely takes advantage of the so-called scale economy, and systems achieve prices that make them marketable. A middle ground between programmable SoCs and ASICs comes in the use of *reconfigurable hardware devices*. Examples include *Field-Programmable Gate Arrays* (FPGA) [44], i.e., chips that can be configured at the hardware level through the use of *Look-Up Tables* (LUTs) instead of logic gates to implement any functionality by proper programming. These systems can provide developers with a very high degree of efficiency, albeit not achieving the frequencies and power consumption of specific hardware, but greatly avoiding the generality of an already determined microcontroller system.

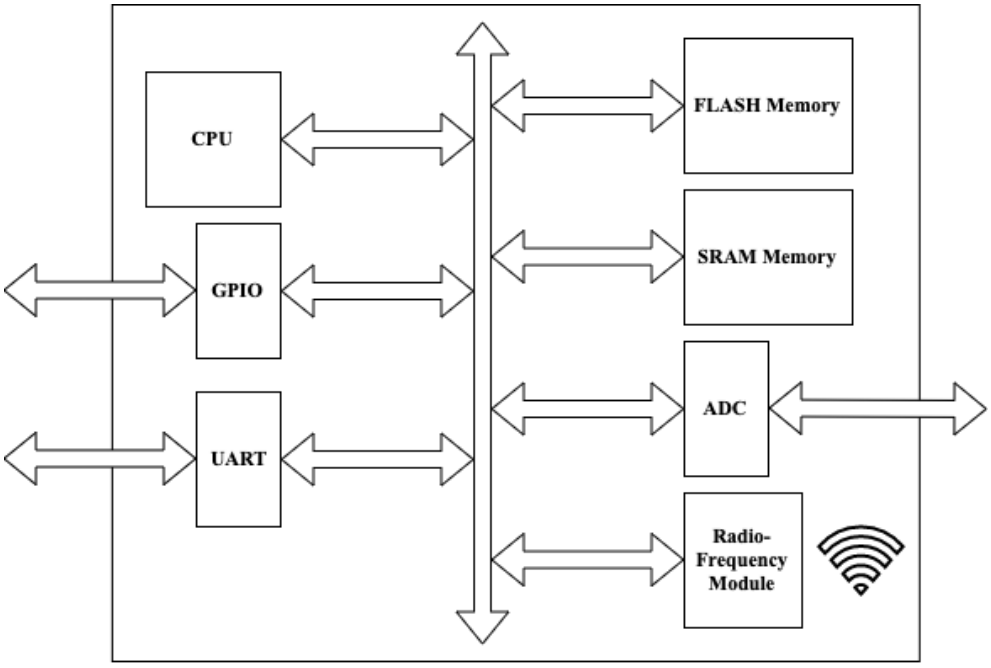


Figure 2.1 Example of embedded SoC internal architecture.

To program microcontroller systems, designers use languages that are strongly close to the machine, such as C, or more rarely C++, and in particular cases (e.g., startup procedures) even Assembly. IEEE Spectrum 2022 pointed to C and C++ as the second and third most widely used languages at all, only following Python [45]. This happens for obvious reasons of optimization in terms of power consumption and code size. Although the most widely used language today is Python, its consumption is estimated to be 40 to 60 times more of C [46], making it one of the least efficient languages in the world. In addition, the typical activity of embedded systems is very different and incompatible with the use of interpreted languages: the code is statically located on a Flash memory, and even if this could host an interpreter, loading a script from a secondary drive (assuming it is supported) would occupy many resources in main memory, leaving little space for data.

Since they are primarily dedicated to functions of monitoring, controlling, or actuating signals in response to stimuli from the surrounding environment, these systems are characterized by a *real-time* execution model: for this model, the execution of a task within a given time frame is a critical issue. This is completely different from the computational model of general-purpose systems, whereby waiting even a few seconds for a task to complete is not mission-critical. In addition, these

systems need to remain always up and running for a time that coincides with the life of the system they are integrated within. Therefore, they cannot go in for repairs or replacements, even for just a battery. Hence, the need to be very energy-saving, and to ensure a high degree of reliability (see the concept of graceful degradation in Section 1.5).

In some cases, when the executed application is unique, the system does not even need an intermediate layer between the hardware resources and the application, and the program runs as *bare-metal* software on the processor. In cases where some level of multitasking is required, the system supports certain libraries that allow contest switching between tasks once a certain amount of time has elapsed, or based on external signals. Such libraries are called *Real-Time Operating Systems* (RTOS), because of their characteristic of allowing real-time execution, but which have very few in common with a full-fledged operating system that we are used to know. Both in the bare-metal and in the RTOS case, all the code that sooner or later the system has to execute is already statically present in memory. So, even in case that an RTOS is there, the application can still be seen as unique.

To date, the presence of embedded systems can be found in any human sector: in industrial automation, automotive, home automation, medical, telecommunications, aerospace, etc. They can also be part of a more complex computing system: a mouse, a keyboard, but also a pair of wireless headsets, an external video camera, or a microphone, are all examples of tiny computers interacting with a larger one. In the industrial domain, embedded systems are used in the sensing and actuation of all manufacturing sectors, and are primarily responsible for what we call *Industry 4.0* [47]. In telecommunications, examples of embedded systems are telephone cells, routers, intermediate hops, and network bridge controllers. Within homes, they control appliances to introduce features, flexibility and efficiency. They control heating, ventilation, air conditioning. They are in charge of physical security of places, controlling physical access to buildings through radio-frequency identifications (RFID) or biometric-type sensors.

The transportation sector is pervaded by embedded systems, starting from the skies and coming down to the streets. We find them in airplanes for inertial guidance system, transponders, GPS, up to all the automated services offered on the user's seat. Ground transportation systems are increasingly using embedded systems to automate control, maximize efficiency, aid diagnostics, and even reduce pollution. Examples

include ABS, electronic stability control, automatic transmission, and front-wheel drive engagement, but also infotainment systems, radio, on-board navigation, down to the silliest functionalities, such as operating windshield wipers or windows.

In the medical field, embedded systems are part of the protheses installed in human bodies, and all the monitoring of patients being treated at hospitals, as well as medical imaging diagnostic tools, such as positron emission tomography (PET) or magnetic resonance imaging (MRI).

In such a global scenario, where finding tools that are not assisted by an electronic component is challenging, the statistic reported above on the number of interconnected embedded devices takes on even greater significance. Beyond the functionality of these systems as isolated units, aimed at collecting data and actuating signals in response to them, these devices contribute to a distributed automation that can make decisions even at the level of an entire building, factory, neighborhood, or city. It is the *Machine-to-Machine* (M2M) interaction model, that preempts the *Machine-to-Human* (M2H) model in an increasing number of scenarios, and constitutes what is today called the *Internet of Things* (IoT).

The topic of networked objects has been known to experts in the field for decades. The first notable example was the discussion made in 1982 about a Coca Cola vending machine located at Carnegie Mellon University, that was able to provide information about its inventory and whether the new drinks loaded were fresh or not [48], on the network then called ARPANET. Today, there are more than 11 billion "things" connected on the Internet, and they are expected to grow by about two billion each year, up to an expected 29.4 billion in 2030 [42].

2.2 Security Issues Overview

The data presented above deliver a picture of a society pervaded by computers, so that it is even difficult to locate a computer in space: they are literally in everything around us. The image of the attacker ideally taking his gun, ready to fire at the data center rather than at the PC full of sensitive information, is nowadays an outdated picture. A substantial portion of that targeted data comes from a multitude of smart objects out there, related to all those application domains we referred to earlier. The attacker thus finds himself in the unique condition of being *immersed* in his target.

The so-called *attack surface*, common expression by which we virtually denote the fertile ground that the attacker seeks to achieve success from his malicious actions, changes to an immense prairie involving home automation objects, wearable devices, city sensors. Without any overstatement, we could say that *if the data is everywhere, the attacker is everywhere too*.

The scenario is even more frightening when considering the average level of sensitivity that people have to security issues with regard to IoT devices. In a recent U.K. government-funded study, 72% of a sample of respondent consumers did not consider security to be a concern when purchasing an IoT device [49]. Unfortunately, the lack of full awareness for the issues runs up the supply chain, and still for a large part of the industry in this area, security is something evaluated *beyond* product marketization (in complete opposition to the Design-for-Security model presented in Section 1.4).

The attack known as Mirai [50], occurred in October 2016, certainly contributed to increase attention to smart object security. Mirai was a *botnet*, which is a coordinated network of IoT devices compromised and forced to implement a large-scale *Distributed Denial of Service* (DDoS) [51], targeting DNS providers like Dyn, and denying resolution services for major giants such as Amazon, Netflix, Twitter, Starbucks, and even the Wall Street Journal. Initially, this attack knocking out these giants for more than 8 hours was thought of as an act of *cyber warfare*, coming from an opposing foreign power. Then, when after one year it was revealed that Mirai had been designed and executed by a group of college-aged kids, and that most of the vulnerabilities exploited to conduct the attack were about breaking of trivial passwords such as `admin`, `password`, or `1234`, governments and companies at national and international level have been able to appreciate the true extent of the danger lurking in this *brave new world*.

To provide further consistency to the picture of the scenario we are describing, let us think of an example of a practical application, such as a *smart lock* that controls access to an apartment (Figure 2.2). Such an application can be functional, e.g., for an online house rental system, where an identification for unlocking is done through an application installed on the customer's cell phone. As can be seen from the Figure, the smart lock communicates via Bluetooth with the mobile device, which in turn communicates with a server on the cloud using a set of APIs over HTTPS. The smart lock is probably a low-power microcontroller system like those depicted in

Figure 2.1, and has no communication with the outside world other than with the user's mobile phone. The latter can be programmed to exploit the Bluetooth channel maliciously, appearing to be authentic due to some weakness on the cryptographic or protocol side, or exploiting a firmware vulnerability to break into the device. By communicating only with the user's mobile, it may not have stand-alone support for *key renewal*, for example; thus, even in the presence of some kind of encryption, there may be a hardcoded key within the device's memory that may be broken, or it may be the same as that of other devices of the same type, and thus compromised. On the other hand, then, the application handling the authentication of the mobile device and the subsequent door unlock has a high probability of being written in the C language (see previous Sections). This language gives the programmer complete freedom to manage system resources, to optimize their usage as much as possible, but without offering any built-in protection. As a result, the small program might contain a vulnerability, whereby the maximum size of characters sent via Bluetooth (e.g., those of the unlock code) is not checked, thus opening the way for memory corruptions that can lead to serious security consequences [52], as we will extensively discuss later in this dissertation.

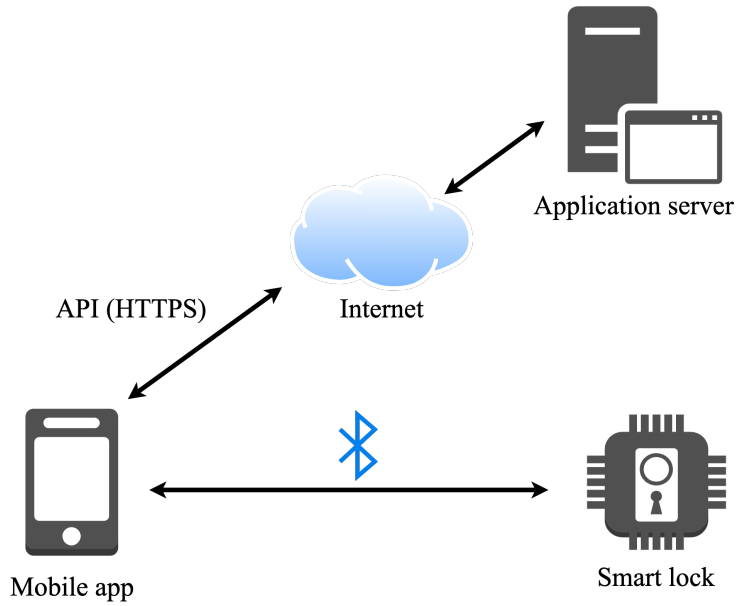


Figure 2.2 Schematic diagram of a smart lock system application.

Alone, these two examples of weaknesses depict a large portion of the problems that IoT devices can suffer from, which were well summarized in 2018 by the *Open*

Web Application Security Project (OWASP), which published a Top 10 list of the most common weaknesses for the Internet of Things [53]. The list includes:

1. *Weak, Guessable, or Hardcoded Passwords*: exactly as in the previous Mirai example, these devices might use factory passwords for simplicity, written in a hard drive such as Flash memory, without supporting their update, and easily the same password between devices of the same class; or, similarly, operators might use simple passwords for easier maintenance;

2. *Insecure Network Services*: when these devices fully support the TCP/IP stack, or make use of common versions of consumer operating systems, albeit with reduced features, it is possible for some unnecessary network services to be left accessible. If the services offered contain a known vulnerability, this provides an entry point for the attacker;

3. *Insecure Ecosystem Interfaces*: the reduced capacity of these devices leads operators to skip a whole series of checks on who is interacting with the device, why, and to what extent. Common problems include lack of authorization, lack of effective encryption on communications, and lack of filtering on embedded system inputs and outputs;

4. *Lack of Secure Update Mechanism*: this macro-class of weaknesses covers all problems related to the difficulty of updating firmware that runs on the device, including the lack of firmware validation, the lack of a secure channel to perform the update, and the lack of support tools such as rollback and security notifications related to updates [54];

5. *Use of Insecure Outdated Components*: in architecting firmware for a system, operators might use substantial portions of code (library or operational) written by unauthenticated third parties, possibly containing backdoors or other vulnerabilities; or, a version of a portion of code is installed on board the device being behind the last update, which might have reduced its attack surface. The class of weaknesses also affects hardware components, since the featured processors may contain security flaws (think of the examples of Meltdown [16] and Spectre [15]);

6. *Insufficient Privacy Protection*: a class of weaknesses that is not hard to imagine, given the pervasiveness of these devices, which may suffer from

problems related to the mismanagement of personal data that are created and exchanged; problems may involve insecure storage or the transfer of data to other entities without any authorization mechanism [55];

7. *Insecure Data Transfer and Storage*: as mentioned in previous points, IoT devices often suffer from lack of proper encryption or clear access control policies on data;

8. *Lack of Device Management*: the small size of these systems often leads operators to think that all the asset management, upgrade, secure decommissioning, or monitoring practices that are usually applied for mainframe systems are instead not necessary;

9. *Insecure Default Settings*: one of the most dangerous classes, involving the inclusion of default (and often unchangeable) features in the device that make the system poorly protected; in addition to easily guessable passwords, consider the example of a hardware interface for maintenance, such as the JTAG port, being left available without authentication during device activity [56];

10. *Lack of Physical Hardening*: without due physical protection of the system's hardware components, it can experience dangers related to its electronic nature, such as side-channel [14] or fault injection [57] attacks.

Making valid threat modelling for embedded devices is far from trivial. The main reason is the jungle that has arisen in putting together proprietary software and unknown or lesser-known protocols, with hardware components ranging from pure ASICs to well-known easy-developing platforms, such as Raspberry Pi[1] or Arduino[2]. Depending on the point of view, the dangers can take on even conflicting facets of each other. For example, many manufacturers create devices from these low-cost, general-purpose platforms, introducing a large chunk of unused software into their systems that, moreover, contains vulnerabilities known by many attackers. Even worse, for performance reasons, developers might bypass security mechanisms that are naturally supported by known platforms, such as Android or light versions of Linux. On the other hand, when there is a need to adapt a tiny operating system with real-time requirement, to minimize processing time, some security standards may

[1] https://www.raspberrypi.com/
[2] https://www.arduino.cc/

be removed or not even introduced. Because of their small capabilities, they may lack basic security controls, such as a whitelist of allowed code segments, or access control policies.

A simple but effective way to approach threat modeling in this domain is to use a known framework, such as the *STRIDE Threat Classification Model* [58], that focuses more on the weaknesses present at the technology level, rather than the assets toward which an attack may be launched, or the attackers who may want to harm the system. The STRIDE acronym represents the following 6 attack modes:

1. *Spoofing*: impersonating an authorized user for the system;
2. *Tampering*: violating the integrity of the system;
3. *Repudiation*: masking the history of malicious actions on the system;
4. *Information Disclosure*: violating the confidentiality of the system;
5. *Denial of Service*: preventing others from legitimately interacting with the system;
6. *Elevation of Privilege*: gaining unpossessed authority over the system.

If we closely look at these items, we can easily superimpose them on the goals that an attack may have, classified in Section 1.1: *tampering* may be superimposed on our *corrupting*, *information disclosure* may be superimposed on our *stealing*, while *denial of service* corresponds to our *inhibiting*. As for the other three, these are attack modes that always have a further goal: the attacker is interested in impersonating someone else, disguising his or her actions, or gaining higher privileges not as an end in itself, but always to perform a harm action that falls in one of the three other classes. However, their identification is valuable in effectively modeling possible attack vectors.

In the remainder of this thesis, among all these attack patterns, the one that will be investigated will be that of tampering, since the specific solution framework studied and presented here explicitly addresses the integrity of software executed by embedded systems, and in particular the integrity of the execution flow, i.e., to prevent the creation of execution paths different from those originally intended, and potentially malicious.

Chapter 3

Control-Flow Attacks: State of Knowledge

The present Chapter is intended to provide the reader with all the necessary background on the specific security issue discussed, namely the corruption of the execution flow of a program. After framing and defining what a binary attack is, the technicalities of Arbitrary Code Execution (ACE) attacks are presented, specifically in its two forms of *code injection* and *code reuse*. After reviewing the main historical defenses, an extensive discussion of Control-Flow Integrity (CFI) techniques is offered, with the proposal of an all-encompassing clustering from the main proposals offered by the literature and the market.

3.1 Binary Attacks and Memory Corruption

Before continuing our discussion, it is necessary to provide the reader with the coordinates within the security space that was defined in the first Chapter. In fact, we are about to deal with a type of security problem that clearly resides in the technology domain, and which develops from vulnerabilities that lie within the software subdomain. The subject dealt with here is attacks that have the first aim to corrupt an asset of a system, i.e., *the set of its intended code paths*. Philosophically, this concept is highly difficult to define. For the reader's primal understanding, think of the set of coded behaviors of one or more programs, reflecting the intention of a stakeholder, and reflected in at least one of the intended use cases. While

conceptually it may be difficult to fully delineate the concept in words, let us identify the set of expected executions of a system as assets under threat here. Later in the discussion, we will go on to define what is formally meant by *legal execution*, and what are the terms for making it *illegal*.

This first picture potentially opens the way for a flood of vulnerabilities and attacks, all of which can be described as the possibility of a piece of code to differ from stakeholder's *desiderata*, and all of which can be summarized under the macro-concept of *malware*. Relatively to our semantics, a malware is any set of machine instructions that, when executed, compromises the security of one of the assets of the system on which it runs. From this perspective, the terms *malware* and *malicious code* are interchangeable. Malware exists of all natures, forms, or purposes: we can distinguish between *viruses*, *trojans*, *worms*, *spyware*, *botnet malware*, *ransomware*, and many other types of malicious code [59]. Making a complete classification of malware here is beyond the scope of this discussion. But to properly give the coordinates of our problem, a fundamental distinction must be necessarily made. In the literature, there is a distinction between *dependent* and *independent* (or *standalone*) malware [35]. Standalone malwares are complete programs that can run on their own, and are typically first intruded into the system through one of its non-volatile drives and then put into execution. For example, worms or malware used to create a botnet belong to this category. Dependent malware are those that instead require a host program to be executed, i.e., they must *infect* a program that is running on a system, forcing that program to execute unintended instructions. Think, for example, of macro viruses that infect documents or malicious browser extensions.

In general, every dependent malware is in fact a specific violation of the integrity of the executable sections of a program. The *infection*, i.e., the actual moment when a vulnerability is exploited to introduce malicious code into the program, can occur when the program is running in main memory, but also when it is *cold* (not running and just stored) in secondary memory. A further distinction can be made on the abstraction level at which this infection occurs: e.g., malicious browser extensions occur at the application level, as the vulnerability lies in the possibility, provided by browsers, to link external modules, namely the extensions, bypassing the possibly present checks. On the other hand, when the infection is at a lower level of abstraction, and in particular, it is at the level of the binary instructions that make up the code of a program, we are speaking of a *binary attack*. We call binary attack

any attack that violates the integrity of the binary code or even data of a program, regardless of whether the program is running or at rest.

A very classic example of a binary attack is *NOP-ing* certain machine instructions. In the code of an application, there might be a call to a procedure, e.g., for verifying the existence of a valid license (Listing 3.1). Such a call is expressed by a sequence of bytes encoding the CALL machine instruction targeting that procedure (Listing 3.2). A vulnerable program, i.e., lacking redundancy codes to detect byte corruption in the code, could allow a cold attack (i.e., with the program still not running) for which bytes corresponding to the CALL machine instruction are replaced by a congruent number of NOP (no-operation) machine instructions, such that they do not misalign the static references of the software (Listing 3.3). When the program runs, instead of invoking the license verification procedure, it executes a series of idle instructions, allowing the software to run unlicensed.

```
if(verify_license()) {
    printf("Welcome, user!");
    run_full_mode();
}
else {
    printf("No license!");
    run_demo_mode();
}
```

Listing 3.1 Dummy example of a C program that checks for valid license.

```
66b:   89 7d fc                mov    %edi,-0x4(%ebp)
66e:   48 89 75 f0             mov    %esi,-0x10(%ebp)
672:   b8 00 00 00 00          mov    0x0,%eax
677:   e8 ce ff ff ff          call   64a <verify_license>
67c:   85 c0                   test   %eax,%eax
67e:   74 1d                   je     69d
680:   48 8d 3d c1 00 00 00    lea    0xc1(%eip),%edi
687:   b8 00 00 00 00          mov    0x0,%eax
68c:   e8 8f fe ff ff          call   520 <printf@plt>
691:   b8 00 00 00 00          mov    0x0,%eax
696:   e8 c1 ff ff ff          call   65c <run_full_mode>
69b:   eb 1b                   jmp    6b8
69d:   48 8d 3d b3 00 00 00    lea    0xb3(%eip),%edi
```

```
6a4:   b8 00 00 00 00          mov    0x0,%eax
6a9:   e8 72 fe ff ff          call   520 <printf@plt>
6ae:   b8 00 00 00 00          mov    0x0,%eax
6b3:   e8 9d ff ff ff          call   655 <run_demo_mode>
```

Listing 3.2 x86 disassembly of original binary from code in Listing 3.1.

```
66b:   89 7d fc                mov    %edi,-0x4(%ebp)
66e:   48 89 75 f0             mov    %esi,-0x10(%ebp)
672:   b8 00 00 00 00          mov    0x0,%eax
677:   90                      nop
678:   90                      nop
679:   90                      nop
67a:   90                      nop
67b:   90                      nop
67c:   90                      nop
67d:   90                      nop
67e:   90                      nop
67f:   90                      nop
680:   48 8d 3d c1 00 00 00    lea    0xc1(%eip),%edi
687:   b8 00 00 00 00          mov    0x0,%eax
68c:   e8 8f fe ff ff          call   520 <printf@plt>
691:   b8 00 00 00 00          mov    0x0,%eax
696:   e8 c1 ff ff ff          call   65c <run_full_mode>
69b:   eb 1b                   jmp    6b8
69d:   48 8d 3d b3 00 00 00    lea    0xb3(%eip),%edi
6a4:   b8 00 00 00 00          mov    0x0,%eax
6a9:   e8 72 fe ff ff          call   520 <printf@plt>
6ae:   b8 00 00 00 00          mov    0x0,%eax
6b3:   e8 9d ff ff ff          call   655 <run_demo_mode>
```

Listing 3.3 x86 disassembly of NOP-ed binary from code in Listing 3.1, with neutralization of the license checks.

But binary attacks may also target programs running in main memory, and may not involve direct corruption of code in memory. In general, as long as a program is running on the main memory of a system, there can be basically three ways to change its executed code, through some vulnerability:

- Code in the executable sections is directly overwritten with arbitrary code (*code corruption*);
- Writable sections of the program, even data sections, are exploited to introduce arbitrary code into memory for execution (*code injection*);
- Program's control data are modified at runtime so that its execution flow passes through instructions present in memory, but not consecutive in relation to the intended flows of the program (*code reuse*);

All these three types of attacks are usually collected under the definition of *Arbitrary Code Execution* (ACE). Code injection and code reuse attacks can be also called *code redirection attacks*, as in both cases, program control flow is forced to be redirected to other portions of code.

None of these attacks is possible without a vulnerability allowing to corrupt one or more data in memory: by corrupting, we mean exactly modifying the contents of memory outside of implicit or explicit permission. Now, we can try to imagine two broad types of weaknesses allowing memory corruption:

- from a physical point of view, by stressing the charge contained in the transistors forming the memory location, so that the bits contained flip, either by external physical equipment [60] or by software techniques (think, for example, of attacks exploiting the *Row Hammer* effect [61]);
- from a logical point of view, through a software vulnerability present in that program (or in another) allowing control over that location to be gained unduly.

This second type of weakness certainly includes *buffer overflow*. Buffer overflow is a problem that arises from the use of pointer arithmetic, whereby a variable expressing an address in memory is incremented or decremented without appropriate boundary checking on the data structure being accessed during a write. This may result in an out-of-bounds write, which corrupts adjacent data in memory.

For better understanding, let us think of an example of a C program that declares, inside or even outside one of a functions, two variables as follows:

```
char my_word[8] = "";
short my_year = 1994;
```

Assuming a big endian architecture, the memory configuration would look like this:

variable name	my_word								my_year	
content	*null string*								1994	
hexadecimal	00	00	00	00	00	00	00	00	07	ca

Suppose the program uses the C library function `strcpy()` to fill the contents of `my_word` as follows:

```
strcpy(my_word, "overflowed");
```

The string `"overflowed"` is 10 characters long, plus the string terminator (`'\0'`). The `strcpy()` function sees `my_word` as a memory address from which bytes start, without knowing the size of the area reserved for characters, nor does it accept a parameter to know it. The result is that the copy of the characters exits outside the space reserved for `my_word`, also overwriting the value of adjacent `my_year` to 25956 (`0x6564`):

variable name	my_word								my_year	
content	o	v	e	r	f	l	o	w	e	d
hexadecimal	6f	76	65	72	66	6c	6f	77	65	64

Buffer overflows primarily occur in functional languages with a high level of machine awareness, such as C and C++. As earlier said, despite the advent of more modern languages, these languages are still widely used because of their efficiency and ability to control machine resources at a low level. However, the direct handling of data structures in memory paves the way for a wide range of weaknesses. In fact, these languages lack memory safety capabilities: they have *weak typization*, unlike other newer languages. In C and C++, pointers to memory are normal variables, and as such can be freely manipulated by the program that uses them: they can be initialized to a random value, incremented, decremented, reassigned, and so on. It is not hard to imagine that when pointers are played around with, inconsistencies are easily introduced.

For MITRE Corporation, maintainer of the CWE database, buffer overflow tops the 2022 ranking of the 25 most dangerous weaknesses [62] with a score of 64.20,

i.e., 20 points ahead of second. This further supports the theory that while much more memory-safe languages exist, problems arising from this type of weakness are far from being a memory.

3.2 Code Injection and Code Reuse

As said, there are basically three ways in which an attacker can force a running victim program to execute arbitrary code. Either the attacker can find some exploit to directly change the bytes being executed, or he must redirect the control flow to a sequence of malicious instructions. These may have been arranged by the attacker himself in some memory location, or they may also already be naturally in memory. In the first of these two cases, the attacker could use one or more memory corruption vulnerabilities to accomplish both of the basic steps of the attack, namely the injection of code bytes and the control-flow redirection. The possibility is offered by the presence in memory of variables indicating a code location, i.e., *code pointers*. Indeed, it is possible that the program, for different reasons, needs to save the address of an instruction location.

Think, for example, of the procedure calling mechanism. It is well known that, when calling a subroutine, computer architectures save somewhere the address of the instruction to which they are to return. Although there are differences among the many families of architectures, all of them recognize the concept of *call stack*, as of a LIFO (Last-In-First-Out) data structure, where function frames are stacked. Such a data structure is naturally adapt to accommodate functions, since a stack exactly traces the cascading hierarchy of function calls: callee stack on top of callers (*push*), and tapping (*pop*) is done in the reverse order, i.e., the last caller is returned first. Because of an old convention (that had and has its good reasons), the stack usually grows by decreasing memory addresses: the callee frame is at lower memory addresses than the caller frame. The frame of a function is the set of local data that are necessary for the identification, execution, and maintenance of that function: the parameters to be handled by the function, the local variables that are declared by that function, and the references to the caller, including the extremes of its frame and the address to which it returns (Figure 3.1). The moment a new function's frame is created coincides with the execution of the CALL instruction, which differs between architectures but is present in each. The CALL is responsible for both (i)

transferring the flow to the subroutine code and (ii) saving the return value. The latter coincides with the address following the address of the calling instruction, and in most architectures it is already contained in the instruction pointer (IP) register, or program counter (PC), at the time of the call. When the execution of the subroutine is complete, a special machine instruction (RETURN), which is present in every architecture in various forms, destroys the frame by taking the saved return address and copying it to the program counter, so as to cause the execution flow to return to the caller.

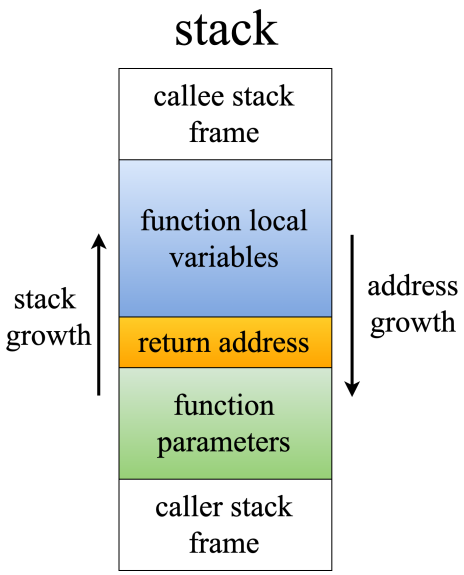


Figure 3.1 The call stack.

Suppose that, among the local variables of a function, there is a string like the one seen in the previous example, `my_word`. The program could make use of a function to copy values into it, like the previous `strcpy()`; but it could also take them from user input, using, e.g., the `gets()` function. Exactly like `strcpy()`, `gets()` does not control the maximum size of the target memory area (if referring to an external input, more properly called a *buffer*, in fact). The consequence is that the user can continue to enter characters: the input stream will only end when the user enters a string terminator (in the `gets()`, sent via ENTER). Here then, an attacker can exploit such a circumstance not only to corrupt the local data of the function, but more importantly, *its return address*: this will ensure that the function, when executing the RETURN, will return to an arbitrary location (Figure 3.2).

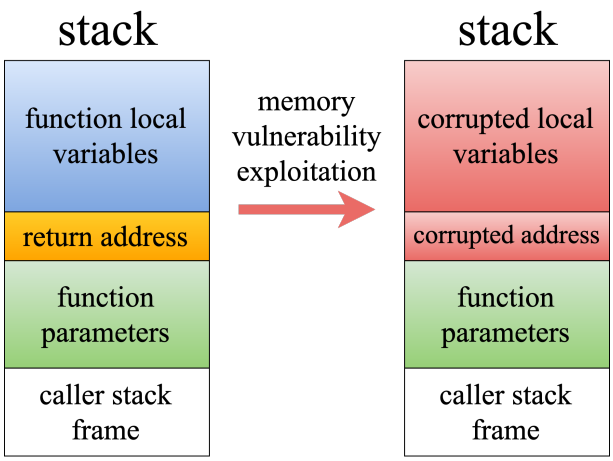


Figure 3.2 The call stack after a memory corruption targeting the return address.

To fix the ideas, suppose an application written in C uses the following procedure, `find_file()`, which searches for a file from a filename entered by the user:

```
int find_file(...) {
    char filename[12];
    /* function body */
}
```

Suppose the program runs on a Linux platform, with an Intel x86-32 architecture processor [28]. The function stack layout is similar to that shown in Figure 3.3.

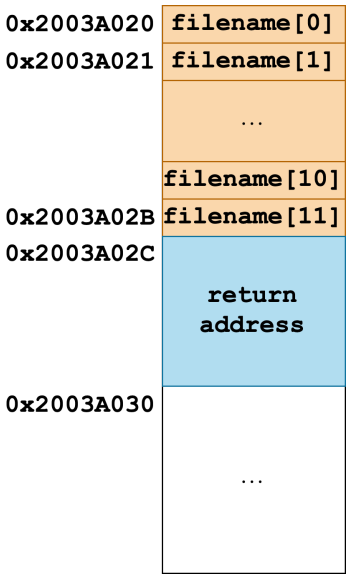


Figure 3.3 Frame of the `find_file()` function.

The memory addresses have been chosen randomly, and only to indicate their progression. As we can see, the variable `filename` is stacked on top of the return address by architecture construction. Suppose the attacker is interested in controlling the machine by interfacing with its *system shell*, which represents the service offered by the Linux operating system to send OS commands[1]. The attacker might be thinking of injecting into memory the bytes that correspond to the execution of the system call `execve("/bin/sh")`, which invokes the system loader to run a shell. Since all 32-bit Linux machines share the same *Application Binary Interface* (ABI), the code responsible for executing `execve("/bin/sh")` is common to all platforms, and can be easily found online. The machine code responsible for running a shell on the system is usually called `shellcode`. What we see in Listing 3.4 is a 28-byte version.

```
31 c0             xor    %eax,%eax
50                push   %eax
68 2f 2f 73 68    push   0x68732f2f
68 2f 62 69 6e    push   0x6e69622f
89 e3             mov    %esp,%ebx
89 c1             mov    %eax,%ecx
89 c2             mov    %eax,%edx
b0 0b             mov    0xb,%al
cd 80             int    0x80
31 c0             xor    %eax,%eax
40                inc    %eax
cd 80             int    0x80
```

Listing 3.4 Linux x86 machine code for `execve("/bin/sh")`.

At this point, the attacker must cause these bytes to be executed. To do this, it exploits `gets()` to perform two simultaneous actions:

1. injecting into data memory the bytes corresponding to the shellcode, at some location;
2. corrupting the return address of `find_file()` so that it points to the beginning of the injected shellcode bytes.

The attacker then provides the following input:

[1] https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html

```
"takeovercntl\x30\x0a\x03\x20\xc0\x31\x50\x68\x73\x2f\x2f\
   x68\x63\x69\x62\x2f\x68\xe3\x89\xc1\x89\xc2\x89\x0b\xb0
   \x80\xcd\xc0\x31\x40\x80\xcd"
```

The consequence is a stack configuration as in Figure 3.4. Upon return, the procedure will jump to execute the bytes of the shellcode, and the attacker will have the ability to interact with a system shell.

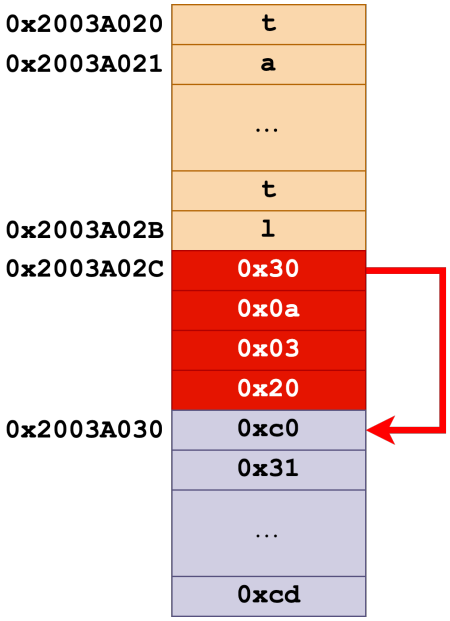


Figure 3.4 Frame of the `find_file()` function after corruption. The return address now points to the corrupted area containing the shellcode.

The attack we have described is a classic example of *shellcode injection*, and has been one of the most common of the attacks exploiting buffer overflows in the past. The danger was mostly neutralized when, about 20 years ago, computer architectures and operating systems began to implement *Write XOR Execute* [63] and *Data Execution Prevention* [64] policies. For such policies, memory pages containing data, and more generally writable by the program at runtime, cannot be executed. In many architectures, the policy is adopted through the implementation of an additional bit that flags non-executable memory pages [65]. If the same policy is viewed in reverse, i.e., executable pages cannot be written at runtime, then the system can be protected not only from code injection, but also from code corruption with the same technology.

However, this type of protection can only prevent the code from being corrupted or injected. Indeed, nothing prevents the attacker from corrupting the return address in such a way that it points to an instruction location that is present in memory, valid, and absolutely uncorrupted. A very large portion of applications written in C make use of the standard library (*libc*), which contains functions that, depending on the parameters, can act as incredible picks against the system. This is the reasoning behind *Return-into-Libc* attacks, an exploit known by experts since the early years of the new millennium [66]. Again to take up the previous example, we might assume that the victim program is legitimately using the `system()` function somewhere, which is functionally identical to `execve()`, and it constitutes its multiplatform version within *libc*. Without injecting any code, the attacker can mount the same attack described above, simply by knowing the address of the `system()` function within memory, and the address of the constant string `"/bin/sh"`. At that point, he can inject them into the stack and let the system do the rest upon return (Figure 3.5).

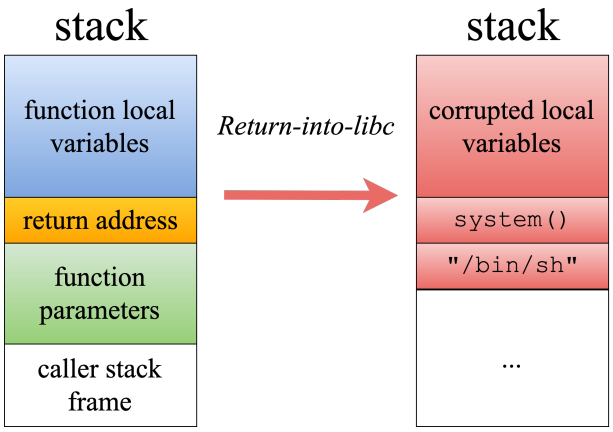


Figure 3.5 Consequences of a *Return-into-libc* attack into the stack.

Uniting the possibility to arbitrarily corrupt memory with addresses and data from a vulnerability, with the functionality of the RETURN instruction, actually grants the attacker even greater freedom than pure calling arbitrary functions. It is sufficient to note that, in such an attack, the address to which a program is to be redirected is any address at all: once arranged the stack appropriately, any data detected as the top of the stack by the RETURN instruction is considered to be the address to return to. Many of the library functions can be used entirely in a general-purpose manner if properly called. But even more can be done, for example, by concatenating pieces of different functions together in such a way as to select only the portions of code actually needed to implement a malware. In a paper by

Schacham *et al.* published in 2007 [67], it is stated that "*in any sufficiently large body of executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able [...] to cause the exploited program to undertake arbitrary computation*".

In other words, from a probabilistic point of view, in a sufficiently large memory image, there would exist combinations of bytes and sequences of instructions that, when concatenated together, would provide the attacker with the same expressive capability as the machine instructions themselves [68]. Exactly as if, to express a written concept, we flipped inside a magazine and cut out all the words needed to write that concept, and then pasted them onto another sheet of paper. Thus comes into play the concept of *gadget*, which is a short sequence of instructions that has absolutely general-purpose behavior, but which must contain an element that allows it to be linked to others to form a chain. This element is given by its termination with a RETURN instruction. In fact, once the attacker has located in memory the addresses of all the gadgets he needs to compose his malware, he can initiate stack corruption by threading them in the order he chooses. Upon execution of the first RETURN, i.e., that of the vulnerable function, control is transferred to the first of the gadgets in the chain. When its execution is finished, its RETURN will trigger an identical mechanism, picking up from the top of the stack the address of the next gadget (Figure 3.6). The cascaded execution can continue at will.

What has been described is what underlies a very powerful attack pattern, which is referred to by the term Return-Oriented Programming (ROP) [69]. This attack is perhaps the most famous example of code-reuse attack. Its success depends on the attacker possessing a *memory dump* of the victim software, i.e., a map that provides the bytes contained in memory and their address. A further point can be underlined about different computer architectures: in fact, in Intel x86 there is no fixed length of instruction encodings, so the gadget search can sometimes be done within other instructions. For example, an x86 program might contain the following instruction:

```
add %bl, -0x3d(%rdi)
```

which adds the value of register `bl` to the byte of memory pointed by the value of register `rdi` minus 61 (`0x3d`). The encoding of this instruction is given by the hexadecimal value `00 5f c3`. However, byte `0x5f` corresponds to the encoding of the `pop rdi` instruction, and `0xc3` is the encoding of the return instruction (`ret`). Given x the address of the above instruction, it will suffice to use $x+1$ to obtain a gadget

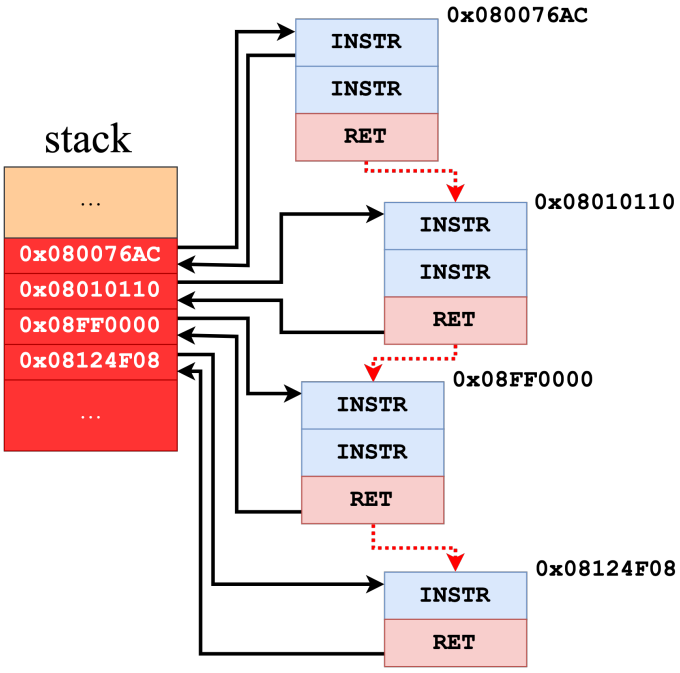


Figure 3.6 Return-Oriented Programming.

that fetches a data item from the stack to put it in the `rdi` register and then passes control to the next gadget. This problem is to be seriously addressed, since a study made some years ago pointed such unintended branches as the majority in a code base: it was estimated that 80% of all libc gadgets are of this nature [70].

On the other hand, in architectures where instruction encoding has a fixed size, this reasoning does not apply, or it is greatly reduced. Fixed-length encoding is more common in more modern RISC architectures, and it is certainly like this in, e.g., major architectures such as ARM and RISC-V, which also are not immune from this attack [71] [72] [73].

Thinking that stack and returns are the only way to implement a code-reuse attack is incorrect. An important paper published in 2011 [74] showed how it is possible to apply the same mechanism using gadgets that terminate with any *indirect jump* instruction (i.e., with the destination encoded not by an immediate, but by a value set at runtime). According to this configuration, it is sufficient to identify within the code a *gadget dispatcher*, i.e., a gadget that acts as a *trampoline* for all the others. At that point, it is necessary for all gadgets to return to the gadget dispatcher before control is passed to the next gadget via an indirect jump. Instead of injecting a set of return addresses on the stack, the attacker can corrupt any data

memory area (such as the heap, the BSS, or the static data region) to inject the gadget address chain, which is named *dispatch table*. This attack paradigm is referred to as *Jump-Oriented Programming* (JOP). The attack method was already presented in a paper in 2010 [75] and takes the name of *Update-Load-Branch* mechanism. The approach was demonstrated as working and applicable to both x86 and ARM architectures. Other variants export the same concept by corrupting, for example, indirect arguments of the CALL instructions [76], or pointers to methods in a class, in use in object-oriented programming languages [77].

3.3 Classical Defenses

Arbitrary Code Execution has been a known problem for more than 25 years. Its knowledge started to spread since a famous first article in 1996, "*Smashing the Stack for Fun and Profit*" [78], in which maybe for the first time it was explained in technical detail how a code injection attack could be carried out by exploiting vulnerabilities in the stack. The title became so famous that in the hacking communities, it is very often quoted and slightly mispronounced, for example, for describing new exploits found. Since then, the computer science community, sensing the great danger of these attacks, has become remarkably interested in how they can be countered.

The first idea on how to neutralize these exploits appeared in a 1998 paper [79], which first proposed modifying the compiler in such a way as to change the way function frames were created. In particular, the paper suggested inserting a memory word between the function's local data and the return address. Such a *sentinel* value is known at the time the frame is created, and must be verified before destroying the frame and returning to the caller. In fact, if during the execution of the function there was an overflow attempt to corrupt the return address, this value is different from the expected one. This serves as evidence to identify an attack, and possibly to invoke an alert and/or recovery procedure. Such is the concept known in the literature as *stack canary*, also called *stack cookie* or *stack guard* (Figure 3.7).

Obviously, the success of such a defense is contingent on the attacker not knowing the value of the canary. The original proposal already addresses this problem, and adopts a canary that is a random word chosen from those in the program's startup library, so that the attacker is forced to check the memory image of the runtime program. The attacker model of this defense is therefore very weak, because it

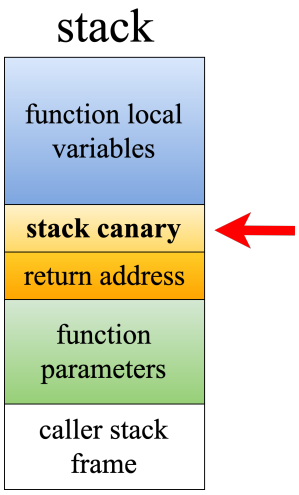


Figure 3.7 The stack canary.

assumes that the attacker is not even able to look inside the program, which he necessarily needs in order to find the addresses, e.g., to mount a code-reuse attack. Another very strong issue is related to processor parallelism: to date, the most common architectures, whether in the mainframe or embedded world, are 32- or 64-bit. This means that every word of memory, including canary, has this length in bits. With today's computational capabilities, brute-forcing a 4- or 8-byte secret has become distinctly affordable. In fact, it is even possible to save this time by finding a way to extract the canary value or change it before the exploit. Format-string attacks [80], which have been known for a long time, exploit a vulnerability in so-called *format strings*, i.e., those used, for example, by the C function `printf()`, which take as an argument a string containing markers that must then be replaced with values found on the stack. Such functions have the characteristic of supporting a variable number of arguments, which translates into the fact that they read from the stack a number of values equal to the format markers they find within their format string. The result is that, by conveniently exploiting an input that then ends up in a vulnerable `printf()`, the attacker may be able to read values on the stack, or even write them (the `\%n` operator in formats is a use case of `printf()` as a write function). Recent work has reengineered the concept of a canary stack, reinforcing it with cryptographic features to make it more difficult to leake and bypass [81]. In any case, the fundamental problem with the canary stack is that it is a defense designed only to guard against attacks on the stack, and it cannot in any way protect against attacks that exploit vulnerabilities elsewhere in memory.

Another historical defense concerns the adoption of *stack redundancy*, particularly for only the return addresses placed at the bottom of function frames. The concept acknowledges two stacks: a normal stack for program execution, and a special stack containing a secure copy of all return addresses that are also put on the normal stack. The goal is to verify, before return, that the return address on the normal stack equals the one on the special stack, which is not affected by a buffer overflow, since it does not store all the other function data (Figure 3.8). Such is the concept of *Shadow Call Stack* [82], or also simply *Shadow Stack*. This is a very powerful concept in defenses against code redirection attacks, so that many modern defenses still contemplate it at their core [83] [84] [85]. As we will discuss later, the shadow stack may be the only way for a security monitor to effectively capture a runtime condition, such as the identity of the last caller of a function. Having a shadow stack certainly requires extra storage allocation, which can be obtained from a dedicated portion of main memory, or enclosed within the processor architecture. In either case, the contents of the shadow stack cannot be accessed as normal data by the program. In the case where it is in main memory, its addresses should be protected from a possible memory vulnerability, or, in the case where the shadow stack is embedded within the processor, it should be handled automatically by CALL and RETURN instructions, transparently with respect to the program. As well as for the stack canary, also shadow stack is a viable defense for stack attacks only.

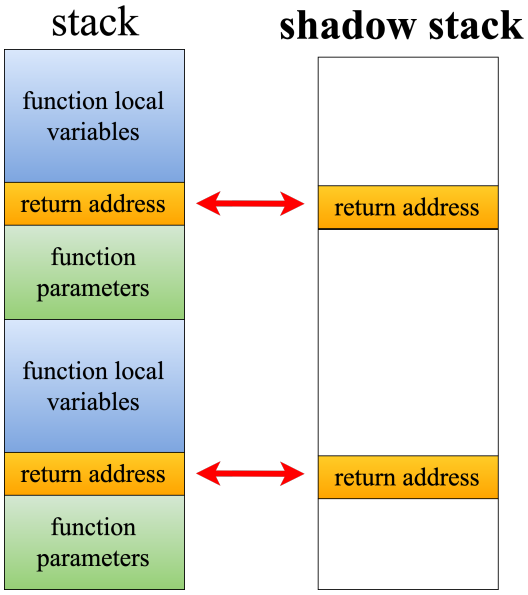


Figure 3.8 The shadow call stack.

While canary stack and shadow stack are two defenses that we might call *deterministic*, *probabilistic* defenses have also been studied in the literature. Rather than forcing a verification based on an identity condition, these defenses aim to confuse the attacker, assuming that it is difficult for him to make the attack under certain conditions. One such contingency may be for the attacker to be faced with a program that randomizes the memory locations of its elements. The randomization may occur at the time of compilation or at the time the program is loaded into memory, or there may also be more fine-grained randomization during the execution of the program itself. In any case, the key concept is to prevent the attacker from building a mapping between the items he is interested in (e.g., gadgets) and their location in memory. This technique is known as *Address Space Layout Randomization* (ASLR) [86]. In code redirection attacks, the attacker often relies on a predetermined order that exists between the sections of the running program (`.text`, `.bss`, `.data`, ...) and even the items themselves within the sections. Randomizing this order at load time is already something that in itself tends to make it difficult for the attacker to mount the attack.

However, just as with the canary stack, ASLR suffers from *low entropy* problems: when applied on a 32-bit architecture, retrieving the base address of items in memory by brute force is not an impossible task. Another problem is that although one can randomize the location of items in memory, a certain order is still to be maintained between instructions, which makes the brute force space reduced to its base addresses [87]. The use of the aforementioned string format vulnerabilities can help alleviate many of the attacker's woes.

Another historical probabilistic approach is that of *heuristic-based* methods, which attempt to model the average behavior of an attack, and what it takes for the attacker to carry out an effective attack. For example, some of these methods control the number of instructions between branches [88] [89] [90], through structures that are present in computer architectures, e.g., Last Branch Record (LBR), or through custom extensions to the processor pipeline. The assumption is that in a typical ROP or JOP attack, the number of instructions executed between jumps is very small, due to the nature of the gadgets (e.g., no more than 5). In [88], authors reported that ROP attacks require at least three gadgets, while in [91], it is said that JOP attacks require at least six gadgets. When an excessively frequent number of branches is encountered, this is indicated as a sign of an attack in progress. Policies can also be refined to such an extent that they look at what individual instructions do between branches, or consider indirect branches only.

The strength of these methods is all in being able to carefully decide the thresholds above which to assume an attack, both to avoid false positives and false negatives. Several works have shown that heuristic methods based on gadget length can be fooled [92] [93] [94]. In fact, the attacker could place intermediate gadgets, i.e., gadgets that are useless for the purpose of the attack but longer than the threshold imposed by the policy, in such a way as to thwart it. On the other hand, policies considering a program under attack when it performs indirect jumps above a certain frequency, can be fooled by *gadget gluing attacks* [90], in which gadgets with direct jumps are chosen to dilute the frequency.

3.4 Control-Flow Integrity

Although many aspects of these techniques have been further investigated in the last two decades, and also employed by many manufacturers families for the defense of their systems (as we will see), none of these approaches is able to encompass all the possible cases of code control-flow redirection on its own. In some cases, the protection is very high, but only designed for attacks targeting the stack; in other cases, the probabilistic reasonings are not sufficient to rule out all the possible paths the attacker can take to succeed in such an attack. The reason is actually simple: none of these approaches takes the problem at its root. Redirecting the flow of control means diverting it from what was originally intended. This means that, at least once, there has existed a trace of all the execution paths allowed for that program, even if only in the head of the programmer, or the designer, or the customer of the system, in explicit or implicit form. None of the approaches seen so far considers an actually very simple reasoning: if the execution of a program never deviates from that set of allowed paths, then there is no flow redirection problem. In other words, if we imagine that one of the properties of a program is the set of execution paths that it can follow (namely, the *control flow*), then the integrity of that property, which concerns the security of that program and, by extension, of the system, is by definition achieved when the program does not take any other path included in this set.

The above is the definition of *Control-Flow Integrity* (CFI), which first appeared in a 2005 paper written by Martin Abadi and his colleagues [95]. The article first discusses security against code redirection attacks as a matter of respecting the

so-called *Control-Flow Graph* (CFG) of the program. Indeed, any program can be represented by a graph $G = (V, E)$, where vertices are sequences of instructions executed sequentially without branches within them. Such elements take the name of *basic blocks*. Connecting the blocks together and acting as edges for the graph are *branch instructions*, i.e., instructions that cause a control-flow transfer different from the naturally intended successiveness for instructions. Branch blocks constitute the arcs and vertices of the control-flow graph (Figure 3.9).

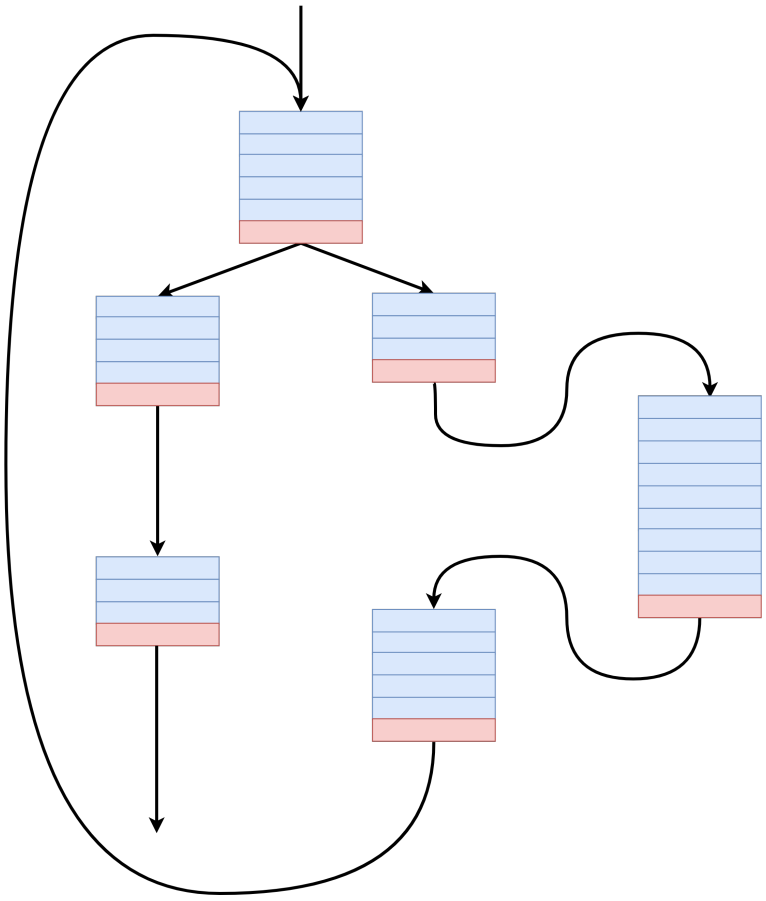


Figure 3.9 A possible slice of a control-flow graph.

But how to obtain such a graph? The paper first presents the concept of *preliminary binary analysis* as a process that, starting from the code of an application before it is executed, extracts its control-flow graph. Such analysis can be done from source code, binary code, or can be completed by execution profiling. Also, appears in the literature for the first time the concept of *labels* as unique identifiers for basic blocks. The principle is the following: if static analysis shows that two basic blocks are

connected to each other by a branch, then the pair of these two identifiers uniquely identifies the branch in turn. Provided we are able to verify at runtime the label of each basic block, it is possible to verify whether each edge is included within the admitted set, i.e., the set E is of the graph G defined above.

For the runtime verification of the labels, the article opts for a *binary instrumentation* performed near each branch. Or rather, near each branch that can be redirected. The principle of *inviolability of direct branches* is thus formalized for the first time. We define *direct branches* as branches having their destination encoded within them, and expressed by an immediate value, which may represent the value of the destination program counter, or more commonly an offset from the current program counter. On the other hand, *indirect branches* are branches having their destination written in a program data, register or memory variable. The latter are the branches that make use of *code pointers*, and examples are the function calls via function pointer, or the returns from a routine, since it cannot know its destination at compile time, and must take it from the top of the stack, or from a register in some other architectures. For their nature, direct branches need no instrumentation, as their argument is fixed and cannot be corrupted.

Before each indirect branch, machine instructions are inserted to check that the destination argument is actually valid and consistent with the analysis made earlier. In the example below showing a snaphsot from an x86 program, it is shown an indirect branch instruction with register argument (`jmp ecx`). According to the static analysis, such a branch has a `mov` instruction as its destination, such as the instruction below (Figure 3.10).

Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction

Bytes (opcodes)	x86 assembly code	Comment
8B 44 24 04	mov eax, [esp+4]	; first instruction ; of destination code

Figure 3.10 Example of an x86 indirect branch to be instrumented, with its destination.

Being a jump destination, the `mov` instruction is at the head of a basic block. The idea is to place the encoding of that basic block before that instruction. Checks at the jump site are to verify that exactly this data is contained in the `ecx` register. If it is not, the program is sent to execute an error procedure. Obviously, to prevent

fetching a constant (which would lead to an illegal instruction exception), the pointer is properly advanced to the actual target instruction.

Bytes (opcodes)	x86 assembly code	Comment
81 39 78 56 34 12	cmp [ecx], 12345678h	; compare data at destination
75 13	jne error_label	; if not ID value, then fail
8D 49 04	lea ecx, [ecx+4]	; skip ID data at destination
FF E1	jmp ecx	; jump to destination code

Bytes (opcodes)	x86 assembly code	Comment
78 56 34 12	DD 12345678h	; label ID, as data
8B 44 24 04	mov eax, [esp+4]	; destination instruction
...		

Figure 3.11 Example of an instrumented x86 indirect branch and destination.

The article introduces for the first time a fundamental assumptions for this type of technique, which from there on will be assumed as guaranteed by any integrity checking technique:

- Values chosen to label basic blocks must be unique, and if used as in the example above, must also be different from any instruction encoding possibly present in the binary program (**UNQ**);
- The code cannot be modified at runtime, otherwise the labels could be changed, or the checks could simply be deleted (**NWC**);
- Executing code from the data sections must not be possible, otherwise it could cause the execution of data that is labeled with expected values (**NXD**).

The paper then formulates another fundamental consideration: such a label-based mechanism can only enforce statically-detectable paths. In other words, if a branch has more than one possible destination, by label checking only it is not possible to verify which of the destinations is the valid one at a given instant of execution of the branch instruction. This issue occurs, e.g., with function returns: subroutines may be called from multiple sites scattered throughout the code, and the label mechanism alone cannot determine which of the valid destinations allowed by the control flow graph is the correct one. The experiments following the formulation in the paper are in fact contemplating the presence of a shadow call stack, isolated by memory protection techniques from the rest of the memory.

A renewed version of the paper published in 2009 [40] brings a final important innovation from the point of view of formal CFI theory. Developing Hamid's previous model for machine languages properties [96], authors demonstrate control-flow integrity as a *safety property*. The running program is described as a succession of states S such that $S_0 \rightarrow S_1 \rightarrow ...S_n$ and $S = \langle M_c | M_d, R, pc \rangle$, where M_c represents the code memory, M_d the data memory, R the set of registers, and pc the program counter. A set of minimal instructions to build any program with is defined (add, $addi$, $movi$, bgt, $jump$, $load$, $store$), and among these is included the presence of infra-code $label$s. Two different types of transitions between states are defined: a normal one denoted by $\rightarrow_n$, and one for the attacker denoted by $\rightarrow_a$. The final theorem states that given G control-flow graph of M_c, and given $I_s(M_c, G)$ as the conjunction predicate between control-flow graph and code (i.e., CFG is always met), then for every i, either $S_i \rightarrow_n S_{i+1}$ and $S_{i+1}.pc = S_i.pc$, or $S_{i+1}.pc \in succ(S_0.M_c, G, S_i.pc)$. Details of the formalism and the proof of the theorem are not reported here for brevity, and are contained in the original paper.

Control-flow integrity is the first of the approaches seen so far that abstracts completely from the type of vulnerability that leads to code redirection, but also from the area of data memory within which the vulnerability is exploited. The label-based approach is an *all-encompassing* one, in the sense that it checks the validity of jumps regardless of how they may be hijacked. The introduction of CFI as such opens the door to the possibility of *fine* checks on the control flow, as opposed to *coarse* checks, as we will discuss in more detail below. The above consideration about label staticity is equally fundamental, as it establishes that it is not sufficient to rely on a pure label-based mechanism to achieve the maximum of granularity.

12 years after the first article, in 2017, when many studies had already delved into the original concept with new proposals, the paper "*Control-Flow Integrity: Precision, Security, and Performance*" [97], which provides a complete systematization for some fundamental control-flow integrity features. The application of CFI policy is first distinguished into two basic and indispensable moments:

1. a pre-execution phase, in which analysis is performed on the code and extraction of the control-flow graph occurs (*offline phase*);

2. a phase concurrent to program execution, in which the branches are validated against what has been defined by the previous phase (*online phase*).

All types of control-flow transfers are then classified. Branches are distinguished not only between *direct* and *indirect branches* (see above), but also between *forward* and *backward branches*. In practical terms, a *forward branch* is a branch whose destination is a code area that has not yet been visited, and which is therefore *forward* with respect to the read sense of the branch area neighbourhood. All indirect function calls belong to this category, but also indirect jumps using a register as an argument. On the other hand, a *backward branch* is a branch whose destination is a code area that has already been visited, and which is therefore *backward* with respect to the read sense of the branch area neighbourhood. All branches returning from a function belong to this category.

In general, all control-flow transfers present in a program necessarily fall into one of these nine categories:

- **CF.1**: backward control-flow;
- **CF.2**: forward control-flow using direct jumps;
- **CF.3**: forward control-flow using direct calls;
- **CF.4**: forward control-flow using indirect jumps;
- **CF.5**: forward control-flow using indirect calls supporting function pointers;
- **CF.6**: forward control-flow using indirect calls supporting vtables;
- **CF.7**: forward control-flow using indirect calls supporting Smalltalk-style method dispatch;
- **CF.8**: complex control-flow to support exception handling;
- **CF.9**: control-flow supporting language features such as dynamic linking, separate compilation.

Categories are then established to classify the *Static Analysis Precision* (SAP) of the performed integrity checks. The following 10 levels for precision on forward branch enforcement are identified:

- **SAP.F.0**: no forward branch validation;

- **SAP.F.1a**: ad-hoc algorithms and heuristics;
- **SAP.F.1b**: context- and flow-insensitive analysis;
- **SAP.F.1c**: labeling equivalence classes;
- **SAP.F.2**: class-hierarchy analysis;
- **SAP.F.3**: rapid-type analysis;
- **SAP.F.4a**: flow-sensitive analysis;
- **SAP.F.4b**: context-sensitive analysis;
- **SAP.F.5**: context- and flow-sensitive analysis;
- **SAP.F.6**: dynamic analysis (optimistic).

On the other hand, for backward branches, the following levels of enforcement are determined:

- **SAP.B.0**: no backward branch validation;
- **SAP.B.1**: labeling equivalence classes;
- **SAP.B.2**: shadow stack.

Over the past few years, proposals on how to implement CFI policies have spread in the research literature, and several applications have been fielded in industry. The present thesis work implies the author's best knowledge of what has been scientifically produced on the subject. Categorizing a vast and varied set of control-flow integrity solutions is not trivial at all. In general, it has been possible to classify each of these examined solutions using the 5 dimensions listed below. In fact, each CFI solution can:

1. be implemented by means of a runtime monitor lying in the hardware domain (*hardware-based*) or in the software domain (*software-based*). In particular, the former may be dependent on a hardware feature already present within the available architectures (*hardware-assisted*), or may require the addition of custom modules (*hardware-extending*); the latter may be implemented directly

and entirely in the running program to be monitored (*autonomous software-based*), or within another software entity running in parallel or in supervision of the program to be monitored (*dependent software-based*);

2. be aware of the program source files (*source-aware*), or act directly on the program machine code, or otherwise be unaware of the original source (*source-free*) [98];

3. be based only on information coming from the static analysis performed in the offline phase (*static policy*) or also on information coming from the execution of the program itself in the online phase (*dynamic policy*);

4. be implemented by instrumentation that occurs entirely in the offline phase (*static instrumentation*), or requiring instrumentation even in the online phase (*dynamic instrumentation*);

5. finely verify each control-flow transfer with respect to its permitted destinations (*fine-grained*), or ensure compliance with soundness conditions of the program, such as only returning to an address following a call, or only forward branching to the beginning of a procedure or a basic block (*coarse-grained*).

The categorization of control-flow integrity solutions is summarized in Figure 3.12.

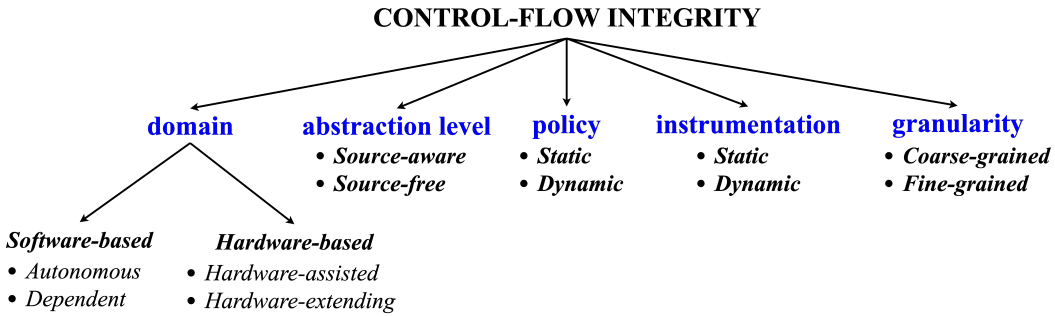


Figure 3.12 Categorization of control-flow integrity solutions.

The first major subdivision that can be made among CFI solutions is the one concerning the domain in which the checks occur. Given a running program to be monitored, the integrity checks can be done by the program itself if properly instrumented, or by a parallel program, such as its parent process, or the supervisor,

or the hypervisor, etc.; In other cases, however, the security checks can be done in the hardware domain, and entrusted to a subset of specific modules of the computing platform, which can be either built into the processor, or connected externally, either offered by the architecture as it is released, or inserted *ad hoc* to enable checks.

3.4.1 Software-based Control-Flow Integrity

Historically, *autonomous software-based* solutions were the first to be investigated. The first paper by Abadi in 2005 speaks in fact about checks inserted within the code of the program to be monitored, and even the earlier classical techniques prior to CFI described checks that augmented the application code (e.g., the canary stack and the relative checks are inserted by the compiler). The first of the famous papers that takes up and revisits Abadi's original concept is a paper from 2011 [99], in which the concept of *Control-Flow Lock* is presented: similarly to labels in the original approach, *key values* are assigned to legitimate branch target locations. Authors propose to instrument the target binary with instructions to lock and unlock control-low paths using these key values. Every indirect control-flow transfer is preceded by a lock operation, i.e., the key value is stored into a buffer. Once program control is redirected to a valid destination, it is immediately succeeded by an unlock operation where the key value is validated, i.e., it is checked against a list of key values that could end up at this target location. If the values match, the key is zeroed out (unlocked) and execution continues as before. When the next control-flow redirection operation must take place, the key buffer is first checked to see if it contains a non-zero value. If it does, an attack is detected, as no legitimate transfer would allow the key buffer to have a non-zero value due to the paired lock-unlock operations.

The same technique is extended by a solution called *Control-Flow Lazily Check* (CFLC) [100], where the state of the control flow is saved in a variable `K`, and there exists a `CF` list, containing all possible consecutive states. Before making an indirect jump, `K` is set to `CF[i]`, and in the valid destination, the value is checked for consistency with the consecutiveness in the list. PCFIRE [101] is another example in which security checks are placed in the source code, precisely at operations that make use of pointers, and for which there might be insecure transfers to other basic blocks. The transfer is interleaved with consistency checks. μRAI [102] is a solution designed to protect return branches only, whereby, through LLVM compiler

modification, a register is reserved to save the XOR between the address of each called function and a key. Upon return, branching to a different caller is prevented by checking this signatures. TypeArmor [103] is a technique primarily designed to protect indirect branches used to link object methods in C++. It follows a label-based instrumentation technique, similar to that of Abadi, and prevents hijacking through type checking, directly applied to the application binary.

Another relevant protection technique applied to the program itself is to augment it with sections that act as a *secure trampoline* for indirect branches. In other words, all indirect branches in the program are switched to direct branches targeting this protected section, which can be directly appended to the executable. From there, an appropriate code verifies the source and destination of each one. The result is the elimination of gadgets through which to hijack the flow. Examples are binCFI [104] and CCFIR [105], two solutions directly applied at the executable level.

The main problem with these solutions may be the cost in terms of increased code area and execution penalties: since the program itself is entirely titled to its controls, it cannot skip additional portions for instructions and/or data. In addition, the normal execution flow itself must be interrupted to allow check execution. CFLC, for example, incurs an average execution overhead of about 400% on the SPEC 2006 benchmarks. For PCFIRE, these increase to 800% at worst.

Dependent software-based solutions, on the other hand, leave the computation required for integrity checks outside the program itself, relying on parallel executions that are usually supported by the presence of an operating system. In the cases of CFIMon [106] and PathArmor [107], the control-flow graph is saved within the kernel, which makes use of some *performance counters* present in the computer architecture, such as the Intel's *Last Branch Record* (LBR) or the *Branch Table Store* (BTS), to verify the compliance of performed branches with the CFG. Kernel checks are invoked when a system call occurs, or by a signal. Of a similar nature is TrackOS [108], designed for real-time operating systems, in which one of system processes there is one dedicated to integrity checks. Check task is executed when all others are idle. The exact same philosophy is also followed by the solution described in [109]. MoCFI [110] is a solution designed for mobile operating systems, in which an OS module aware of the CFG is invoked before every indirect call and return. Instructions to invoke the service are added dynamically at load time. μShield [111] even uses two protection modules, a Runtime Protection Module (RPM) that runs

in parallel with the process to be protected, and a Kernel Protection Module (KPM) that instead lies in the kernel. The former hosts a shadow stack and is given control functions for forward branches, all of which must target a function prologue. The second module is hooked to certain system calls, and performs heuristic checks: for example, it checks that that system call was not called as part of an attack by checking the state of the last branches.

Compared to the previous ones, these solutions certainly have a smaller impact on runtime (just over 6%, for example, in CFIMon and PathArmor), as execution does not have to be interrupted to perform the checks, and the parallel modules are in charge of performing recovery of the possibly violated process. Nevertheless, they could be affected by a *lazy security* problem: in TrackOS, but also in [109], integrity checks are performed when it is possible to schedule the monitoring task, which cannot run at a higher priority, as it has to guarantee the board the schedulability of others. Therefore, violations might be discovered when the attacker's goal has already been achieved. Other problems could be related to their unsuitability for systems that do not support parallel software executions.

3.4.2 Hardware-based Control-Flow Integrity

Among solutions that instead have a *hardware-based* monitor, we can further distinguish two subcategories. One of these includes solutions using a *cryptographic* extension of the processor of the system they are applied to. One example is PATTER (Pointer AuThenTication for kERnels) [112], which is based on the Pointer Authentication extension introduced in ARM processors [113] (detailed later in the Chapter). PATTER is implemented as an extension to the LLVM compiler to add a cryptographic signature to all function pointers and return addresses before they are saved in memory. When these are loaded back for use, the signature is verified. The authentication and branch operations are atomic together, through the use of the `blraa` and `retaa` instructions provided by the ARM PA extension. Of the same nature is Camouflage [114], which additionally uses a local modifier to uniquely identify the signature and prevent reuse attacks on authentic pointers. PAL [115] is a transposition of the same concept to the commodity OS domain (e.g., Linux, FreeBSD). First of all, these solutions are mainly designed for mainframe architectures, as the extension for Pointer Authentication has been introduced by architectures from 64-bit onward, i.e., hardly used in the embedded domain. For

the pointer signature key protection, the use of kernel privileges is required. Unix benchmarks are used to evaluate the performance of these solutions, and the penalty lies between 10% and 30%. These techniques may suffer from TOCTOU attacks [116], because performing load and authentication (or signature and store to memory) at two different times, even with two consecutive instructions, may allow the attacker to change the signature in the middle by exploiting a sudden interruption in execution [117]. This concept will be discussed more below.

On the other hand, the other family of solutions relying on extensions to existing architectures includes techniques that make use of tools for memory protection and address space isolation, or secure enclaves for protecting sensible code. One such extension is, for example, ARM *TrustZone* [118]. TrustZone allows a processor to support two isolated execution domains, *secure* and *non-secure*, each with its own address space and with the secure domain supervising the non-secure domain. In CFI-CARE [119], for example, indirect function calls and returns are replaced by service procedure calls (via the ARM `svc` machine instruction), that make the program trap within the memory area protected by the TrustZone, which performs checks with the help of a shadow stack and a branch table containing all valid destinations, i.e., function prologues. TZmCFI [120] represents a reengineering of the same technique, with the addition of support for exception handling and multitasking. The most obvious problem for these solutions may be due to the fact that not all architectures support a secure enclave like the TrustZone. CFI-CARE also has a performance penalty of 513% measured on CoreMark, while TZmCFI can reach 550% when all attack cases are considered.

In case the TrustZone is not available, the protected area where to perform checks and store control-flow graph information can be created through the use of a *Memory Protection Unit* (MPU). This is the case in RECFISH [121], a solution involving the addition of a specific `.cfi` section for performing checks, separate from the `.text` section that contains the normal code. In particular, every function prologue and epilogue, as well as every indirect branch within functions, is instrumented directly at the binary level to become a direct jump to this new section. Similar protection is applied by Silhouette [122], which uses the MPU to implement a shadow stack, in which the value of the *link register* (`lr`) is saved whenever there is a function call. For forward branches, a label-based system is used to enforce transfers only to function prologues. Instrumentation is implemented through modifications to the LLVM compiler. Kage [123] represents an improvement of Silhouette, with

support for real-time operating systems. Here, the runtime overhead is contained in varying percentages, ranging from 3% of Silhouette to 30% of RECFISH on the BEEBS benchmarks. The real big problem on these types of solutions concerns *schedulability*: even being originally designed for real-time operating systems, they cannot always guarantee task schedulability. For RECFISH, for example, 15% of workloads are reported to be unschedulable. This is an opposite issue to software-based dependent solutions, where guaranteed schedulability comes at the expense of security.

Then, there is a very large family of proposals that introduce custom modifications to the architecture of processors or computing hardware to perform control-flow integrity checks. A subset of these integrates modules within the architecture for encryption of addresses and/or target instructions, that are sensitive for CFI purposes. In LEA-AES [124], authors propose to insert a module that automatically encrypts the routine return address before pushing it onto the stack at the call time, and that decrypts it when the `ret` is executed. Such an on-the-fly pointer encryption/decryption mechanism is also presented in HW-CDI [125]. In LEA [126], a slightly different approach is instead adopted, which involves the encryption not of the addresses, but of the indirect jump target instructions. This encryption is done at load time, when the code is loaded into memory. At runtime, whenever an indirect branch is performed, the processor automatically decrypts the target instruction, and if an invalid instruction results from the decryption (i.e., the attacker redirected the branch elsewhere), the execution stops. In both cases, the key is generated from a PUF of the processor [127], which can produce a value that does not have to be saved in memory, but can be produced each time needed, since it encodes a physical property of the electronic artifact that cannot be replicated. The platform used to test the solution is the RISCY core [128], a 32-bit RISC-V processor. A very similar but improved approach also appears in HCIC [129], designed for SPARC processors. In SOFIA [130], the concept is applied to the LEON3 processor of the same family. Here, the binary is instrumented in such a way that it contains encrypted instructions that then must be decrypted at runtime. Specifically, entry points of functions are encrypted with the address of the caller, and return points are encrypted with the address of the return instruction in the callee. Since after an indirect branch the module decrypts the instruction after fetch, a branch that targets an instruction outside these encryption pairings results in an illegal instruction. A not dissimilar mechanism, improved in

performance, is presented in FH-CFI [131], Where encryption involves the use of HMAC-SHA-256 and the technique is tested on an ARM-based system.

Another group of solutions verifies CFI by calculating the *hash* of consecutive basic blocks that are executed. If it is true that CFG represents the set of paths allowed for a program, then that path will be formed by the consecutive execution of certain basic blocks. Each of these paths can be represented by a *digest* on the program counter value, which updates upon execution of each basic block. Obviously, the monitor has the list of all hashes representing valid execution paths. An example of this is LO-FAT [132], which integrates control modules within the pipeline of the PULPino microcontroller [133]. The solution presented in [134] also uses a processor-integrated monitor that continuously reads the *instruction register* and the program counter. Then, extracted instructions are digested by the hash module, and the start address of the current basic block is compared with the pre-extracted valid start address in monitor model memory. As well presented on RISC-V platforms are CCFI-Cache [135] and the solution described in [136], where, however, control modules are placed between the processor and the instruction cache, and sniff the instruction address bus.

Instead of introducing modules that verify the integrity of the control flow in a totally transparent manner, there are solutions proposing modifications to the processor's *Instruction Set Architecture* (ISA), to support special instructions for interacting with data structures used for checks. These data structures are located inside the processor (in fact, if they were located outside, there would be no need to use other instructions than those provided for data transfer, and there would again exist a problem of protecting sensitive CFI information). For example, Sullivan et al. [137] modified the SPARC LEON3 processor by introducing two security-based private data structures, the *Label State Register* (LSR) and the *Label State Stack* (LSS), and 5 additional instructions to control them. When an indirect call is to be performed, `cfibr lbl` is executed first, which pushes a unique label `lbl` on top of LSS, indicating the call site. `cfiprc lbl` then saves into the LSR a unique label for the function location. The first instruction of each function is always `cfichk lbl`, which verifies that the content of LSR is the label `lbl`. At return time, the processor enters a state which only accepts `cfiret lbl`, otherwise it is blocked. Therefore, all the instructions following an indirect call are `cfiret` in such an architecture. Indirect jumps are instead instrumented with `cfiprj lbl`, which store `lbl` in the

LSR, and at all possible destinations, `cfichk lbl` verifies that the content of LSR is in fact `lbl` (Figure 3.13).

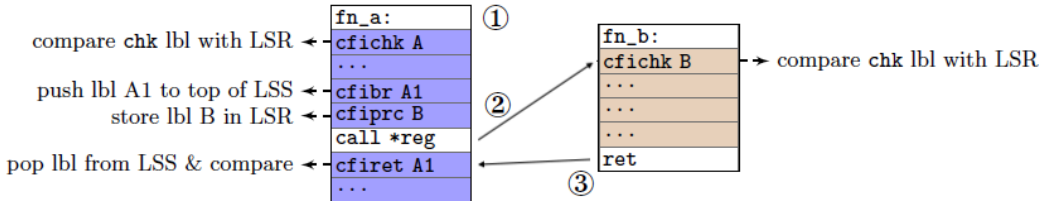


Figure 3.13 Tracking of CFG in Sullivan *et al.* solution [137].

HAFIX [138] follows a pattern almost identical. It forces the first instruction of each function to be a custom instruction `CFIBR`. This instruction loads the label of the function into a dedicated memory area, called *label state memory*, to indicate that the function is active. Internally, direct and indirect call instructions lead to a processor state switch in which the processor only accepts `CFIBR`. To deactivate a function, HAFIX uses the `CFIDEL` instruction which effectively removes the label from the label state memory. Hence, `CFIDEL` instructions are executed just before a function return instruction. Finally, only returns to the `CFIRET` instruction are permitted, in particular those `CFIRET` instructions that define a currently active label in the label state memory. A finer, faster, and more accurate version of HAFIX is presented in HCFI [139], in a paper of the following year. With respect to an unmodified LEON3 core, the runtime overhead is less than 1% on their FPGA implementation for the SPEC INT 2000 benchmarks. FIXER [140] is instead a technique implemented using the RISC-V RocketChip core generator [141], that uses secure versions of `call` and `ret` (called `cfi_call` and `cfi_ret`), that are implemented using the custom instructions already provided by the architecture to interface with a coprocessor (in this case, the secure monitor), without making any changes within the processor. The monitor is basically composed of a shadow stack and a *policy matrix memory*, which contains a logic network necessary to detect a possible match between the caller and the callee addresses in indirect function calls. Because of its characteristics, FIXER can actually also be used in contexts where modules external to the CPU are present, e.g., implementing the monitor in ASICs or FPGAs.

Rather than directly invading processor design to connect, for example, security modules to the instruction bus or internal pipeline registers, some solutions experiment with the usage of *debug interfaces* to monitor the running program. One case is CONVERSE (*CONtrol-flow VERification SystEm*) [142], in which the verification

module is connected to the NEXUS debug interface present in a Freescale microcontroller. Verifications revolve around an internal data structure in the verification module, called *Branch Destination Table*. Each entry in the Branch Destination Table contains a source address (i.e., the address of the branch instruction) and a destination address (i.e., the address of the branch destination). Captured debug traces are intended to verify whether all pairs of program counters consecutive to each other are compliant with this table. In FastCFI [143], an external FPGA is connected to an ARM Cortex-A9 via the Trace Port Interface Unit (TPIU). From the analysis of the CFG, a synthesized Verilog is generated on the FPGA, which then reads directly from the debug interface of the traced processor, and validates transitions between PCs, according to the internal state machine design, which mirrors that of the program.

Other solutions requiring hardware extensions cannot be categorized into any of the previous subcategories, and are still based on monitors that need close proximity to the processor to read its runtime information directly, and understand whether control-flow violations are occurring. In [144], the integration of a branch regulation module within the pipeline of x86 processors is proposed. By simultaneously reading the instruction register and program counter, it makes sure that each call instruction targets the prologue of a function, as well as each return branch actually returns to its caller through the adoption of a shadow stack. OCFMM (*On-Chip Control-Flow Monitoring Module*) [145] is the exact transposition of the same concept, implemented as an external monitor to the LEON3 processor. In BB-CFI [146], a module able to read the queue of instructions executed by a SPARC processor, validates them against control-flow integrity by using a table of basic block entry addresses, and a return address stack. BBB-CFI [85] imagines to apply the same mechanism by reading from the Last Branch Record (LBR) table of Intel processors. Very similar is also DExIE [147], which brings this concept to RISC-V architectures for IoT systems, validating branches based on a state machine synthesized from the CFG.

In general, all hardware-extending solutions have a very low cost in terms of execution time compared to software or even just hardware-assisted techniques. Customizing the architecture clearly tends to minimize the effort required by the software. The runtime overhead is always around 1%, except in cases where changes are applied to ISA: here, a slightly higher overhead is encountered. The competition between these solutions is all about the amount of additional resources required

in terms of chip area. Even when no changes to the architecture are explicitly requested, but external modules are added to act as monitors, overhead is to be taken into account. In the vast majority of cases, these proposals are made on open architectures (such as SPARC and RISC-V), i.e., requiring no license to be used. Similarly, the experiments conducted are in no case made using ASICs, which are replaced by more malleable FPGAs. The use of FPGAs can give a good indication of the amount of additional resources in terms of gates and memory units (flip-flops and registers) to be added to the architecture; however, it is less accurate when additional consumption and power are to be estimated. Mostly, these proposals are intended to give a future direction to be taken in computing architectures design. In some specific cases, however, special-purpose systems are implemented on FPGA for flexibility reasons. Here, a valid porting of these solutions may already be a ready defense.

3.4.3 Abstraction Layer

As illustrated in Figure 3.12, another dimension used to categorize CFI solutions is the level of software abstraction at which they are placed. A CFI solution is in fact applied following a static analysis, which certainly gathers information about the program's CFG and possibly applies instrumentation. Such instrumentation consists of additional code, that must either check the validity of the flow, or trigger the modules that are intended to do so. In the case where source code of an application is available, one may consider putting such checks directly into the source. Such solutions are called *source-aware*. For example, Control-Flow Lock and CFLC are part of this category: here, it is the source itself that is in charge of its own control-flow verification, as variables and data structures are inserted that are directly managed by the program. In particular, the functionality of the program remains the same, and the code is interspersed with controls where needed. However, source-aware solutions are also those that contemplate a parallel execution dedicated to checks, for which a dedicate task in the source code has been written for this purpose. This is the case with TrackOS and the solution presented in [109]. Finally, we can consider source-aware all solutions that are applied at the compiler level. In many cases, the protection is applied when the *intermediate representation* of the program has already been generated, i.e., an abstraction present in compilers that corresponds, in short, to a pseudo-Assembly code, to be then converted into the target

machine language. At that point, it is possible to deal with a program description that is already described in machine terms, but still lacks all the complications of executable binary code: for example, references to objects (data and functions) are still absolute and not relative to the location where they are referenced. In all cases, such techniques cannot be adopted when the source of the application is not known. Examples are all solutions that make use of ARM Pointer Authentication, such as PATTER, Camouflage, and PAL, but also Silhouette, which makes use of the MPU, and TZmCFI, which makes use of the ARM TrustZone. μRAI, while using a native and completely software-based encryption scheme, is implemented through modification of LLVM steps. Even when it is necessary to extend the hardware with custom modules, which operate via solicitations due to custom instructions added in the ISA, may be required to modify the compilation toolchain. This is what happens, for example, in HAFIX, HCFI, and Sullivan's solution. The main problem with these solutions is their non-usability in cases where sources are unknown on not accessible. To fix the ideas, we can think to the example of a security team in charge of protecting the infrastructure of a company, which has several systems running programs with expired support, and it is impossible to trace the source files.

Then, there is another category of solutions that instead work regardless of knowledge or possession of the source files of the program to be protected. Such techniques are called *source-free*. Most of these techniques take the compiled binary of the program, and modify it in such a way as to add the verification instructions. Such is, for example, the approach of Abadi's original work. But binCFI, CCFIR, and TypeArmor also modify or instrument branch instructions directly at the executable level to introduce checks into the program itself. PathArmor and μShield use binary instrumentation to invoke kernel checks. Also, part of this category are so-called *trampoline-based solutions*, i.e., those replacing all indirect branches with branches directed to a separate instrumented area, such as CFI-CARE and RECFISH. Here, the advantage is skipping problems arising from address relocation: the layout of the machine code does not change if the instrumentation redirect branches to an area outside the original code section. On the other hand, solutions not adopting a trampoline area and inserting checks in the same area of the original code, must necessarily deal with the shift of all static references that are encoded within the instructions.

Source-free category also includes hardware-based solutions, such as LEA or SOFIA, that use instruction or target encryption techniques, applied directly to the

binary. Instructions for interacting with an external monitor may be directly added at the binary level, as in the case of FIXER, or even there may be no need at all for a knowing the program code, in the case where monitors directly read the execution context from the processor modules. This is the case with BB-CFI, BBB-CFI, but also with all solutions based on hash evaluation of basic blocks, shadow stack, or monitoring via debug interface.

3.4.4 Policy

Solutions can then be distinguished depending on the stage at which the information leading to policy decisions is originated. A CFI solution is said to have a *static policy* when all the information to derive its decisions at runtime has already been collected in the offline phase. On the other hand, when information to determine some decisions can only be known in the online phase, then that solution is said to have a *dynamic policy*.

Most of the solutions identified in the literature need information that can only be extracted at runtime. For example, all solutions that involve a shadow stack are dynamic policy solutions, since it is clear that it is not possible to determine the exact return location to the last caller through information captured purely in the offline phase. Solutions based on external custom monitor, debug interface, hash calculation, and ISA extension have all dynamic policies. Instead, examples of purely static-policy solutions are those that make use of Pointer Authentication extensions: in this case, there is no need to extract any information from the running program. CCFIR, binCFI, and TypeArmor also rely on pointer usage protection that is fully determined in the offline phase.

3.4.5 Instrumentation

CFI solutions can be differentiated according to when code instrumentation occurs. The vast majority of solutions involve instrumentation in the offline phase: through compiler modification or patches applied directly to the executable, a *different* program is in fact produced in the offline phase than the original. These are called solutions with *static instrumentation*. In all analyzed cases, for example, where a cryptographic or memory isolation module already present in processors is used,

this type of instrumentation is adopted. Abadi's classic solution, Control-Flow Lock, CCFIR, binCFI, but also PCFIRE, TypeArmor, or techniques that extend ISA such as HAFIX and Sullivan's, are all in this category. The techniques analyzed that instead use instruction or target address encryption, namely LEA, LEA-AES, SOFIA, FH-CFI, and HCIC, all apply this encryption at the time the program is loaded. Since the loader activity can already be placed in the online phase, these can be all called with *dynamic instrumentation*. The instrumentation is also dynamic for MoCFI, which introduces instruction for invoking verification software modules only at load time. The same applies for PathArmor. Then, there are solutions that do not need any instrumentation at all, as the monitor is directly able to supervise the running program: this is the case with solutions using the debug interface, those based on basic block hash computation, or with a monitor connected to the pipeline for verification.

The extent of instrumentation is in fact what determines the overhead in terms of execution time and code size. In solutions where no instrumentation is required, or the instrumentation consists of exchanging a normal instruction for an instrumented instruction, no overhead is actually observable, except possibly due to architecture changes that lengthen instruction execution time. In BBB-CFI, for example, an overhead of 0.0003% is reported, which is zero in practical terms.

In general, instrumentation tools can be architected undependently of the applied protection, that can be customized. Also, the analysis part can be useful for any type of enforcement. Literature have offered several example of such analyzers and instrumenters, both for static and dynamic instrumentation. PEBIL [148] (*PMaC's Efficient Binary Instrumentation Toolkit for Linux*) is a binary instrumentation toolkit that enables the building of custom instrumentation tools. instruments the executable statically by placing a branch instruction at the instrumentation point that will redirect the execution to the instrumentation code. The instrumentation code saves the program state, and will perform the actions required by the instrumentation tool. In order to support ISAs with variable length instructions, PEBIL takes care of relocating and transforming code for each function in order to make instrumentation code reachable from the instrumentation points. PEBIL only supports x86 architecture.

Pin [149] as a dynamic binary instrumentation framework developed by Intel. Pin aims at providing a platform for building program analysis tools, called *pintools*. A pintool consists of three type of routines: *instrumentation routines*, that inspects

the application's instructions and insert calls to *analysis routines*, invoked when the program executes an instrumented instruction; then, there are *callback routines*, called when an event occurs. PIN will use a *Just-In-Time* (JIT) compiler in order to apply instrumentation at runtime to the application. The framework is provided for Windows and Linux platforms, but works only for program compiled for the Intel architecture, making it not suitable for other architectures.

Dynist [150] is a binary instrumentation and analysis framework leveraging both the static and dynamic approach for code instrumentation. Dynist provides a representation of the program in terms more higher level structures, like function's basic blocks, and the user leverages this representation in order to instrument code in *anywhere* in the binary. Dynist emphasize *anytime* instrumentation by providing both static and dynamic instrumentation capabilities. The user is allowed to modify or remove instrumentation at anytime. In addition, the analysis techniques performed allows to insert new instruction without influencing non-instrumented code, that can run at native speed, in order to minimize the overhead in performances.

3.4.6 Granularity

But the most important distinction that exists between CFI solutions concerns the level of *fineness* of its checks, i.e., how deep checks actually adhere to the CFG. All analyzed solutions in fact can be distinguished into two broad categories: some solutions tend to consider as valid an attack pattern that makes intensive use of short gadgets, thus composed of the last bench of instructions in the epilogue of a function, or before an indirect branch. To protect against such an attack pattern, it may be sufficient to verify that the program respects some *semantic consistency* properties. One of these might be ensuring that all return branches have an instruction following a call as their target. In the case of indirect function calls, instead, one might simply verify that the target is the prologue of a function. This certainly removes many of the weapons that are in the hands of the attacker to complete his job. Such solutions are called *coarse-grained*. Coarse-grained solutions account for about 75% of the solutions analyzed by this study and presented in this Chapter.

There are coarse-grained solutions of all kinds. binCFI, for example, inserts instrumentation that forces returns only to branch to an instruction following a call. In CCFIR, indirect call and jump instructions can branch to any function pointer

stubs in the Springboard. In some ISA modification mechanisms, such as HAFIX and HCFI, targets of indirect function calls must be present in an allowed set, but there is no enforcement on which function can or cannot be called from a given branch. In some target instruction encryption mechanisms, such as LEA or LEA-AES, there is no enforcement against reaching a valid target from a valid source, but not for that source. BB-CFI, while claiming to be fine-grained, actually only checks that all forward branches have the beginning of a basic block as their destination. The same applies for BBB-CFI. Even in debugging schemes such as CONVERSE or OCFMM, any return branch can return to any of its callers, according to static analysis. In CFI-CARE, it is explicitly said the call must only target the beginning of a subroutine prologue, among those included in a table of allowed entry points. The same applies for Silhouette. In some hash-based solutions such as CCFI-Cache and LO-FAT, all valid execution paths, compliant with a valid hash, are considered good, so even when a function does not return to its actual caller, no violation is detected. In general, all those methods that do not involve a shadow stack are necessarily coarse-grained, because while it is possible to verify the return of a callee to its last caller, they do not. For some time now, studies have begun to appear demonstrating the feasibility of code redirection attacks even in the presence of coarse-grained CFI solutions. One example is [151], published in 2014, in which CCFIR and binCFI are tested, and enough gadgets are found to launch a shellcode attack, even if transiting the protected area. Another paper in 2018 [152] presents a new attack paradigm, *Function-Oriented Programming* (FOP), and shows that it is possible to achieve Turing-complete malicious execution even under policies that allow indirect forward branches only to function prologues.

Instead, *fine-grained* solutions are those adopting all possible means at a monitor's disposal to verify that a branch is actually targeting *its* valid destination, and not, in general, *one* valid destination. This means, for example, recognizing the last caller of a function, so that the return branch goes back exactly where it should. Or, making indirect function calls reach only the functions that are forecasted to be called from that point according to the static analysis, and not *any* function.

The granularity of a solution does not necessarily descend from its family. For example, SOFIA is a solution very similar to LEA or LEA-AES, but it includes a method for authenticating the last caller and destination of a branch when it has multiple sources. FH-CFI also follows the same protection scheme. Sullivan's method is a fine-grained solution, because a forward branch can only reach its

intended and predefined destinations. Then, all solution using pointer authentication facilities, such as PATTER, PAL, Camouflage or HW-CDI, are fine-grained, because by definition, if through authentication a pointer to the code cannot be corrupted, that will necessarily point to the intended destination. RECFISH, which uses a trampoline method, correctly checks that each forward branch goes to a destination among those valid for it, and contains the notion of a shadow stack. Based on the categories identified so far, it can be concluded that a purely static policy solution cannot be also fine-grained at the same time, as to be so, it necessarily needs to make all return branches have the last caller as their target, and this information is only known at runtime.

The distinction between coarse-grained and fine-grained solutions appears as self-evident in a scientific discussion such as this, but it is not this evident in the real world. In fact, what is rightly regarded as unacceptable because of its security consequences from a scientific point of view, may instead be something entirely positive in a system in which, for example, forward-branch enforcement to function entry points only reduces the attacker's potential to tolerable levels. For example, in the domain of mainframe machines (e.g., PCs, tablets, servers), where general-purpose operating systems dominate, coarse-grained solutions very often have no viable alternative, because accurate static analysis of processes is practically unfeasible. The programs running on these systems are complex and dynamic, the environments are highly multicore and multitasking, and it is therefore unfeasible to ensure CFG compliance for individual applications. At the same time, however, the adoption of good coarse-grained solutions helps to significantly reduce the danger, and avoid most attacks. This concept is not hard to imagine if one thinks of ROP attacks that attempt to achieve full control of the machine, and that seek their gadgets in function epilogues. In these cases, adopting a coarse-grained solution that enforces branches to function prologues only, or even just returns to after-call locations, is already a very valid defense. This is so true that, in this domain, the very definition of fine-grained often assumes a different connotation, and even merely guaranteeing the return of a callee to one of its possible callers is included in a fine-grained view of control-flow integrity.

3.4.7 Commercial Off-the-Shelf Solutions

Right for what was said in the last words of the previous Subsection, it is important to mention what the system manufacturers and the company domain have been able to produce so far to deal with the problem of code redirection attacks. First of all, it should be mentioned that in all practical cases, systems today have the basic protections that were already laid at the foundation of Abadi's first article, i.e., executing data from writable pages, and modifying executable code once loaded is forbidden. In addition, most compilers, such as GCC or LLVM, today support by default the inclusion of a stack protector (a.k.a. stack canary) via the `-fstack-protector` option, which blocks all ROP-based attacks by itself.

In 2016, Intel introduced the so-called *Control-Flow Enforcement Technology* (CET) in its chips [153]. The technology basically has two main features: the presence of a shadow stack and *Indirect Branch Tracking* against forward branch hijacking. When shadow stack is enabled, the Intel instruction `CALL` pushes the return address on both the data and shadow stack. The opposite `RET` instruction pops the return address from both stacks and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP). The shadow stack is not located within the processor, but in dedicated memory pages appropriately flagged as such. Data transfer instructions to and from this area are not allowed. On the other hand, as far as indirect forward branches are concerned, all their targets can be marked via the new `ENDBRANCH` instruction. This instruction opcode is selected to be one that is a `NOP` on legacy machines such that programs compiled with `ENDBRANCH` new instruction continue to function on old machines without the CET enforcement. Whenever an indirect forward branch is executed, the CPU enters a special `WAIT_FOR_ENDBRANCH` state that expects to fetch an `ENDBRANCH`, otherwise a protection exception is raised. While the shadow stack can be enabled at the machine level, the `ENDBRANCH` instructions must instead be placed inside the binary at compile time.

As for the ARM family, support for so-called *ARM Pointer Authentication* (PA) [113] has been included since version 8.3 of the ISA. The idea behind this feature is that the 64-bit address space is unused in most of the 64-bit deployments, as the amount of addressable bytes with 64 bits is certainly out of needs for actual applications. This leaves the topmost bits of all pointers unused, and the intended use of PA is to repurpose these bits to store a cryptographic message authentication

code (MAC). The MAC is from 16 to 32-bit long depending on the configuration, and calculated in hardware over the pointer value. When reloading this pointer from memory after having stored it, evidence of tampering can be easily discovered by checking the pointer signature. Signature creation and verification are performed using a hardware implementation of the QARMA algorithm [154], and are triggered by the explicit use of variants of two new instructions, `PAC` and `AUT`, respectively. The authentication code results in being computed as:

```
PAC = QARMA(key, pointer, context)
```

where `context` is a parameter chosen in relation to the execution context, for increasing uniqueness of the signature.

ARM PA also stocks a general purpose MAC primitive — the instruction variant `PACGA Xd, Xn, Xm` which computes PAC for a value stored in register `Xn`, using a modifier stored in `Xm`, and a key, and stores the result in the upper 32 bits of `Xd`.

The original presentation document discussed the possible issue of *pointer substitution*, which is the most likely kind of attack to be used against PA. Pointer substitution means that the attacker replaces an authenticated pointer with another, which could be obtained, e.g., through a memory disclosure vulnerability. An appropriate use of *local modifiers* can lower the problem. For example, the modifier could be chosen in order to mix spatial and temporal information (e.g., the stack pointer and the loop counter). To present the problem, authors describe how an application using the stack pointer as context for return addresses could be exploited to collect several authenticated pointers for different values of the stack pointer, and reuse them in the program to string together a series of gadgets to execute a ROP attack. Therefore, accurate and smart choices need to be made when deciding the modifiers for the PAC generation, to reduce as much as possible the threat of pointer substitution.

Another possible issue could be related to guessing the PAC through brute-force attacks. The feasibility of such attempts depends on the system, as the size of the PAC depends on how many unused bits are left over in the pointer, which depends on the addressing space size. The size of the signature could range from 3 bits up to 31, which implies making further considerations when using such a mechanism. ARM PA has been adopted, reproposed and reshaped in many literature works [] that tried to maximize its benefits and minimize its drawbacks. Its usage has also

been proposed for addressing the memory corruption problem in its nature, e.g., by authenticating any CPU register that is automatically spilled in memory due to *register pressure* in the compilation phases [155].

Since the release of Windows 8.1, Microsoft has adopted its own control-flow integrity mechanism, called *Microsoft Control-Flow Guard* [156]. The option can be entered via the proprietary linker, for example, using a version of Visual Studio higher than 15. In short, for each process, a bitmap containing all valid target addresses is created. The presence of an address in the bitmap is then checked at runtime each time an indirect branch is performed. The `_guard_check_icall` function in fact instruments each indirect branch to verify the presence in the address in the bitmap, and terminate the program on the contrary. The solution is clearly coarse-grained, but Microsoft has already announced an enhancement (as yet unreleased), called *eXtended Flow Guard* (XFG), the details of which were for now only previewed at an event in Shanghai in 2019 [157]. Under such an enhanced scheme, the targets of an indirect branch would be narrowed down to a subset of all possible targets through a signature scheme. Before making the indirect call, a register would be used to save the signature of the target function. Upon arrival at the target, the function signature would be computed and verified.

In 2014, Google also presented in an article its CFI technique, called *Indirect Function-Call Checks* (IFCC) [158], as an extension of the GCC and LLVM compilers. The mechanism consists of generating jump tables for indirect-call targets, and adding code at indirect-call sites to transform function pointers, ensuring that they point to a jump-table entry. Any function pointer that does not point into the appropriate table is considered a CFI violation, and is forced into the right table by IFCC.

Chapter 4

Control-Flow Integrity for Embedded Devices

After the discussion of the CFI principle and having listed and analyzed the main solutions offered by the state of the art, the thesis now comes to address the problem of offering such security guarantees to the systems discussed in Chapter 2, the so-called *resource-constrained embedded systems*. For the purposes of this discussion, the terms *embedded system*, *embedded device*, and *resource-constrained device* are interchangeable. After discussing the common attack scenario in which they are immersed, we will move on to look specifically at a set of open issues, which are in our view fundamental to the conjunction of the embedded domain with CFI. Some of the CFI solutions that explicitly target these systems are then presented to highlight their strengths and weaknesses in relation to the identified issues.

4.1 Attacker Perspectives

In the previous parts of this thesis, we have already discussed the nature of embedded systems: their extreme pervasiveness in the reality around us, their reduced computational capabilities, and their need to accomplish a specific task. The primary goal of designers is to minimize size, weight, power consumption, cost, and to simplify operations as much as possible. As anticipated in the previous Chapters, the scene is largely dominated by *microcontrollers*, i.e., systems integrated on a single chip that include a microprocessor, memory devices, and controllers for peripherals of

various types. We have also talked about the massive presence of individual computational units within daily life realities, such as, e.g., a modern vehicle. Electronic units control the vehicle in a wide variety of operations, from infotainment systems projected on the in-vehicle monitor, to the extremely critical functionalities for driver assistance. Because of their high degree of integration within the objects they control and their hard accessibility, these systems could disregard a regular software update process. As a result, vulnerabilities within the control code might not be resolved for a long time, leading attackers to find and exploit them for whatever purpose. In addition, such systems have today improved with not only intra-vehicular, but also *inter-vehicular* connectivity, with an increasing number of radio-frequency interfaces such as Wi-Fi and Bluetooth, which naturally constitute an avenue on which the attacker can stand. Given all these elements, it is impossible to ignore the issue of security for these devices (Section 2.2).

An undoubted feature of resource-constrained systems is that they have software designed to adhere as closely as possible to the capabilities of the hardware platform on which they are implemented. More than others, this feature makes them prone to the use of languages that translate directly into machine code for the application: in place of today's more common interpreted languages such as Python, embedded systems designers still prefer the use of C-like languages, and in some specific cases, even choose to program in Assembly. Especially in the many cases where standard programming norms are not followed [25] [24], the use of the C language make the programmer introduce vulnerabilities causing memory corruption problems. As already discussed, buffer overflow tops the list of the most common and most dangerous vulnerabilities still in 2022 [62]. If we combine this with the diffusivity of the C language in the embedded domain, and the ubiquitousness of these systems in contemporary society, we find a very greedy scenario for the attacker.

The fact that embedded systems were also prone to code redirection attacks has been known since the early days. Picking up on Shacham's original thesis [67], in 2010 the possibility of ROP attacks on the ARM architecture was theorized [72], under the reasoning that if the binary code of libraries for a given system can be inspected, then there is an algorithm that can determine whether such code can contain the gadgets needed for a successful attack. The only difference from the classical gadget search mode in the x86 environment is that in ARM instruction encoding has a fixed length, which in the microcontroller domain is 32-bit, with a few 16-bit exceptions for the Thumb subset [159]. This implies that the search

for gadgets cannot generally be conducted throughout the entire instruction space, but must be specifically looked for in the actual indirect branch instructions, which cannot result as subsets of other instructions. Another peculiarity for the ARM family is the presence of the *link register* (LR), a fundamental register of the architecture that contains, for each call, the address of the instruction to return to once the callee is finished. Unlike the x86 family, where the machine instruction CALL at once pushes the return address onto the stack and jumps to execute the subroutine, and the RET instruction at once fetches the return address from the stack and jumps back to execute the caller, in ARM the function call is implemented via a *branch and link* (BL) instruction, which copies the return address into the LR register before branching. The return to the caller is thus implemented via a simple indirect branch that has LR as its argument. By itself, this would seem to reassure against the possibility of an attack, since a register in the CPU cannot be corrupted by a memory vulnerability. However, the link register is unique in the architecture, so nested call functions necessarily result in a *stack spill* of the old link register to make room for the new one. The epilogue of a function in ARM is similar to what shown below:

```
LDMIA SP!, {R4,R5,R6,R7,R8,R9,R10,R11,LR}
BX LR
```

where before the return to the caller is made, the registers that the subroutine used for its calculations are loaded from the stack, and the value of LR is also restored. The LDMIA instruction stands for *Load Multiple and Increment Address*: as can be seen, the register used as the address to self-increment at each load is the *stack pointer* (SP). The BX instruction is the one that finally implements the indirect branch to the value of a register.

In the RISC-V open architecture, which was developed more recently and is today central in a great range of embedded applications, routine call handling is almost identical. The *jump and link* (JAL) instruction branches to the function after copying the return address to the register specified by the first operand, by convention the X1 register, also aliased as RA (return address). The return, encoded in the RET pseudo-instruction, is actually a jump to register (JR) with argument X1.

A work of 2017 [160] tried to mount a ROP attack on a device from Texas Instrument, the Tiva TM4C123GH6PM [161], equipped with an ARM Cortex-M core. The authors were able to effectively perform two ROP attacks to erase and program the Flash memory of the device at will, using a very small set of gadgets and

simply relying on store instructions targeting the Flash control logic. The gadgets were retrieved from the TivaWare software [162], just 1 KB thick on the device Flash and representing the basic configuration part of the firmware. They also managed to detect the presence of a gadget for each machine operation, demonstrating the Turing-completeness of the small software.

Succeeding in such an attack presupposes two elements: (i) the knowledge of the binary code from which to extract the gadget, and (ii) their location in the Flash memory of the device to derive its address. In the case of the work mentioned above, the code is taken from a known and available library, so it is accessible to an attacker. Similarly, the location in Flash memory of that software is given by the specifications of the device itself. In embedded systems, Flash memory is always contained within a set of static addresses given by the specifications of its *memory map*, and is usually sized around 1 MB, corresponding to about 20 bits of addresses. Finding gadget locations does not necessarily require brute-force attempts on all possible addresses (2^{20}), but rather, as is well known, the compiler provides a certain *hierarchy* between the subsections of the code and the symbols contained in it. For a given embedded platform, there are very few different alternatives of usable compiler toolchain, and indeed, in many cases the choice is forced. Therefore, it is not difficult to derive a narrow range of possible layouts, on which to guess the address of the instructions of interest.

The above paper is an important work in refuting one of the most common counterarguments when it comes to control-flow attacks on embedded systems. Indeed, one might be misled into thinking that since embedded platforms possess a high degree of customization right by virtue of their special purpose, their software is predominantly uncommon and therefore difficult (if not impossible) to attack, in contrast to desktop and server systems, where the high presence of shared libraries grants a wide range of gadgets from which to draw material. This thesis is blatantly refuted by articles such as the one above, demonstrating real attacks on platforms that could be called custom. Additional trivial evidences are the existence of *Hardware Abstraction Layer* (HAL) libraries, present in all platforms of this kind, and representing a fixed codebase prodigal with potential gadgets. Another element are library functions (e.g., *libc*), statically compiled into most of embedded applications needing minimal functionalities, such as communicating characters over a standard interface (e.g., UART) or copying values to memory (e.g., `strcpy()`, `memcpy()`, `memset()`, etc.). Furthermore, such an argument is deriving from a concept of *security through*

obscurity, whereby the potential of the attacker would be limited by his inability to know the details of the platform he attacks. Such a concept is strongly rejected by this dissertation, which always considers valid the scenario in which the attacker finds, in one way or another, the details necessary for the operation of his attack, as argued in Section 1.4.

4.2 Issues and Challenges

Having ascertained that code redirection attacks to these platforms are possible and have a high degree of danger, it is now a matter of seeing how to combine the peculiarities of these systems with the need for the most comprehensive and efficient protection possible. Therefore, the discussion will now continue in identifying a set of points that, in our opinion, should be included in the study for an ideal control-flow integrity solution for embedded systems.

Granularity. The small computational capabilities of these systems could induce security designers to perform coarse checks on control-flow integrity. Relatively simple solutions, such as combining a stack protector and a restriction for indirect forward branches to function prologues via binary instrumentation, could prove to be a good compromise between security and efficiency. However, this type of solution has been strongly defeated in the literature due to the substantial lack of security [152]. Moreover, even from the efficiency point of view, instrumenting every function entry point could even be disadvantageous, since not all these functions could be the target of an indirect branch, just as not all indirect branches could be hijacked, as we will explain in the next Chapter. This thesis therefore considers as indispensable a *fine-grained* approach, that relies on a full static analysis in the offline phase to derive a CFG tracing the intended paths of the program, and on an online monitor able to ensure the consistency of each branch, forward or backward, on time.

Overhead & Schedulability. This is probably one of the biggest challenges when approaching an control-flow integrity solution. In terms of absolute limits, the ideal would be a solution that involves *no overhead* over the original execution environment. In reality, this is unattainable, and we have seen in the previous Chapter how each solution imposes some costs, in terms of execution time, code size, and added hardware required for protection. From the particular point of view

of embedded systems, this issue becomes even more relevant, as the additional cost on resources from the necessary minimum is a critical issue. For sure, this is an issue with regard to the overhead on execution time, since many embedded systems require *real-time* guarantees. To give an example, in advanced driver assistance systems (ADAS), components such as the anti-lock braking need to complete multiple control loops per second to keep the vehicle safe. If such a system misses a deadline, beyond the full correctness of its computation, the consequences could even result in loss of life. Whatever security mechanism one wishes to adopt, it cannot jeopardize this constraint, under any circumstances. In the presentation of solutions, the schedulability issue is often not discussed at all in the literature, since it is not a critical element for general-purpose systems. On executions lasting several tens of seconds, like those of the benchmarks generally used, having a few seconds of overhead may be tolerable.

The same limitations on execution time are echoed from a *resource* point of view. In fact, such systems have weaker computational capabilities than general-purpose systems such as desktops and servers. As already stated, if mainframe computing frequencies nowadays reach more than 4 GHz, the operational activity of processors in this domain hardly exceeds 200 MHz. The *power storage* potential for these devices is small, and in many cases, the power needed comes from batteries, which must last for a very long time, or it is harvested intermittently, e.g., from solar cells. Any added instructions for security checks are certainly impactful for these reasons. On the other hand, reserving specific execution slots or chip subareas to ensure security comes with a cost that very often can be unaffordable. In addition, because of the relative simplicity of the hardware designs of these systems, many mechanisms originally designed for general-purpose systems cannot be imported into this domain. E.g., all the techniques requiring advanced memory management features, reading of execution metadata (e.g., Last Branch Table), isolation via virtual memory, and the like, are not available for most microcontrollers. Similarly, features that may seem basic but in fact are not, such as a TrustZone, or even a Memory Protection Unit (MPU), cannot be taken for granted in these systems. Finally, the application of protection directly at the source level, such as Control-Flow Lock or CFLC, which in general-purpose systems reaches even 4x penalty peaks, can be absolutely unsustainable because of their overhead on execution: adding instrumentation at the source level for embedded systems can lead to distinctly longer control loops, for example.

On the other hand, the application of such techniques could give rise to a *security laziness* problem. In the solutions just mentioned, but also in solutions based on a control task such as TrackOS, the ability to defer CFI checks may seem a promising mechanism in environments where hard real-time constraints exist. However, the drawback of these approaches lies in the lack of expressiveness of their attacker model: in particular, it does not consider that an attack may succeed in the time frame that occurs between checks. On the other hand, preempting a task to execute a security check may be incompatible with their priorities with respect to the environment in which they run. Therefore, it is advisable to seek a solution that simultaneously respects the urgency of the checks and the schedulability of the tasks. For the purposes of this discussion, any solution that defers security checks at a later time falls into a coarse-grained view of CFI, that is not considered to be acceptable.

Parallelization. The impact of CFI solutions on execution time is theoretically solved by enabling *parallel execution*, in which one or more processing units are dedicated to perform security operations. However, the embedded landscape is largely populated by *single-core* devices, where a single control program is executed, or where the multiple parallel tasks compete for CPU usage with the support of an operating system that, for example, rotates the execution of the various tasks one after the other (*Round-Robin scheduler*), using a watchdog (e.g., timeout) to indicate the time when one task must preempt another. For example, this is the case for all microcontrollers with an ARM Cortex-M core. Applications requiring higher levels of computational capability may be based on dual-core processors, or much more rarely quad-core. For example, this is the area of ARM Cortex-A, the most extensive use of which is in the mobile and user-interactive IoT sectors. In the latter cases, however, the added computational capacity over the simpler microcontrollers is necessary for the very purposes of the applications, and cannot be reserved completely or even partially for security controls. An idea could be adding an *external module* that does not interfere with the operation of the device, ideally without necessarily involving the redesign of the device. External monitor solutions like BBB-CFI are tuned with these view, but they require a level of integration unattainable for legacy systems, since they need to directly read signals into the core. Instead, solutions based on external modules attached to the debug interface are less invasive.

From another point of view, one could exploit the mixed criticality nature of many multicore applications in this domain. For example, in the smart vehicle sector,

onboard system could be made of two fundamental parts. On the one hand, it must handle graphic functions for the user interface, allow interaction with the video cameras and the multimedia system, and so on. For this, it might need advanced libraries and services typically integrated into a general-purpose operating system. On the other hand, it must monitor and process in real time data coming from the sensors equipping the physical parts of the machine, activate the ABS, or perform other critical routines with precise periods of execution: for this, it might need to employ an real-time OS. Thus, a possible solution path could be about employing an operating system with laxer constraints to allocate tasks for monitoring the real-time execution domain. The entire system could be handled using a hypervisor, that provides the necessary protection guarantees for the additional memory parts needed to execute the integrity checks [163].

Portability. Very often, solutions for embedded systems try to import universal concepts of the CFI framework, such as the shadow stack or the labeling mechanism, trying to circumvent some of their limitations that may prevent their applicability in the specific. However, these workarounds overly adhere to this or that hardware platform used to present the work. An example is given by Silhouette, which is built on ARM Cortex-M and modifies the store instructions accordingly to use the MPU on this architecture. Since the limitations are hardware-specific, importing this solutions into other environments can sometimes be not trivial. Unlike desktop or server systems, where the dominant hardware platform (i.e., Intel x86) provides the greatest degree of cross-compatibility between systems, and the software background is also highly inter-compatible (e.g., Linux ABI), embedded systems use some of the most diverse architectures (e.g., ARM, RISC-V, MIPS, etc.), and this is compounded by a multiplicity of microcontroller variants with different features and *ad-hoc* customized software bases. For example, ARM offers the intellectual property of its cores under license, but this can be further adapted by customers. For these reasons, usually the proposed solutions hardly abstract from practical reality and offer a high-level model that can be adapted to different situations, although experiments can be conducted on this or that specific platforms. This in no way means considering only software-based solutions and excluding hardware-based ones only because, by definition, they are not portable. Given the variety of the embedded world, the objective should be to offer a solution model that can then be implemented through techniques ranging from pure software monitor to completely customized hardware for security.

Interrupt Awareness. Embedded systems often are immersed in an environment from which they must sample information and possibly actuate controls. This connote them as highly *event-based* computing platforms, i.e., to which some computation is asked in response to a signal coming from outside. For sake of completeness, embedded devices usually perform one or more tasks continuously inherent to their functionality (e.g., periodically collect data from sensor to process, send, store, etc.), but they are also continuously listening on the peripherals they are equipped with. For example, if on a digital line connected to the system, the value changes from negative to positive, it could mean that an external ancillary system is asking for its attention, perhaps to provide it with data, or perhaps to alert it to an abnormal situation to which a response is required. When this handling of external digital signals is incorporated into the computer architecture itself, we speak of *interrupt handling* mechanisms. An interrupt is an asynchronous signal from outside a processor, which usually has the power to pause normal execution to request an action. Each signal is associated with the execution of an *Interrupt Service Routine* (ISR), which may take a higher priority than any other execution. In such cases, the interrupt is said to be served immediately, with the highest priority. At the discretion of the system designer, there may be interrupts that are served with a lower priority (e.g., as soon as the system is in an idle condition relative to other priority tasks), or even some of these may be silenced if they are not deemed important enough to be considered or served. At the end of the execution of an ISR, execution picks up where it left off.

In the context of CFI, the nature of such sudden asynchronous and unpredictable events necessarily clashes with the static nature of the analysis that is done in the offline phase. When an interrupt is triggered, it is like a pair of *phantom edges* is created, outside the CFG. One connects the just-executed instruction to the initial basic block of the ISR, while another connects the exit point of the ISR to the instruction following the interruption site. At offline analysis time, it is not possible to know where these edges will be located. Moreover, their occurrence is not just untraceable, but also against the definition of CFG, as they can start from (and end to) an instruction internal to a basic block. Before serving an interrupt request, computer architectures usually save the context of the currently executed program in order to be able to restore it when the execution is resumed. The problem is that the code of the ISR is a just another piece of code, not immune to vulnerabilities from which an attack can start. In particular, if the routine contains one of the memory vulnerabilities presented above, the return address may be corrupted, and an attacker

may gain control over the program execution redirecting it to potentially dangerous code. Or, equivalently, the static analysis considerations on, e.g., register values that are later used to compute a branch target are made invalid. Given the intrinsic asynchronous nature of interrupts, no static analysis can provide a valid mean to monitor this type of transfers at runtime.

Let us try to explain the concept in practical terms. The Abadi's original label-based approach relies on modifying the compiled binary to insert unique IDs at the beginning of each basic block. Before each indirect branch, instructions are inserted to check if the destination basic block's ID is in fact targeted by the instruction. Anyway, attacks are still possible if an interrupt is served in the middle of the verification code.

The following code is used to ensure that the `jmp ecx` at the bottom reaches the code starting with that ID:

```
cmp [ecx], 12345678h
jne violation
lea ecx, [ecx+4]
jmp ecx
```

Anyway, let us suppose that an attacker has already corrupted the content of `ecx` after a memory vulnerability, with the aim to exploit that indirect branch. If an interrupt is served between the `cmp` and the `jne` triggering the violation, the *Processor Status Word* (PSW) is automatically pushed with the return address on top of the stack by x86 specification. The ISR may contain a vulnerability, and the PSW may be maliciously overwritten, such that the Zero Flag (ZF) is flipped and, when returning, the violation is bypassed and the desired piece of code is reached by the attacker.

The following instructions:

```
mov eax, 12345677h
inc eax
cmp [ecx+4], eax
jne violation
jmp ecx
```

represents an alternative way to instrument the code if the destination is instrumented as follows:

```
prefetchnta [12345678h]
mov eax, [esp+4]
...
```

i.e., using a side-effect-free x86 prefetch instruction. This version is even more exposed to interrupt issues: the previous exploitation is still possible if a vulnerable ISR is executed between the `cmp` and the `jne`, but another attack is possible. The `cmp` with the ID resorts to a register instead of an immediate. Therefore, let us suppose that a vulnerable ISR is triggered after the `mov` and before the `cmp`. If the ISR makes use of the `eax` register for its operations, it has to push it, and a stack corruption may permit to modify the content of `eax` with the desired label or even with the binary target of the attack. This is not unlikely: `eax` is a general-purpose register which may be used by the ISR.

Let us consider now the Control-Flow Locking approach. We can consider the following code snippet as the resulting instrumentation of a function epilogue:

```
L_lock: cmp lck, 0
        jne violation
        mov lck, key
        ret
```

targeting its intended return point, instrumented on its own as follows:

```
L_unlock : call <function>
           cmp lck, key
           jne violation
           mov lck, 0
```

Here, the lock code sets a lock variable to a value, while the unlock code verifies whether the value is the lock one before proceeding. The lock code also verifies if the just-executed code was unlocked and thus allowed to run, otherwise it notifies a violation. As in the label-based approach, problems may stem by the fact that the flags are possibly altered during the execution of a vulnerable ISR, so the violation can be skipped. In addition, when this solution is to be imported in a RISC machine, comparisons with memory locations are not allowed, and lock and key value are to be transferred from the secure storage into registers. In such a situation, if an interrupt arrives during the manipulation of the lock value, and the register used

is pushed because the ISR needs it, then it is possible that it may be restored as corrupted, with consequent defeat of the defense.

Hardware-based CFI solutions are not immune to possible breaks due to interrupts as well. Let us take the Sullivan's method as example (Figure 3.13):

```
fn_a :  cfichk A
        ...
        ...
        cfibr A1
        cfiprc B
        call *reg
        cfiret A1
        ...
        ...
        ...
fn_b :  cfichk B
        ...
        ...
        ret
```

Let us assume that an interrupt is triggered after `cfibr A1`. The ISR reached cannot obviously verify the caller identity with a `cfichk`, because this instruction accepts a static label, which is unknown at code writing time. The ISR may be vulnerable, and the return address may be tampered with. Again, the return site cannot be instrumented. The attacker can thus enter into the middle of any function body, and execute any wondered piece of code. At a certain time, when the `ret` is executed, the top of the LSS is written with `A1`, so the function returns back to the original call site, and no violation is sensed, as the `cfiret` performs a valid check.

Interrupt awareness is certainly an issue that must be considered when designing a control-flow integrity solution for embedded systems. In other words, although interrupts cannot be predicted, care must be taken to ensure that their execution makes no change to what is established by static CFG. This results in a general protection of the context that is tapped on the stack at the time the ISR is executed.

4.3 State-of-the-Art Discussion

In light of what said so far, it is interesting to specifically inspect some of the most notorious CFI solutions proposed in the literature explicitly targeting resource-constrained systems, and to discuss their characteristics with respect to the above identified issues.

RECFISH

As already presented, RECFISH [121] is a specific solution for real-time systems, which uses a mix of binary instrumentation and shadow stack, thanks to the support of the Memory Protection Unit (MPU) of ARM Cortex-M platforms. In order not to break the binary layout, the instrumentation is not applied directly to the `.text` section, but a special `.cfi` section is appended to the executable. Fixed code is placed in that section, acting as a destination for each critical point to be instrumented. A branch that then needs to be verified is transformed into a direct branch to this new section, where a protection code is executed, and then the flow is redirected back to the intended destination. Specifically (Figure 4.1):

- All function prologues are instrumented such that their link register is pushed onto the shadow stack; access to the shadow stack is via a supervisor call (`svc`) instruction to enter the protected area via MPU;
- All indirect branches go through a code that verifies their target;
- All function epilogues are instrumented to retrieve the link register from the shadow stack.

To make the solution work even in the presence of a real-time operating system, RECFISH proposes the extension of the *Task Control Block* (TCB) with an extra field for the shadow stack, just as the functions used to create and load a task also take care of the assignment of a region of the shadow stack and the switch between regions.

RECFISH is a fine-grained solution, which therefore guarantees punctual checks over CFG. However, from the point of view of the impact on execution time, the same authors report the results of a schedulability analysis done on a taskset taken

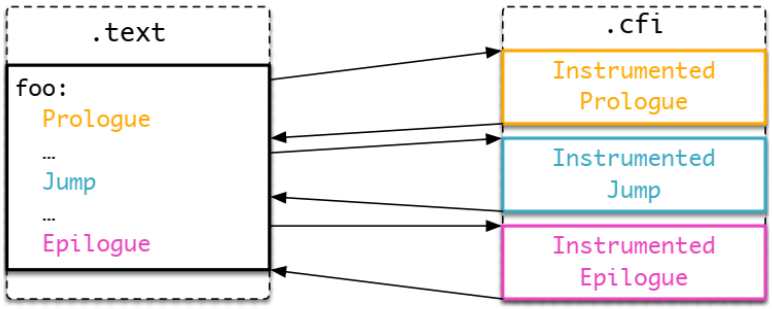


Figure 4.1 RECFISH's trampoline-based instrumentation [121].

from Brandenburg [164]. About 15% of these tasks resulted unschedulable after instrumentation. In addition, a performance loss of 30% is measured. RECFISH does not provide any degree of parallel execution, since execution from the normal code section moves to the instrumented section to perform security tasks. The solution is then specifically based for the reference architecture, and gives no indication of its possible generalization. In the paper, no mention is made whatsoever of the interrupt problem. An ISR usually executes with greater privileges than normal execution, i.e., the same privileges that are needed to set the MPU and access the shadow stack. A vulnerability within one of these could generate, for example, the loss of security information.

TrackOS

TrackOS [108] is a solution that adopts a deterministic CFI scheme within a real-time OS. Beyond the various operational tasks, a task is created and scheduled to act as a CFI monitor. Instead of interrupting execution to perform security operations, TrackOS schedules these at the time the monitoring task is scheduled (Figure 4.2). The latter owns the CFGs of all the various tasks, and is able to check the integrity of the stack against it and also of pointers to the code in general, adding checks based on heuristics (e.g., knowing the execution time of a function, it checks that its frame does not remain on the stack for more than a given time). Although TrackOS can be considered a perfect conjunction of fine checks and schedulability, its main problem remains the deferment of checks, before which the attacker may have already achieved his goal. In fact, as long as a task is running, no violation of the

control flow is detected. One possible solution might be instrumenting all monitored tasks to invoke the monitor at each indirect branch. However, this would break the whole point of TrackOS, which is designed precisely to prevent abnormal scheduling breaks. By the way, this would put the monitoring task at the level of all others, and if indirect branches were frequent, other tasks could be starving until the end of the checks. For these reasons, the granularity of TrackOS cannot be properly called fine, since it lacks the timeliness and the immediacy of the checks. From the point of view of execution penalties, no explicit data is reported, and only a 2 KB cost on the program memory of the system executable is mentioned. Experimental data are measured on ArduPilot [165], an open software for autopilot, of which no actual size in memory is reported, but only its length in lines of code, estimated to be around 10K. The level of parallelization of this solution is debatable depending on the point of view. One could say that the solution is highly parallel, because even in the presence of a single core, the checks do not interrupt execution. On the other hand, this is simply not the case, because the checks are deferred over time. The solution is originally designed for FreeRTOS, but the monitoring task could in principle be integrated into any RTOS with the appropriate adjustments. Finally, interrupt service routines are mentioned in relation to the task rotation mechanism, and stack integrity is ensured in such interstates. However, a general discussion of the problems that sudden interruptions during task execution might bring does not appear.

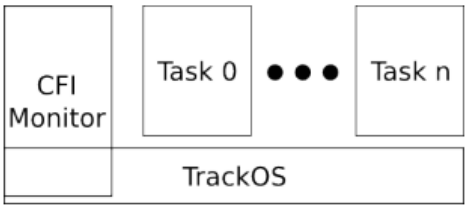


Figure 4.2 View of the tasks running on TrackOS. The monitoring task is represented as overlapping TrackOS to represent the privileged mode it runs in [108].

PATTER

The solution known in the literature as PATTER (Pointer AuThenTication for kERnels) [112] is instead designed to introduce CFI to the kernel of operating systems. PATTER relies on the pointer authentication extension introduced by ARM [113], and it is a protection applied by the compiler. To protect forward branches, all function pointers in memory are saved in the same address space, so that their offset can be used as a *modifier* for their signature and subsequent authentication, to prevent pointer substitution attacks. All pointers are signed before they are saved in memory, and authenticated each time they are loaded from memory. Indirect forward branches are executed via the `blraa` machine instruction, that simultaneously authenticates the pointer and jumps to the destination, to prevent TOCTOU attacks [116] that exploit a positive authentication result to use an inauthentic pointer. For backward branch protection, the same mechanism is used when the return address is stored in memory. After reloading, the `retaa` instruction authenticates the pointer and jumps back to the caller at once. The system kernel takes charge of initializing pointer authentication modules, setting registers for keys, and protecting them appropriately by using privilege levels, as well as authenticating function pointers that are statically initialized, before storing them in memory.

PATTER can be considered a fine-grained solution, since each indirect branch uses an authenticated pointer. As discussed earlier, pointer authentication makes a solution automatically fine-grained. Benchmarks from the UnixBench library [166] were used for performance evaluation, and an impact of between 10% and 20% was estimated. There is no degree of parallelization to discuss, since the security modules here are already present in the processor architecture and used during normal execution flow. PATTER's portability is conditioned on the use of ARM Pointer Authentication, and is therefore off limits for all platforms that do not support this facility or an equivalent one. Regarding interrupt issues, although the approach involves atomic instructions for authentication and branching, this does not make PATTER exempt from possible attacks whereby, for example, an interrupt intervenes between the loading of the pointer and its use in an authenticated branch. Without interrupt-specific protection, therefore, it cannot be excluded that the attacker, for example, exploits a vulnerability in an ISR to replace an authenticated pointer with another. Upon returning to normal flow, `blraa` and `retaa` will not detect any problems.

Camouflage

Camouflage [114] is a solution successive to PATTER, that uses the same principle of authenticating pointers via the appropriate ARM extension. Regarding forward branches, Camouflage distinguishes between pointers to functions present in a *static branch table* and those present within structures or classes. Since the former are based on a table of static values that are saved in program memory following the branch instruction, they run no risk of hijacking, unlike the latter, which are in fact authenticated. Protection of backward branches is improved over PATTER, as the signature is produced using a modifier that comes from a concatenation of stack pointer and function address. Camouflage uses a hypervisor to protect the keys used for authentication, storing them within a portion of memory that is marked as *Execute-Only Memory* (XOM) at the kernel level. The substantial difference with the previous solution, however, is the non-use of `blraa` and `retaa`, but the simple use of the `pac` and `aut` instructions for pointer signing and authentication, with the store or branch instruction occurring only at the next instruction.

As PATTER, Camouflage is a fine-grained solution, because it uses only authenticated pointers for its branches. Performance was evaluated using `lmbench` [167], and the results showed an overhead ranging from 10% to 30%. On parallelization, the same considerations made for PATTER apply, as well as portability. There is no mention of context protection in occurrence of interrupts. Splitting signature and authentication operations, as well as pointer transfer and utilization, create a TOCTOU problem similar to that present for PATTER, whereby the attacker exploiting a vulnerable ISR could even replace an authentic pointer with an inauthentic one. Upon returning to normal flow, a branch to a completely arbitrary location could be made.

Silhouette

Silhouette [122] is another solution outlined for ARM Cortex-M, that uses the MPU to support a shadow stack and also supports forward branch checks. Unlike the previous solutions, Silhouette is designed for bare-metal systems, i.e., embedded firmware that directly run on bare hardware does not use RTOS libraries. The code for a system equipped with Silhouette involves a distinction between a *privileged* codebase, including HAL functions and MPU configuration, and an *unprivileged*

codebase containing the normal application. The solution is implemented as modifications to LLVM's passes. One of these is the *Shadow Stack Transform*, which modifies each function prologue to save the return address on the shadow stack, and retrieve it at function epilogue. In addition, there is also a forward branch protection pass, which inserts label-based instrumentation such that each indirect call targets a function prologue. The protection of the shadow stack is done through the proper setting of the MPU and through a compilation step called *Store Hardening*, whereby all privileged store instructions are lowered to unprivileged instructions in such a way that the MPU controls every access. This ensures that the shadow stack is also protected from accesses in privileged code.

Silhouette is a coarse-grained solution, because forward branches can reach any function prologue. Its performance was evaluated using two well-known benchmark suites in the embedded domain, namely CoreMark [168] and BEEBS [169]. The runtime overhead was found to be 1.3% and 3.4% on average, respectively, for the two suites, a result that certainly makes it attractive in this domain. Even though the checks are very narrow in size, their degree of parallelization is zero, and in any case, checks are performed by the program itself. Without the support of the MPU, the solution cannot be ported other contexts. In the paper, there is no mention of interrupts, and even if the attacker has difficulty in modifying the secure information on the shadow stack because of Store Hardening, he still has the possibility of exploiting a vulnerable interrupt being triggered in the middle of the instrumentation code, to hijack a forward branch to an arbitrary destination.

Kage

Kage [123] is an extension of Silhouette for environments with a real-time operating system, again based on ARM Cortex-M architecture. Kage modifies the FreeRTOS [170] operating system by introducing a distinction between *trusted kernel* and *untrusted tasks*. The untrusted code undergoes the Store Hardening compiler pass introduced in Silhouette to transform all stores into unprivileged operations. On the other hand, the kernel code is considered trusted, and its instructions are kept privileged to make access to any memory locations. To access to trusted kernel facilities, the unprivileged code uses a secure API designed to vet arguments from untrusted code, so to avoid overwriting of control information. From the point of view of forward and backward branch controls, Kage does not differ from Silhouette.

However, it adds two very important rules, on context protection in cases of context switches and interrupts. In fact, in the context switch between RTOS tasks, Kage ensures that the task's saved processor state will always be the same as it was when the task was taken off of the CPU. Similarly, the processor state saved on interrupts and exceptions is enforced to be maintained across service routine execution. When an exception whose handler is untrusted occurs, an appropriate dispatcher executes first. The dispatcher function saves all processor state to the shadow stack and configures the MPU such that the entire task stack and task shadow stack regions are read-only (for both trusted and untrusted code). Only after this, the control is transferred to the untrusted exception handler. After the untrusted handler returns, the dispatcher restores the processor state and restores the MPU configuration.

Exactly as Silhouette, Kage is a coarse-grained solution, because all function prologues are considered as valid target points for a forward branch. The CoreMark suite was used to evaluate the performance of the solution, which is around 5.2%, slightly wider than the previous solution. On parallelization and portability, the same considerations made for Silhouette apply. On the contrary, it is important to note how Kage is, at the author's best knowledge, the only solution for resource-constrained devices that explicitly talks about context protection in the case of interrupts.

Table 4.1 summarizes the evaluations made so far on these six sample solutions drawn from the state of the art, reporting their key characteristics with respect to the issues identified in Section 4.2.

Solution	**Granularity**	**Overhead**	**Parallel**	**Portable**	**Interrupts**
RECFISH [121]	fine	30%	No	No	No
TrackOS [108]	coarse	n/a	Yes	Yes	No
PATTER [112]	fine	10-20%	No	No	No
Camouflage [114]	fine	10-30%	No	No	No
Silhouette [122]	coarse	1.3-3.4%	No	No	No
Kage [123]	coarse	5.2%	No	No	Yes

Table 4.1 Summary of analysed solutions and their coverage of the main issues.

Chapter 5

Fine-Grained, Real-Time, and Interrupt-Aware CFI: A New Formal Model

In light of what has been discussed so far in the present dissertation, this Chapter is aimed at outlining a new formal model for a control-flow integrity general philosophy for resource-constrained devices, and in particular, for the design of *vulnerability-tolerant architectures*. Thus, the goal is not to simply present a new solution, but to offer an adaptable framework to multiple architectures with different specifications, which is first based on a *formal theory* proving the validity of the offered model.

The framework offered in this thesis is named **PROLEPSIS**, a term derived from the Greek word πρόληψις (prólēpsis), which comes from the verb προλαμβάνω (prolambánō), which means "take beforehand, anticipate". In ancient times, the word had a meaning that today can be assimilated to that of *prevention*. In fact, the framework locates in the reference philosophy discussed in the introductory Chapter, such as to determine design patterns for computers that *anticipate* the dangerous condition in which they will certainly be (cfr. Zero-Trust Model, Section 1.4), and therefore know how to translate into architectures that are tolerant with respect to vulnerabilities, in this case present in the code executed on them, with the specific goal of blocking code redirection attacks.

First of all, it must be said that PROLEPSIS intends to fill the current gaps left by proposals in the literature and in the commercial domain for control-flow integrity

of embedded systems, with particular reference to the five issues presented in the previous Chapter. In other words, the proposal intends to be foundation for CFI solutions:

- **Fine-grained**, i.e., ensuring timely branch validity checks with respect to the intended CFG of a program;
- **Real-time**, i.e., having minimal impact on the average execution time of workloads to enable their proper schedulability and to preserve their determinism;
- **Interrupt-aware**, i.e., aware of the problems arising from not acknowledging asynchronous and sudden events such as interrupts as theoretically destructive elements with respect to models based on static analysis, thus needing due integration;
- Abstracted with respect to the implementation details of systems and **portable** to different situations, e.g., both within legacy architectures and in customizable processor designs;
- Performing security checks in **parallel** with respect to the main execution, to ensure greater protection and efficiency at the same time;
- Aimed to generate architectures that **resist** software vulnerabilities regardless of the software features, and thus, enabling from the *hardware* layer the security of the upper layers (cfr. *Hardware-based Security*, Section 1.3);
- **Source-free**, i.e., applicable even in the absence of a program's source files, so to protect legacy applications that, for example, are proprietary, or for which the source has been lost, or for which support is expired (cfr. Figure 3.12 in Section 3.4);
- With a **dynamic policy**, i.e., capable of detecting from the running program as much information as possible to identify the expected target of a branch on time (cfr. Figure 3.12 in Section 3.4);
- With **static instrumentation**, i.e., producing a possibly augmented binary code from the original, with the minimal instructions needed to interact with the security modules (cfr. Figure 3.12 in Section 3.4)[1].

[1]The use of dynamic instrumentation could be limiting in the many cases where, in the reference domain, there is no support for parallel executions and/or supervisor/hypervisor.

Being the embedded domain taken as a reference by PROLEPSIS, the framework bases its theory on some fundamental assumptions, some of which are already assumed by all state-of-the-art solutions. These are listed in the following.

UNC *The software running on embedded systems, also referred to as firmware, is a unique, discrete, and fully distinguishable executable entity. All instructions eventually executed are already present in the program memory of the device ahead of runtime. Dynamic code linking is considered as out of scope.*

NWC *None of the memory sections that contain executable instructions of the firmware can be modified during the system's activity phase* (cfr. [95]).

NXD *None of the memory sections that contain data can be used to fetch instructions for execution* (cfr. [95]).

UNQ *All identifiers used to label locations within the code are unique and cannot have replication within the firmware* (cfr. [95]).

SCR *The processors with which the systems are equipped are single-core, with no hardware support for parallel execution of processes. Parallelism is possibly emulated by contention of the only execution unit present on the system.*

INT *The firmware is composed of a single main program that executes the special-purpose task for which the system was created, possibly divided into several subtasks that are, however, never executed in parallel; to this, a given number of interrupt service routines is added, with a routine provided for each interrupt supported.*

STA *The location (address) of each object in memory is determined at compile time, i.e., the reference compilation model is the common one for embedded systems, with static code and absolute references to symbols within the code, determined by a linker script edited according the system's memory map and without the possibility of code relocation.*

Moreover, PROLEPSIS considers control-flow transfers as those possibly generated by the compilation of a program usually written in C, i.e., types CF.1, CF.2, CF.3, CF.4, CF.5, and CF.8 (cfr. Section 3.4, [97]). In terms of Static Analysis Precision (SAP), intended levels of the framework are SAP.F.1c for forward branches and SAP.B.2 for backward branches (cfr. Section 3.4, [97]).

5.1 ToyRISC Language

In this Section, our simplified RISC-style Instruction Set Architecture (ISA), called ToyRISC, is presented. The main purpose of ToyRISC is to abstract away all the aspects of the computation that are irrelevant with respect to control-flow integrity. According to the author, the present formalism is an extension of that presented by Abadi in his work [40], which was itself taken from Hamid [96]. In particular, it aims to describe in a formal way the behavior of an application in the reference domain, i.e., the embedded one, and not in a totally general way.

5.1.1 Syntax

$inst$ ::=	$\texttt{B}\ lab \mid \texttt{BR}\ r \mid \texttt{BL}\ lab \mid \texttt{BLR}\ r \mid \texttt{CBZ}\ r, lab \mid \texttt{RET}$	(branch)
	$\mid \quad \texttt{PUSH}\ r \mid \texttt{POP}\ r \mid \texttt{PUSH}_{lr} \mid \texttt{POP}_{pc} \mid \texttt{INC}\ v \mid \texttt{DEC}\ v$	(stack)
	$\mid \quad \texttt{MOV}\ r_1, r_2 \mid \texttt{MOVI}\ r_1, v \mid \texttt{AOP}_{\boxplus}\ r_1, r_2, r_3$	(data)
	$\mid \quad \texttt{LDR}\ r_1, [r_2] \mid \texttt{STR}\ [r_1], r_2$	(memory)
$prog$ ::=	$lab_1 : inst_1, \ldots, lab_n : inst_n$	

Table 5.1 ToyRISC Assembly language syntax.

The syntax of ToyRISC follows is given through the BNF grammar of Table 5.1. Briefly, a program *prog* is a finite ordered sequence of labeled instructions ($lab : inst$). Intuitively, labels are assigned to indicate location (addresses) of each instruction of the program. We use $Addr_{prog}$ to denote the ordered set of all labels of a program *prog*, being init($prog$) its initial, entry address. Being $Addr_{prog}$ an ordered set, the sum ($+$) and the difference ($-$) operation with natural values are meaningful, and the operation produces labels that that precede or follow that label by a certain number of positions. Also, when labels are immaterial, we feel free to omit them.

ToyRISC acknowledges the presence of n general-purpose registers, named as r_1 to r_n and collected in the set R. These are accessible to data manipulation instructions, e.g., `MOV`. Furthermore, the Instruction Set Architecture (ISA) of ToyRISC assumes the presence of three special-purpose registers, pc, sp and lr. The *program counter* pc stores the address of the current instruction. The *stack pointer* sp contains the current height of the program stack (see below). Finally, the *link register* lr contains the return address to be returned to when a function ends.

Program instructions belong to four categories. Branching instructions can be a simple branch to a label (`B` *lab*), a simple branch on a register value indicating the target address (`BR` *r*), a branch with call-back link (aka "branch & link", `BL` *lab*), a "branch & link" on a register (`BLR` *r*), a conditional branch that is taken if the operand register value is zero (`CBZ` *r*, *lab*), or a return to linked caller (`RET`, semantically equivalent to `BX` *lr*).

As usual, stack instructions amount to pushing/popping to/from the stack from/to a register (`PUSH` *r*, `POP` *r*). Dedicated instructions are included to push the value of the link register ($\texttt{PUSH}_{lr}$), and pop the value of the program counter ($\texttt{POP}_{pc}$). These instructions are mainly devoted to handle nested routines. Moreover, instructions `INC` and `DEC` allow to allocate and de-allocate (respectively) a constant number *v* of stack positions by modifying the stack pointer.

Memory instructions are `LDR` and `STR`. The former has destination operand as register and source operand as memory (pointed by a register value). The opposite happens for the latter, of course.

Finally, `MOV` is used for register-to-register data copy, while `MOVI` is for immediate-to-register, i.e., copying an immediate value *v* inside the destination register. $\texttt{AOP}_{\boxplus}$ denotes a generic, 3-registers arithmetic/logic operation indicated with the operand $\boxplus$, where the destination register (the first one) is written with the result of the combination of the two sources (second and third register).

Without reducing the expressive power of ToyRISC, we assume that all programs are *well-formed*.[2] Well-formed programs rely on instructions `BL` and `BLR` for invoke code sections that, eventually, return the control to the call site by using either `RET` or $\texttt{POP}_{pc}$. For instance, this happens when function calls in a high-level language are compiled. For better highlighting this behavior, we call *forward* and *backward* branch instructions $\{\texttt{BL}, \texttt{BLR}\}$ and $\{\texttt{RET}, \texttt{POP}_{pc}\}$, respectively. Moreover, and with no loss of generality, we assume that each forward branch instruction is associated with a unique backward one. In symbols, given a forward branch instruction $l : inst$, we denote the label of its unique, associated backward branch instruction with $\text{assoc}(l)$.

[2] E.g., we can think of well-formed programs as those generated by a trusted compiler.

5.1.2 Program Semantics

Definition 1 (Program Memory)**.** *A **program memory** M is a finite mapping between locations loc and elements v. We write $[]$ for the empty memory and $[v/loc]M$ for the memory where loc is (re)allocated to v. Access to location loc is defined as:*

$$M(loc) = \begin{cases} v & \text{if } M = [v/loc]M' \text{ (for some } M') \\ M'(loc) & \text{if } M = [v/loc']M' \text{ (for some } M' \text{ and } loc' \neq loc) \end{cases}$$

Definition 2 (Program Stack)**.** *The **program stack** is a dynamically managed subsection of the program memory. In addition to random access, stack data allocation and de-allocation is driven by a special register called* stack pointer*, denoted with sp. Intuitively, sp contains the free location on top of the current stack.*

Definition 3 (Configuration)**.** *A **configuration** is a tuple $\langle prog, M, R, pc, sp, lr \rangle$ where $prog$ is a ToyRISC program following the syntax of Table 5.1, M is the program memory, $R = r_1, \ldots, r_n$ are the general-purpose registers, pc is the program counter register, sp is the stack pointer and lr is the link register.*

We use $R[r]$ and $R[r] \leftarrow v$ to denote reading and writing (respectively) a general register r. Slightly abusing the notation, we may use, e.g., $R[sp]$ for accessing a special register. Nevertheless, we assume that pc, sp, and lr can only be accessed in read mode by general-purpose instructions.

The initial configuration of a program $prog$ has the form

$$\langle prog, M, [0, \ldots, 0], \text{init}(prog), 0, 0 \rangle$$

where $\text{init}(prog)$ is the entry point of the program and M is any arbitrary memory allocation (e.g., storing program's inputs).

Definition 4 (Small step semantics)**.** *The small step semantics of ToyRISC is defined through the rules of Table 5.2.*

$$\frac{prog[pc] = \texttt{B}\,l}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, l, sp, lr\rangle} \text{ (Branch)}$$

$$\frac{prog[pc] = \texttt{BR}\,r \qquad R[r] = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, v, sp, lr\rangle} \text{ (BranchR)}$$

$$\frac{prog[pc] = \texttt{BL}\,l}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, l, sp, pc+1\rangle} \text{ (Branch\&Link)}$$

$$\frac{prog[pc] = \texttt{BLR}\,r \qquad R[r] = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, v, sp, pc+1\rangle} \text{ (Branch\&LinkR)}$$

$$\frac{prog[pc] = \texttt{CBZ}\ r,\ l \qquad R[r] = 0}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, l, sp, lr\rangle} \text{ (Conditional0)}$$

$$\frac{prog[pc] = \texttt{CBZ}\ r,\ l \qquad R[r] = v \qquad v \neq 0}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, pc+1, sp, lr\rangle} \text{ (Conditional1)}$$

$$\frac{prog[pc] = \texttt{RET}}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, lr, sp, lr\rangle} \text{ (Return)}$$

$$\frac{prog[pc] = \texttt{PUSH}\,r \qquad R[r] = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, [v/sp]M, R, pc+1, sp+1, lr\rangle} \text{ (Push)}$$

$$\frac{prog[pc] = \texttt{POP}\,r \qquad M(sp-1) = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R[r \leftarrow v], pc+1, sp-1, lr\rangle} \text{ (Pop)}$$

$$\frac{prog[pc] = \texttt{PUSH}_{lr}}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, [lr/sp]M, R, pc+1, sp+1, lr\rangle} \text{ (PushLr)}$$

$$\frac{prog[pc] = \texttt{POP}_{pc} \qquad M(sp-1) = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, v, sp-1, lr\rangle} \text{ (PopPc)}$$

$$\frac{prog[pc] = \texttt{INC}\,v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, pc+1, sp+v, lr\rangle} \text{ (Increment)}$$

$$\frac{prog[pc] = \texttt{DEC}\,v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R, pc+1, sp-v, lr\rangle} \text{ (Decrement)}$$

$$\frac{prog[pc] = \texttt{AOP}_{\boxplus}\,r_1\,r_2\,r_3 \qquad R[r_2] = v \qquad R[r_3] = v' \qquad v \boxplus v' = v''}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R[r_1 \leftarrow v''], pc+1, sp, lr\rangle} \text{ (AOp)}$$

$$\frac{prog[pc] = \texttt{MOV}\ r_1,\ r_2 \qquad R[r_2] = v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R[r_1 \leftarrow v], pc+1, sp, lr\rangle} \text{ (MoveRegister)}$$

$$\frac{prog[pc] = \texttt{MOVI}\ r_1,\ v}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R[r_1 \leftarrow v], pc+1, sp, lr\rangle} \text{ (MoveImmediate)}$$

$$\frac{prog[pc] = \texttt{LDR}\ r_1,\ [r_2] \qquad R[r_2] = v \qquad M(v) = v'}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, M, R[r_1 \leftarrow v'], pc+1, sp, lr\rangle} \text{ (Load)}$$

$$\frac{prog[pc] = \texttt{STR}\ [r_1],\ r_2 \qquad R[r_1] = v \qquad R[r_2] = v'}{\langle prog, M, R, pc, sp, lr\rangle \to \langle prog, [v'/v]M, R, pc+1, sp, lr\rangle} \text{ (Store)}$$

Table 5.2 Small step semantics of ToyRISC.

Informally, (Branch) states that `Bl` instructions make the execution jump to constant address l by storing it inside pc, while (BranchR) defines how `BRr` jumps to the address contained in r. Symmetrically, (Branch&Link) and (Branch&LinkR) jump to l and $R[r]$ (respectively), but they also store $pc+1$ inside the link register lr. Rules (Conditional0) and (Conditional1) define the behavior of `CBZrl` which jumps to either l or $pc+1$ depending on whether r contains 0 or not. Rule (Return) replace the program counter pc with the content of the link register lr. Rules (Push) and (Pop) add/remove (respectively) the top element of the stack. When pushing, the top element is taken from register r and the stack pointer is increased by one. Vice versa, popping results in the top element being stored in r and a decrement of the stack pointer. A similar behavior is defined by rules (PushLr) and (PopPc), with the difference that, the former pushes lr and the latter pops into pc. Rules (Increment) and (Decrement) allocate and deallocate (respectively) v positions on the stack by increasing/decreasing the stack pointer. Three-operands arithmetic operations are executed through rule (AOp), while register values and constants are copied into a register through rules (MoveRegister) and (MoveImmediate), respectively. Finally, (Load) accesses the memory at location $R[r_2]$ and copies it into register r_1, while (Store) takes the content of r_2 and writes it in memory at location $R[r_1]$.

5.1.3 Interrupt Semantics

The semantics of Definition 4 describes the linear execution of a program. Nevertheless, as already introduced, most real architectures permit to define procedures that are executed when certain events occur. Typically, this is done by introducing the notion of *interrupt*, i.e., sudden events that trigger an handling routine. Below, we introduce a interrupt semantics for ToyRISC.

Definition 5 (Interrupt Table)**.** *An **interrupt table** I is a finite mapping between interrupt signals $\{i_1,\ldots,i_k\}$ and interrupt handlers $\{prog_1,\ldots,prog_k\}$. When an interrupt signal i is fired, we denote it with $\lightning i$.*

Definition 6 (Interrupt Semantics)**.** *Given an interrupt table I, the interrupt handling rule is defined below.*

$$\frac{\lightning i \quad I(i) = prog_i \quad \text{disable}(\lightning)}{\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog_i, [pc/sp]M, R, init(prog_i), sp{+}1, lr\rangle}\ \text{(Int)}$$

Furthermore, we assume that every interrupt handler $prog_i$ has a single exit point that consists of a custom instruction, `IRET`.

$$\frac{prog_i[pc] = \texttt{IRET} \qquad M(sp-1) = v \qquad \text{enable}(\lightning)}{\langle prog_i, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, v, sp-1, lr\rangle} \text{ (RetInt)}$$

The interrupt semantics $\twoheadrightarrow$ of ToyRISC is obtained from the rules of Table 5.2 by adding rules (Int) and (RetInt).

With no loss of generality, we assume that interrupt handlers cannot be nested, i.e., they cannot be interrupted. This means that, if an interrupt is fired during the execution of a handler, the new interrupt's handler is postponed after the end of the current one. To achieve this, we introduce two support functions, called enable($\lightning$) and disable($\lightning$), for enabling and disabling (respectively) the interrupt signaling mechanism.[3] When the interrupts are disabled, $\lightning\, i$ cannot occur (for any possible i) and, thus, rule (Int) cannot be invoked.

Definition 7. *We say that interrupt handler $prog_i$ is* well-formed *if, upon termination, it restore the original values contained in R before the occurrence of the interrupt.*

Intuitively, Definition 7 requires that interrupt handling does not disrupt the execution context, i.e., R. Often, this is ensured by the compilation process that introduces instructions for registers' backup and retrieval, e.g., by using the program stack. From now on, we assume all the interrupt handler to be well-formed.

5.2 CFI Definition

In this Section, we provide our definition of control-flow integrity for ToyRISC programs. It is first necessary to define the threat model. For the purposes discussed here, the attacker is a malicious actor interested in hijacking the control flow of the program to an arbitrary execution of his choice. To do this, he needs to corrupt that data in memory that will later be used as pointers to the code by indirect branch instructions (see below). So, it is reasonable to model the attacker as being able to intervene in the memory write instructions, and change argument data at will.

[3] When disable($\lightning$) is called multiple times, the same number of enable($\lightning$) must occur for reactivating interrupts.

Definition 8 (Adversarial Rules)**.** *The adversarial behavior of ToyRISC is modeled by the following transition rules.*

$$\frac{prog[pc] = \texttt{PUSH}\, r}{\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow_A \langle prog, [v/sp]M, R, pc{+}1, sp{+}1, lr\rangle} \textit{ (PushAtk)}$$

$$\frac{prog[pc] = \texttt{PUSH}_{lr}}{\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow_A \langle prog, [v/sp]M, R, pc{+}1, sp+1, lr\rangle} \textit{ (PushLrAtk)}$$

$$\frac{prog[pc] = \texttt{STR}\ [r_1],\ r_2 \qquad R[r_1] = v'}{\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow_A \langle prog, [v/v']M, R, pc+1, sp, lr\rangle} \textit{ (StoreAtk)}$$

for arbitrary v.

Briefly, the rules of Definition 8 allow the attacker to freely replace any value written by operations that modify the memory during the execution. The attacker semantics is obtained from the interrupt semantics by adding the rules of Definition 8. The attacker's goal is to force an *illegal run*.

Definition 9 (Run)**.** *A **run** is defined as the transitive closure of the union between transition relations $\twoheadrightarrow$ and $\twoheadrightarrow_A$. We denote a program run with $\twoheadrightarrow_A^*$ and we use $\twoheadrightarrow^*$ for program runs where rules of Definition 8 never occur.*

Definition 10 (Control Flow)**.** *Given a program prog, we define $L_{prog} : Addr_{prog} \to \mathscr{P}(Addr_{prog})$ as a finite mapping between a source and a finite set of destination labels. Intuitively, $L_{prog}(l) = \{l_1, \ldots, l_n\}$ means that the execution of $prog[l]$ must result in a configuration such that $pc \in \{l_1, \ldots, l_n\}$.*

Moreover, we say that L_{prog} is sound *if and only if for any run $C \twoheadrightarrow^* \langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M', R', pc', sp', lr'\rangle$ (where C is an initial configuration for prog), it holds that $pc' \in L_{prog}(pc)$. From now on, we always assume L_{prog} to be sound.*

Intuitively, L_{prog} models the abstract notion of static *control-flow graph* (CFG). Since actual CFGs can be built with different algorithms, e.g., each with its own approximations and peculiar features, our definition can be applied in general. As a matter of fact, the property stated in Definition 10 must hold on any construction algorithm for *sound* control-flow graphs, i.e., graphs that reflect the actual runtime behavior of the program.

Definition 11 (Illegal Branch and Illegal Run)**.** *Given a configuration* $\langle prog, M, R, pc, sp, lr\rangle$ *and a mapping* L_{prog}*, we say that* $\langle prog, M', R', pc', sp', lr'\rangle$ *is an* ***illegal branch*** *if either*

1. $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M', R', pc', sp', lr'\rangle$ *with* $pc' \notin L_{prog}(pc)$*, or;*
2. $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow C_i \twoheadrightarrow^* C_i' \twoheadrightarrow \langle prog, M', R', pc', sp', lr'\rangle$ *where* $pc' \neq pc$ *and* $C_i \twoheadrightarrow^* C_i'$ *is a run of interrupt handler* $prog_i$ *s.t. in* C_i' *the program counter is* pc_i *and* $prog_i[pc_i] =$ `IRET`*.*

We define an ***illegal run*** *any run of program prog that includes at least one illegal branch.*

Example 1. *Consider the following ToyRISC program.*

```
l1 : PUSH lr
l2 : INC  10
l3 : MOVI r1 , 1
l4 : MOV  r2 , sp
l5 : AOP_ r2 , r2 , r1
l6 : BL   get_a_char
l7 : STR  [r2] , r0
l8 : CBZ  r0 , la
l9 : B    l5
la : DEC  10
lb : POP pc
```

The code above provides a compact example of stack corruption through buffer overflow, which is an example of how in the real world the attacker succeeds in overflowing a buffer and in corrupting data that are successively used to jump to a custom address. Let us suppose that r2 is used as index to progressively fill data from stack pointer, and that there exists a function, `get_a_char`*, that simply stores in register r0 the code of a character inserted through the standard input.*

From the code, it is clear that a 10-character buffer is allocated onto the stack, and if there is no NULL character (= 0) inserted in the first 10 characters taken, the buffer boundary is overrun and the return address (pushed onto the stack at the beginning) is corrupted. Instruction at l_b *make the attacker jump to any location.*

We now state two formal properties of our attacker semantics.

Lemma 1. *Every arbitrary long illegal run is of the form* $C \twoheadrightarrow_A^* C_0 \twoheadrightarrow C_1 \twoheadrightarrow_A^* C'$ *where* (i) $C \twoheadrightarrow_A^* C_0$ *is a legal run and* (ii) $C_0 \twoheadrightarrow C_1$ *is an illegal branch.*

Briefly, Lemma 1 states that every illegal run must have a minimal illegal prefix, i.e., a finite number of computational steps ending with an illegal branch. Thus, following [171], illegal runs can be effectively detected by a safety monitor.

Another interesting property is the following.

Lemma 2. *If* $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow C$ *is an illegal branch w.r.t.* L_{prog}*, then* $prog[pc] \in \{\texttt{BR}, \texttt{BLR}, \texttt{IRET}, \texttt{POP}_{pc}\}$.

5.3 CFI Enforcement

PROLEPSIS consists of three core elements, i.e., (i) a vulnerable branch detection strategy, (ii) a reference monitor, able to punctually and timely enforce vulnerable branches not to originate illegal runs, and (iii) a program instrumentation procedure, to possibly insert the necessary interaction steps between the program and the monitor. Briefly, the PROLEPSIS reference monitor implements a classic Policy Decision Point (PDP) architecture [172], where Policy Enforcement Points (PEP) are instrumented whenever an illegal branch may occur, and Policy Information Point (PIP) are instrumented at specific points containing useful information for later decisions of the PDP. Below, we detail the three core elements of the framework.

5.3.1 Vulnerable Branch Detection

According to our attacker model, the adversary can only compromise branch instructions by tampering with the program memory during writing operations. Although he has no direct control on the program registers, the attacker can inject them when, potentially malicious, data are loaded. Thus, the attacker can hijack the control flow by means of a *tainted flow*, as defined below.

Definition 12 (Tainted Branch)**.** *Given program prog and label l, we say that r is a* tainted register *(at l) if at least one of the following conditions hold.*

1. $prog[l] = \texttt{LDR}\, r, [r']$ *(for any* r'*).*

2. $prog[l] = \texttt{POP}\, r$.

3. $prog[l] = \texttt{MOV}\, r, r'$ *and* (i) l *is a* tainted label *or* (ii) $\exists l'$ *s.t.* $l \in L_{prog}(l')$ *and* r' *is tainted at* l'.

4. $prog[l] = \texttt{AOP}_{\boxplus}\, r, r', r''$ *and* (i) l *is a* tainted label *or* (ii) $\exists l'$ *s.t.* $l \in L_{prog}(l')$ *and either* r' *or* r'' *is tainted at* l'.

Furthermore we say that l *is a* tainted label *if there exists* l' *such that* $l \in L_{prog}(l')$ *and*

1. l' *is a tainted label, or;*

2. $prog[l'] = \texttt{CBZ}\, r, l''$ *with* r *being a* tainted register.

Finally, we say that $prog[l]$ *is a* tainted branch *when either* (i) $prog[l] = \texttt{BR}\, r$, (ii) $prog[l] = \texttt{BLR}\, r$ *(both with* r *tainted at* l*) or* (iii) $prog[l] = \texttt{POP}_{pc}$.[4] *The set of tainted branches of a program* $prog$ *is denoted by* $\mathrm{TB}(prog)$.

Example 2. *Consider the following ToyRISC program.*

```
l0: MOVI r8, 0x24000000
l1: MOVI r11, 0x5A00
l2: AOP+ r5, r8, r11
l3: MOVI r2, 0x8007AA5
l4: STR [r5], r2
l5: MOVI r3, 0x24005A00
l6: LDR r4, [r3]
l7: BR r4
```

Example 3. *Consider the following ToyRISC program.*

```
l0: MOVI r8, 0x8000000
l1: MOVI r11, 0x200
l2: MOVI r4, 0x40
l3: AOP+ r5, r8, r11
l4: AOP× r3, r4, r5
l5: BR r3
```

[4] Notice that in case (iii), pc is tainted and used as branch operand in a single instruction.

In the case of Example 2, the tainted flow occurs between l_6 and l_7, as programmer expect to have `0x8007AA5` in r_4 at l_7, but actually the result may be different if the attacker has replaced M with M' at his will. Instead, Example 3 has no tainted flow: even if l_5 contains a `BR` instruction, the value contained in r_3 at l_5 cannot be influenced by the attacker.

Example 4. *Consider the following ToyRISC program.*

```
l0 : MOVI r0, 0
l1 : MOVI r1, 0x8004200
l2 : MOVI r2, 0x21002000
l3 : STR r1, [r2]
...
l4 : LDR r3, [r2]
l5 : CBZ r3, la
l6 : MOVI r4, 1
l7 : AOP+ r0, r0, r4
l8 : AOP- r3, r3, r4
l9 : B l5
la : BR r0
```

A flow can be tainted even if not in an explicit way. In the strange case of Example 4, register r_0 is inizialized with zero, and then incremented by 1 in each step of a loop. The loop ends when r_3, containing the value `0x8004200` that has been loaded from memory at `0x21002000`, reaches 0. While r_3 is decremented at each pass, r_0 is incremented. At the end of the loop, r_0 contains `0x8004200` and in l_b, this instruction address is reached through a branch. A program analyzer could consider that `BR` r_0 is a non-tainted branch, as r_0 is a non-tainted register. However, the value `0x8004200` has been stored in memory at `0x21002000` before reloading, and this makes r_3 tainted. Since r_3 decides on the value of r_0 through the loop, r_0 is a tainted register too, and, thus, the branch using it is tainted.

There exist several alternative techniques for the identification of tainted flows. Since we use tainted flows for driving the program instrumentation, we must resort to a static approach. In particular, we opted for *static program slicing* [173], as the one for example used by `angr` software toolkit[5]. Briefly, given a root instruction and a set of leaf instructions, a program slicing algorithm extracts a *slice* of the program

[5] https://docs.angr.io/built-in-analyses/backward_slice

code which denotes all the possible program executions where data may flow from (a subset of) the leaves to the root. A slice is typically generated through a *backward visit* that, starting from the root instruction, searches for all the leaves that can be reached by going up through the program instructions.

5.3.2 Reference Monitor

We start by providing a formal definition of our reference monitor.

Definition 13. *As usual, a* stack *S is a finite data structure such that:*

- $\perp$ *is the empty stack;*
- $\text{push}(x,S)$ *is the stack obtained by allocating x top of S;*
- $\text{pop}(S)$ *is the stack where the top element of S has been removed, and;*
- $\text{top}(S)$ *is the element allocated at the top position of S.*

Definition 14. *A* reference monitor *configuration $\Omega = \langle S_\omega, R_\omega, src, t \rangle$ is a four-tuple where:*

- *S_ω is a stack which serves as a* shadow *call stack of the program;*
- *R_ω is the* shadow *register set used to back up R during interrupt handlers;*
- *src is a variable containing the last branch source label, and;*
- *t is the step counter, i.e., the number of computational transitions occurring during a program run (see Definition 6).*

Based on the step counter of Definition 14, we introduce a *timeout* operator. Briefly, the timeout is set at step t through timeout $\leftarrow t$ and it is deactivated by timeout $\leftarrow \infty$. Moreover, re-assigning the timeout when it is already set has no effect. Finally, the timeout is tested by means of $t \leq$ timeout (for some value of t).

With respect to a classical PDP infrastructure, our reference monitor implements the inner logic of the policy controller. Following the classical inlined reference monitor approach [174], we also assume PIP and PEP instructions to be deployed properly deployed within the program code (see Section 5.3.3). To this aim, we

introduce two new opcodes, called `PEP` and `PIP`, within the ISA of ToyRisc. These instructions follow the syntax below.

$$\texttt{PEP}\, val, policy \qquad\qquad \texttt{PIP}\, val, policy$$

Their parameters represent (i) a value *val*, e.g., a datum or a label, to be transmitted to the PDP, and (ii) a *policy* code, representing the meaning of that value for the policy to be verified. In fact, each PEP surrounds a tainted branch (see above) and, during the execution, it is responsible for querying the PDP about whether the ongoing operation preserves the program CFI. Each PIP brings instead additional information to be used by the PDP for later decisions.

Specifically, to achieve a complete fine-grained CFI, it is necessary to enforce the L map. Given the above premises, these are the two resulting policies to be enforced:

1. **Forward Branch Policy (FBP)**: *Each forward branch shall reach only one of the allowed destinations according to the static analysis mentioned in 5.3.1;*
2. **Backward Branch Policy (BBP)**: *Each backward branch shall reach only the return site of its last caller, included among the allowed destinations of that branch according to the static analysis mentioned in 5.3.1.*

Since there is no risk from branches that are not tainted, enforcement to comply with the policy should only be done around tainted branches. Enforcement is achieved by checking the actual destination of the branch (given by its argument at runtime) against the pre-calculated destination by the static analysis (i.e., given by the L_{prog} map), which the PDP is aware of.

As a matter of fact, the policy derived from the static analysis can only determine whether the branch points to one of the (possible) multiple destinations. Therefore, tainted backward branches need a different level of instrumentation, based on information that is passed to the policy verifier at runtime.

In particular, in order for the PDP to be informed about the identity of the last caller, instrumentation must also be performed on the forward branches associated with backward branches, regardless of whether these forward branches are vulnerable or not. If they are not, the PDP must simply record their identity; if they are, the PDP must also verify that the forward branch itself is pointing to a valid destination. The

information relates to the identity of the caller, and is used at the time of function return to enforce the branch to the caller site.

At this point, policy enforcing could be architected as follows:

- before the execution of a tainted forward branch, a PEP could be inserted to query the PDP about the L map related to that branch, i.e., to validate the branch instruction argument and make sure that it is included in the set of legitimate destinations;
- before the execution of a tainted backward branch, a PEP could be inserted to query the PDP about the identity of the last caller, appropriately communicated to it by a previous PIP at call time; the PDP should verify its identity against the current argument of the return branch.

Forward Branch Policy (FBP) enforcement occurs as showed in Figure 5.1. Right before a register-based branch instruction (i.e., `BR` or `BLR`) is executed, a PIP instruction is triggered to alert the PDP (1). The PDP stores the branch address l[6] inside variable *src*, and sets a timeout allowing the execution of only one ToyRISC instruction (2). Then, the program executes the branch instruction and reaches location l' (3), where a PEP instruction triggers the PDP (4) to check whether l' is a legal destination for the branch located at *src* (5).

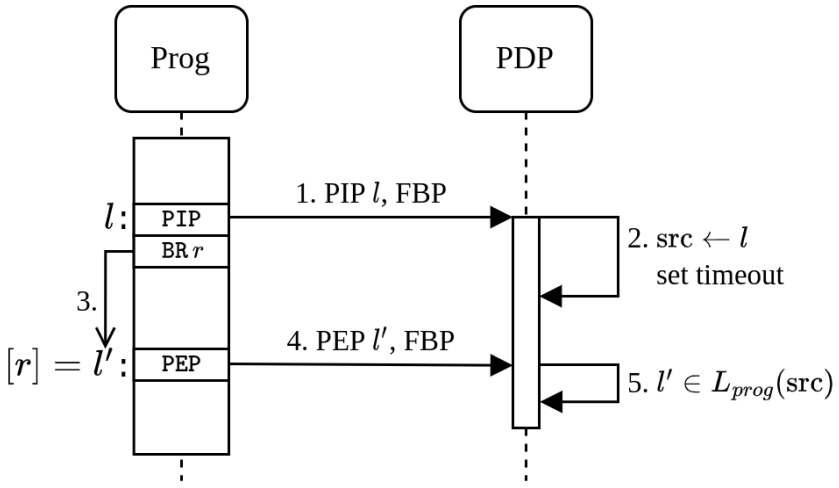


Figure 5.1 Forward Branch Policy (FBP) enforcement.

[6]For simplicity, we assume that PIP and PEP share a single address with the ToyRISC instruction they refer to. Full details are given in Section 5.3.3.

Backward Branch Policy (BBP) enforcement occurs instead as showed in Figure 5.2. To let the PDP know about the intended return point of a backward branch located at l, a PIP instruction informs the PDP right before the associate forward branch (tainted or no) at location l' (1). The PDP pushes $l'+1$, i.e., the return location, on its shadow stack S_ω (2), and the execution proceeds with the branch (3). Right before the $\texttt{POP}_{pc}$ instruction is executed, a PIP instruction is triggered to alert the PDP (4). This has the same meaning as for FBP policy, as it only serves to inform the PDP about the beginning of a tainted branch. The only thing that is done by PDP is the timeout setting (5), as the copy of l into *src* is meaningless here. After this, the control returns to the original branch location by means of a $\texttt{POP}_{pc}$ instruction (6) and a PEP is triggered (7), so that the PDP can compare the return address with the top element of its shadow stack (8).

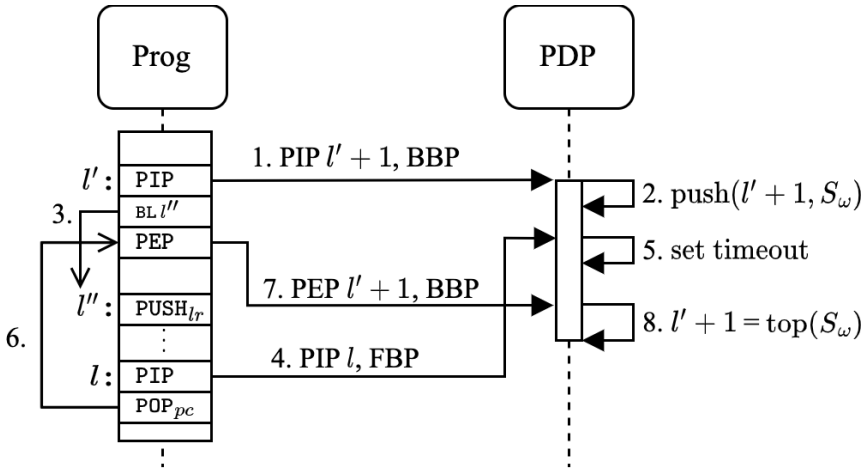


Figure 5.2 Backward Branch Policy (BBP) enforcement.

The above protections would be enough to cover all the cases in which a branch can be redirected to an unintended destination, were it not for the presence of interrupts (see Subsection 5.1.3 and Section 4.2). When an interrupt is fired ($\lightning i$), a pair of branches oustide the L_{prog} map are created: one connects current $prog[pc]$ to $\text{init}(prog_i)$, while the other one connects the exit point of $prog_i$ to $prog[pc+1]$. At offline analysis time, it is not possible to know source and destination locations of these branches.

In addition, when executing $prog_i$, the processor spills the context of the currently executed program in stack, in order to be able to restore it when the execution is

resumed (see Definition 6). $prog_i$ could contain attacker steps as in Definition 8. In particular, the following can occur:

- the reloaded value of *pc* might have been corrupted at return time, with no possibility of recovering the valid value. In fact, no BBP enforcement can be done without the knowledge of call site, as in this case;
- all the other register values might have been corrupted too, making Definition 12 invalid, as new unexpected flows, out of the L_{prog} map, might have created new tainted branches.

Therefore, it is necessary to add a third policy to be enforced:

3. **Interrupt Handler Policy (IHP)**: *Each interrupt handler program shall not bring any change to the context and the control-flow of the main program as from the static analysis mentioned in 5.3.1.*

In addition to the creation of a new specific policy, another issue related to interrupts links to the proper application of the already-existing FBP and BBP policies. Indeed, suppose that $\lightning i$ occurs exactly between a PEP and the following just-validated branch. The branch argument might undergo a change within $prog_i$ as a result of the rule in Definition 8, and even if a correct return is made from $prog_i$, the branch instruction might take a different path from the one previously validated. This would create an evident case of TOCTOU (Time of Check vs. Time of Use) vulnerability.

Therefore, to preserve the context of the main program and avoid the problems introduced from interrupts described above, a PIP could be inserted at the beginning of $prog_i$ to transmit to the PDP the value of the top of the stack, containing the program counter of the main program. The same type of protection could be adopted for every register, as they might be used by the interrupt handler for its computation. Before the interrupt handler exits, the code might restore the old values of the registers. Immediately before the IRET, a PEP could be used to check the identity of the register values with the ones at the handler entry, and of the destination program counter, which is on top of the stack.

The third policy enforced by the PDP is thus displayed in Figure 5.3, and it proceeds as follows. When an interrupt $\lightning i$ happens at location l, the execution jumps

to the interrupt handler prog_i (1). Then, a PIP instruction triggers the PDP by sending the element currently occupying the top of the program stack, denoted by $[sp-1]$, which amounts to l (2). This value is then pushed on the shadow stack S_ω, and a backup of all the registers R is done using R_ω (3)[7]. Finally, at the end of the interrupt handler execution, a PEP triggers the PDP (4) to check the top of the program stack $[sp-1]$ and the values of the registers R against those stored by the PDP in S_ω and R_ω, respectively.

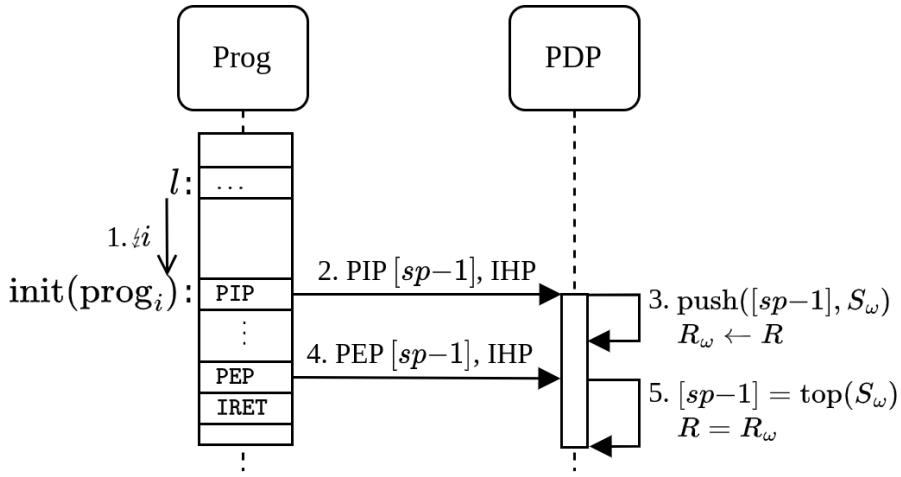


Figure 5.3 Interrupt handler policy enforcement.

There are two types of cases in which the PDP detects an illegal condition with respect to control-flow integrity:

1. During any failure in the three policy verifications described above;
2. When $t \leq$ timeout.

Any of the two results in the PDP halting the target program. In particular, as for the second case, this is crucial for security, because the timeout is set equal to the number of steps needed to perform a branch (i.e., 1). PIP and PEP instructions are not advancing the count, so that they are not influencing the exact step measurement. If an illegal branch is taken, the flow continues at a program label of attacker's choice after the execution, e.g., of the PIP and the branch itself. Without the timeout, the PDP could be forever waiting the corresponding PEP, without sensing the attack.

[7]Backing up all registers at once is a non atomic operation that, when PROLEPSIS is adopted in actual implementations, can be done, e.g., through multiple stores, as we will discuss in the next Chapters.

$$\frac{\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M', R', pc', sp', lr'\rangle \qquad t \leq \text{timeout}}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M', R', pc', sp', lr'\rangle, \langle S_\omega, R_\omega, src, t+1\rangle)} \text{(Skip)}$$

$$\frac{prog[pc] = \texttt{PIP}\, val\,, FBP \qquad \text{timeout} \leftarrow t+1 \qquad \text{disable}(\lightning)}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle S_\omega, R_\omega, val, t\rangle)} \text{(PIP-F)}$$

$$\frac{prog[pc] = \texttt{PEP}\, val\,, FBP \qquad val \in L_{prog}(src) \qquad t \leq \text{timeout} \qquad \text{timeout} \leftarrow \infty \quad \text{enable}(\lightning)}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle)} \text{(PEP-F)}$$

$$\frac{prog[pc] = \texttt{PIP}\, val\,, BBP \qquad \text{disable}(\lightning)}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle \text{push}(val, S_\omega), R_\omega, src, t\rangle)} \text{(PIP-B)}$$

$$\frac{prog[pc] = \texttt{PEP}\, val\,, BBP \qquad val = \text{top}(S_\omega) \qquad \text{timeout} \leftarrow \infty \qquad \text{enable}(\lightning)}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle \text{pop}(S_\omega), R_\omega, src, t\rangle)} \text{(PEP-B)}$$

$$\frac{prog[pc] = \texttt{PIP}\, val\,, IHP}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle \text{push}(val, S_\omega), R, src, t\rangle)} \text{(PIP-I)}$$

$$\frac{prog[pc] = \texttt{PEP}\, val\,, IHP \qquad val = \text{top}(S_\omega) \qquad R = R_\omega}{(\langle prog, M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon} (\langle prog, M, R, pc{+}1, sp, lr\rangle, \langle \text{pop}(S_\omega), R_\omega, src, t\rangle)} \text{(PEP-I)}$$

Table 5.3 Formal semantics of reference monitor.

Table 5.3 shows the rules of the formal semantics of programs executed under the control of our reference monitor. A monitored program state is a pair (C, Ω) where C is a program configuration and Ω is a reference monitor configuration. Briefly, these rules implement the behaviors for the forward (rules (PIP-F) and (PEP-F)), backward (rules (PIP-B) and (PEP-B)) and interrupt (rules (PIP-I) and (PEP-I)) policies. In addition to the already described behaviors, as far as FBP and BBP policies are concerned, all PIP disable interrupts, while all PEP enable them, with the only exception of PIP-B: this happens because, since a PIP precedes a branch and a PEP follows it, the execution of this triad (PIP - Branch - PEP) must be made as an atomic operation, to prevent an interrupt from inserting itself in the middle of the protection and leading the flow to an invalid execution (see Section 4.2).

Moreover, rule (Skip) is applied during the execution of program instructions that are neither PIP nor PEP. During the execution of a (Skip) step, the only thing that happens is advancing the step counter t, and checking that it has not gone beyond the limit imposed by the timeout.

5.3.3 Program Instrumentation

The goal of the program instrumentation is to deploy PIP and PEP within the instructions of the target program. As depicted in Figures 5.1, 5.2, and 5.3, PIP and PEP operations are expected to *wrap* the execution of certain branches, e.g., those detected as explained in Section 5.3.1. The procedure instrument a target program *prog* by adding instructions `PIP` and `PEP`. The result is a different program, $Instr(prog)$, which has all the labels of *prog* plus those added by the algorithm.

The pseudo-code of the instrumentation procedure for the main program is given in Algorithm 1.

Input: Program *prog*
Output: Program $Instr(prog)$
1 **foreach** l *s.t.* $prog[l] \in \mathrm{TB}(prog)$ **do**
2 **if** $prog[l] \in \{\mathtt{BR}\,r, \mathtt{BLR}\,r\}$ **then**
3 **instrument** $\mathtt{PIP}\,l, \mathtt{FBP}$ **at** l;
4 **foreach** $l' \in L_{prog}(l)$ **do**
5 **instrument** $\mathtt{PEP}\,l', \mathtt{FBP}$ **at** l';
6 **else if** $prog[l] = \mathtt{POP}_{pc}$ **then**
7 **instrument** $\mathtt{PIP}\,l, \mathtt{FBP}$ **at** l;
8 **foreach** l' *s.t.* $l \in \mathrm{assoc}(l')$ **do**
9 **instrument** $\mathtt{PIP}\,l'+1, \mathtt{BBP}$ **at** l';
10 **instrument** $\mathtt{PEP}\,l'+1, \mathtt{BBP}$ **at** $l'+1$;

Algorithm 1: Program instrumentation.

For each address l which contains a tainted branch (line 1), one of the following actions executed. If the branch refers to either `BR` or `BLR` (line 2), a $\mathtt{PIP}\,l, \mathtt{FBP}$ instructions is instrumented *right before* it. (i.e., at location l). The instrumentation is performed through the **instrument** *inst* **at** l command (line 3) which adds the instruction *inst* at location l. Also, this command creates a fresh label l' located right after l where the former occupant of l is moved[8]. Then, the algorithm proceeds by instrumenting $\mathtt{PEP}\,l', \mathtt{FBP}$ at every valid branch destination l' (lines 4-6). Otherwise, if the tainted branch at location l amounts to a $\mathtt{POP}_{pc}$ instruction (line 7), the algorithm behaves as follows. First, as for `BR` and `BLR`, a $\mathtt{PIP}\,l, \mathtt{FBP}$ is injected at l (line 7).

[8]The instrumentation process creates a new program from the previous one. When PROLEPSIS is imported into a real implementation, this process must take into account *static references* in the binary code, and must "shift" them accordingly. This will be duly discussed in subsequent Chapters. Here, we simply follow our definition of ToyRISC labels (see Section 5.1), which allows us to freely create new labels without compromising the program structure of the original program.

Then, for each l' such that $l \in \text{assoc}(l')$ (line 8) we instrument $\texttt{PIP}\,l'+1, \texttt{BBP}$ at l' and $\texttt{PEP}\,l'+1, \texttt{BBP}$ at $l'+1$ (lines 9 and 10).

Instead, Algorithm 2 reports the code for interrupt handlers instrumentation. It simply instruments each interrupt handler $prog_i$ by inserting a $\texttt{PIP}\,[sp-1]$, $\texttt{IHP}$ instruction at its very beginning and a $\texttt{PEP}\,[sp-1]$, $\texttt{IHP}$ instruction right before its (single) $\texttt{IRET}$.

Input: Handler $prog_i$
Output: Handler $InstrH(prog_i)$
1 **instrument** $\texttt{PIP}\,[sp-1]$, $\texttt{IHP}$ **at** init$(prog_i)$;
2 **let** l *s.t.* $prog_i[l] = \texttt{IRET}$;
3 **instrument** $\texttt{PEP}\,[sp-1]$, $\texttt{IHP}$ **at** l;

Algorithm 2: Interrupt handler instrumentation.

The full instrumentation of program $prog$ and its interrupt handlers $prog_i$ (with $i \in \{1,\ldots,n\}$ is denoted by $Instr(prog)$ and $InstrH(Instr(prog_i))$, i.e., every $prog_i$ is instrumented by both algorithms. For brevity we write $InstrH(prog_i)$ in place of $InstrH(Instr(prog_i))$.

5.3.4 Correctness

All the formalism presented above is adopted to prove the two theorems below, which are intended to give a formal proof to the correctness and completeness of the checks provided by PROLEPSIS. Specifically, the first theorem states that an execution instrumented and monitored as described above is formally equivalent to a legal execution. The *safety* of PROLEPSIS is thus theorized.

Theorem 1. *If $C \twoheadrightarrow_A^* C'$ is a legal run of $prog$, then it is also a run of $Mon(Instr(prog))$.*

In contrast, the second theorem states that a program instrumented and monitored as described above cannot original illegal executions. The *soundness* of PROLEPSIS is thus theorized.

Theorem 2. *For every program $prog$ and mapping L_{prog}, $Mon(Instr(prog))$ cannot lead to illegal runs.*

5.4 Technical Proofs

The present Section is intended to give the reader the technical proofs for the lemmas and the theorems presented in the current Chapter.

Lemma 1. *Every arbitrary long illegal run is of the form* $C \twoheadrightarrow_A^* C_0 \twoheadrightarrow C_1 \twoheadrightarrow_A^* C'$ *where* (i) $C \twoheadrightarrow_A^* C_0$ *is a legal run and* (ii) $C_0 \twoheadrightarrow C_1$ *is an illegal branch.*

Proof. By Definition 11, any illegal run $C \twoheadrightarrow_A^* C'$ contains at least one illegal branch. Let $C_0 \twoheadrightarrow C_1$ be the first such illegal branch occurring in the run. We have two cases. Either (i) $C \twoheadrightarrow_A^* C_0$ is a legal run, which suffices to conclude the proof, or (ii) $C = C_0$. However, case (ii) cannot occur. As a matter of fact, assuming (ii) holds, $C \twoheadrightarrow C_1$ would be illegal, thus contradicting Definition 10. □

Lemma 2. *If* $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow C$ *is an illegal branch w.r.t.* L_{prog}*, then* $prog[pc] \in \{\texttt{BR}, \texttt{BLR}, \texttt{IRET}, \texttt{POP}_{pc}\}$.

Proof. We prove that instructions different from `BR`, `BLR`, `IRET`, and $\texttt{POP}_{pc}$ cannot cause illegal branches. We proceed by structural induction over the small step semantics rules.

- Case (Branch). In this case, $prog[pc] = \texttt{B}\,l$ and $C = \langle prog, M, R, l, sp, lr\rangle$. By Definition 10, $l \in L_{prog}(pc)$.
- Case (Branch&Link) Here $prog[pc] = \texttt{BL}\,l$ and $C = \langle prog, M, R, l, sp, pc+1\rangle$. Again, by Definition 10, $l \in L_{prog}(pc)$.
- Case (Return). Here $prog[pc] = \texttt{RET}$ and $C = \langle prog, M, R, lr, sp, lr\rangle$ with $lr \in L_{prog}(pc)$ by Definition 10.
- Case (Conditional0). In this case $prog[pc] = \texttt{CBZ}\,r\,,\,l$ and $C = \langle prog, M, R, l, sp, lr\rangle$ with $l \in L_{prog}(pc)$.
- Cases (Conditional1), (Push), (Pop), (PushLr), (Increment), (Decrement), (AOp), (MoveRegister), (MoveImmediate), (Load), (Store). In all these cases, $C = \langle prog, M', R', pc+1, sp', lr\rangle$, for some M', R' and sp'. Again, for each of them, $pc+1 \in L_{prog}(pc)$.

□

Lemma 3. *Let S_ω and S'_ω be the monitor stack when an instrumented program receives an interrupt signal and the monitor stack returned after the execution of the interrupt handler (respectively). Then, $S_\omega = S'_\omega$.*

Proof. We first notice that only rules (PIP-B), (PIP-I) and (PEP-B), (PEP-I) push and pop (respectively) to/from S_ω. To conclude we just observe that instructions push and pop instructions always happen in pairs and no interrupt can occur in between. In the case of (PIP-B) and (PEP-B), Algorithm 1 puts the PIP and PEP instructions in sequence before and after the target label l'. Instead, Algorithm 2, puts PIP and PEP instructions for rule (PIP-I) and (PEP-I) in the beginning and in the end (respectively) of the interrupt handler. In both cases, interrupts cannot occur between PIP and PEP instructions. In the first case, due to rule (PIP-B), which disables interrupts. In the second case, due to the interrupt semantics. □

Theorem 1. *If $C \twoheadrightarrow_A^* C'$ is a legal run of $prog$, then it is also a run of $Mon(Instr(prog))$.*

Proof. For Theorem 1 to hold we must prove that for every legal run $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow_A^* \langle prog, M', R', pc', sp', lr'\rangle$ there exists a corresponding run $(\langle Instr(prog), M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}^*$ $(\langle Instr(prog), M', R', pc'', sp', lr'\rangle, \langle S'_\omega, R'_\omega, src', t'\rangle)$, where $pc'' \in \{pc', pc'+1\}$ (depending on whether $prog[pc']$ is instrumented or not). We proceed by induction on the length of the run.

- Base step. Trivial, since the empty run is legal and always admitted by the monitor.
- Inductive step. We assume the property holds for $C \twoheadrightarrow_A^* \langle prog, M, R, pc, sp, lr\rangle$ and we prove it for $C \twoheadrightarrow_A^* \langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow_A \langle prog, M', R', pc', sp', lr'\rangle$. Then we proceed by induction on the derivation rules of $\twoheadrightarrow_A$. By Lemma 2, if $prog[pc] \notin \{\texttt{BR}, \texttt{BLR}, \texttt{IRET}, \texttt{POP}_{pc}\}$ the whole trace is legal. Also, following Algorithm 1, $Instr(prog[pc]) = prog[pc]$, that is, $prog[pc]$ is not instrumented and rule (Skip) is applied. To conclude these cases, we just need to notice that, since no $\texttt{PIP}F$ instruction appears before $prog[pc]$, timeout is set to ∞. Then we prove the remaining rules as follows.
  - (BranchR). In this case $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, v, sp, lr\rangle$ where $prog[pc] = \texttt{BR}\,\texttt{r}$ and $R[r] = v$, with $v \in L_{prog}(pc)$. By Algorithm 1,

$Instr(prog)[pc] = \texttt{PIP}\,l, FBP$ (with $l = pc$) and
$Instr(prog)[pc+1] = \texttt{BR}\,r$. Thus the following derivation takes place.

$(\langle Instr(prog), M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \quad \twoheadrightarrow_{Mon}$ (PIP-F)
$(\langle Instr(prog), M, R, pc+1, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t\rangle) \quad \twoheadrightarrow_{Mon}$ (Skip)
$(\langle Instr(prog), M, R, v, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t+1\rangle)$

Then, by Algorithm 1, $Instr(prog)[v] = \texttt{PEP}\,v, FBP$ and the computation proceeds with

$(\langle Instr(prog), M, R, v, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t+1\rangle) \quad \twoheadrightarrow_{Mon}$ (PEP-F)
$(\langle Instr(prog), M, R, v+1, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t+1\rangle)$

which suffices to conclude the case.

– (Branch&LinkR). Similar to the previous case.

– (PopPc). Here $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, v, sp-1, lr\rangle$ with $prog[pc] = \texttt{POP}_{pc}$ and $v = M(sp-1)$ for some $v \in L_{prog}(pc)$. By Algorithm 1, $Instr(prog)[pc] = \texttt{PIP}\,l, FBP$ (with $l = pc$) and $Instr(prog)[pc+1] = \texttt{POP}_{pc}$. Moreover, $v = l'+1$ for some l' s.t. $pc \in \text{assoc}(l')$. Thus, by Algorithm 1, $Instr(prog)[v] = \texttt{PEP}\,\texttt{v}, \texttt{BBP}$ and $Instr(prog)[v-1] = \texttt{PIP}\,\texttt{v}, \texttt{BBP}$. From this we have that $S_\omega = push(v, S'_\omega)$, for some S'_ω. Then, the computational steps are the following.
$(\langle Instr(prog), M, R, pc, sp, lr\rangle, \langle push(v, S'_\omega), R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}$ (PIP-F)
$(\langle Instr(prog), M, R, pc+1, sp, lr\rangle, \langle push(v, S'_\omega), R_\omega, pc, t\rangle) \twoheadrightarrow_{Mon}$ (Skip)
$(\langle Instr(prog), M, R, v, sp-1, lr\rangle, \langle push(v, S'_\omega), R_\omega, pc, t+1\rangle) \twoheadrightarrow_{Mon}$ (PEP-B)
$(\langle Instr(prog), M, R, v+1, sp-1, lr\rangle, \langle S'_\omega, R_\omega, pc, t+1\rangle)$

– (RetInt) Here $\langle prog_i, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, v, sp-1, lr\rangle$ with $prog_i[pc] = \texttt{IRET}$ and $v = M(sp-1)$ for some v satisfying the second case of Definition 11. By Algorithm 2, $InstrH(prog_i)[pc] = \texttt{PEP}\,[sp], IHR$. Moreover, by Lemma 3, $S_\omega = push(sp, S'_\omega)$, for some S'_ω, and, from Definition 7, $R_\omega = R$. Hence, the following execution steps conclude the proof.
$(\langle InstrH(prog_i), M, R, pc, sp, lr\rangle, \langle push(sp, S'_\omega), R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}$ (PEP-I)
$(\langle InstrH(prog_i), M, R, pc+1, sp, lr\rangle, \langle S'_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}$ (Skip)
$(\langle Instr(prog), M, R, v, sp-1, lr\rangle, \langle S'_\omega, R_\omega, pc, t+1\rangle)$

□

Theorem 2. *For every program prog and sound L_{prog}, $Mon(Instr(prog))$ cannot lead to illegal runs.*

Proof. By Lemma 1, every illegal run consists of a finite prefix of legal steps $C \twoheadrightarrow_A^* C_0$ followed by an illegal branch $C_0 \twoheadrightarrow C_1$. By Theorem 1, $C \twoheadrightarrow_A^* C_0$ is also a run of $Mon(Instr(prog))$. Also, by Lemma 2, illegal branch can only occur at instructions BR, BLR, IRET and $\texttt{POP}_{pc}$. Thus, we conclude by proving that such illegal branches do not belong to the monitor semantics.

- Case BR. In this case we have $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, v, sp, lr\rangle$ where $prog[pc] = \texttt{BR r}$ and $R[r] = v$, with $v \notin L_{prog}(pc)$. By Algorithm 1, $Instr(prog)[pc] = \texttt{PIP}\,l, FBP$ (with $l = pc$) and $Instr(prog)[pc+1] = \texttt{BR}\,r$. Thus the following derivation takes place.

  $(\langle Instr(prog), M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}$ (PIP-F)
  $(\langle Instr(prog), M, R, pc+1, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t\rangle) \twoheadrightarrow_{Mon}$ (Skip)
  $(\langle Instr(prog), M, R, v, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t+1\rangle)$

  Then, since $v \notin L_{prog}(pc)$, three cases arise for $Instr(prog)[v]$, i.e., either (i) it is a PEP instruction, (ii) it is a standard ToyRISC instruction, or (iii) it is a PIP instruction. In case (i) the computation is halted by the monitor since $v \notin L_{prog}(pc)$. In case (ii) the timeout is reached and the program terminates. In case (iii) the execution may proceed. However, since the timeout cannot be reset, after a finite number of PIP instructions, the execution must reach a ToyRISC instruction as in case (ii).

- Case BLR. Symmetric to the previous one.

- Case $\texttt{POP}_{pc}$. In this case we have $\langle prog, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, M(sp-1), sp-1, lr\rangle$ where $prog[pc] = \texttt{POP}_{pc}$ with $M(sp-1) = v \notin L_{prog}(pc)$. By Algorithm 1, $Instr(prog)[pc] = \texttt{PIP}\,l, FBP$ (with $l = pc$) and $Instr(prog)[pc+1] = \texttt{BR}\,r$. Thus the following derivation takes place.

  $(\langle Instr(prog), M, R, pc, sp, lr\rangle, \langle S_\omega, R_\omega, src, t\rangle) \twoheadrightarrow_{Mon}$ (PIP-F)
  $(\langle Instr(prog), M, R, pc+1, sp, lr\rangle, \langle S_\omega, R_\omega, pc, t\rangle) \twoheadrightarrow_{Mon}$ (Skip)
  $(\langle Instr(prog), M, R, M(sp-1), sp-1, lr\rangle, \langle S_\omega, R_\omega, pc, t+1\rangle)$ Then, since $v \notin L_{prog}(pc)$, the same three scenarios of case BR can occur, which suffices to conclude.

- Case IRET. Here we must consider the second case of Definition 11. Hence, we have $\langle prog_i, M, R, pc, sp, lr\rangle \twoheadrightarrow \langle prog, M, R, M(sp-1), sp-1, lr\rangle$ where $prog_i[pc] = \texttt{IRET}$ with $M(sp-1) = v$, for some value of v. By Algorithm 2, $Instr(prog)[pc] = \texttt{PEP}\,[sp-1], IHP$ and $Instr(prog)[pc+1] = \texttt{IRET}$. Also, by assumption, this is the first illegal branch in the execution (and interrupts cannot be nested), according to Algorithm 2 and rule (PIP-I), $S_\omega = push(v', S'_\omega)$ for some $v' \neq v$. Which suffices to conclude.

□

Chapter 6

Design for Security Approaches and Experimental Assessment

The theory presented in the previous Chapter provides a formal basis for an embedded systems protection framework that, in the author's view, brings several innovations from the current state of knowledge. First, a formal abstraction with this level of completeness can be adapted to any embedded application context as long as it falls within the boundary outlined by the assumptions made at the beginning of Chapter 5.

The domain of resource-constrained devices is a diverse one, where multiple realities coexist, resulting in software and hardware designs with a plethora of declinations. As discussed in the previous Chapters, design choices for an embedded system can stand out on different *degrees*, ranging from the highest degree of *flexibility* to the highest degree of *specificity*. In fact, we start from designs that are completely customized even from the silicon point of view, such as ASICs that execute algorithms only by virtue of their FSMs with datapath, without even software, through FPGAs configured to execute a specific algorithm, custom-ISA hard processors, custom-ISA soft processors implemented through FPGAs, down to off-the-shelf microcontrollers for which custom firmware is designed.

The present thesis focuses on a type of design distinguishing a software part from a hardware part, and that specifically want to design the hardware part to defend against vulnerabilities that may be present in the software (*vulnerability-tolerant architectures*). In such a context, there are two kinds of situations in the design phase: either there is the possibility to customize the *instruction set* of the processor, e.g.,

to make it more adherent to the specific mission of the device, or there is no such possibility, and the behavior of the system is completely decided by the software layer. From the point of view of the framework offered here, it must be granted that, in all cases, there is the possibility of supporting the actual implementation of a reference monitor, which performs the security checks in a way that has the least possible impact on the application for which the system was intended. From the rules derived from the previous Chapter, it can be noticed that PROLEPSIS does not mandate on *how* and *where* to implement the monitor. So, a solution that integrates the monitor into the software itself is perfectly compliant with the framework. However, PROLEPSIS specifies that the monitor must be a separate entity, at least logically. The main reason for this is that it contains *sensitive information* about the control-flow graph of the program, which needs to be duly protected from the intrusions of an attacker. When the reference monitor is implemented within the program itself (following, for example, the example of solutions such as Silhouette or Kage), it is natural that the monitor still continues to be a circumscribed and protected entity (in that case, for example, by the MPU).

Instead, in this Chapter, we present two different approaches for the design of vulnerability-tolerant architectures in the sense of control-flow integrity, which make use of *hardware monitors*, in both cases where one can or cannot customize the processor instruction set. In other words, the monitor can be:

- *External*, i.e., strictly connected via an interface (e.g., serial or parallel) to the processor responsible for executing the program, which receives the information needed to perform the checks;
- *Internal*, i.e., integrated within the processor and confined to a set of modules to perform verifications;

In both cases, the monitor:

- does not interfere in any way with the normal operation of the processor, i.e., normal program activity is completely transparent to the existence of the monitor;
- intervenes only when duly addressed (cfr. PIP, PEP) and in no other case;

- stops program activity only when it detects a violation, i.e., does not pause execution to perform checks (a feature that necessarily distinguishes it from a fully integrated software monitor, for example).

In this Chapter, two possible real transpositions of the framework into two practical defenses are presented, i.e., one based on an external monitor and one based on an internal monitor. Of these instances, implementation details are provided, and experimental results from their application on two reference platforms are presented, in terms of overhead on execution time, code size, and additional hardware required. From the point of view of quantitative security evaluation, a research work done to adapt a reference framework in the domain of runtime evaluators is presented.

Then, the implementation of Algorithms 1 and 2 in a binary instrumentation tool adapted for multiple machine languages is presented, within which PIP/PEP instrumentations can be translated into a custom instruction sequence and decided by contingencies and the type of monitor employed.

6.1 External Monitor

For the empirical evaluation of the case where the CFI monitor is implemented via an external hardware parallel module, the ARM Cortex-M4 CPU was chosen as the target architecture. The CPU supports the ARMv7-M ISA, among the most commonly used in the microcontroller industry. Specifically, the experiment was conducted on the STM32F429 microcontroller from STMicroelectronics[1]. The reference monitor logic is described in VHDL and synthesized on Lattice Semiconductor's MachXO2 FPGA[2]. The connection between the two chips is made possible by the Flexible Memory Controller (FMC), an interface provided by STMicroelectronics to communicate with external parallel memory modules, addressable in a logical address space fixed by those in the architecture's memory map. The implementation was first presented in 2020 [175].

Table 6.1 shows the branch instructions found in the ARMv7-M ISA, associating each of them with the abstraction in the ToyRISC formal language under which

[1] https://www.st.com/resource/en/datasheet/stm32f429ig.pdf

[2] https://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/MachXO23/MachXO2FamilyDataSheet.pdf

each instruction can be assimilated, to create a correspondence with the PROLEPSIS reference theory.

ARMv7-M	**ToyRISC**
`ADD PC, #imm12`	`B` *label*[3]
`ADD PC, PC, Rm`	`BR` *r*
`B label`	`B` *label*
`BL label`	`BL` *label*
`BLX Rm`	`BLR` *r*
`BX LR`	`RET`
`BX Rm`	`BR` *r*
`CBZ Rm, label`	`CBZ` *r*, *label*
`CBNZ Rm, label`	`CBZ` *r*, *label*[4]
`LDR PC, [Rn, #offset]`	`BR` *r*[5]
`MOV PC, Rm`	`BR` *r*
`POP PC`	$\texttt{POP}_{pc}$
`SUB PC, #imm12`	`B` *label*[3]
`TBB [Rm, Rn]`	`BR` *r*
`TBH [Rm, Rn]`	`BR` *r*

Table 6.1 ARMv7-M ISA branch instructions, with corresponding ToyRISC abstraction.

The correspondence between other ToyRISC instructions in Table 5.1 and ARMv7-M instructions for data and memory operations can be easily identified and is not relevant to this discussion.

At vulnerable branch instructions and entry/exit points of the ISRs, identified by a proper implementation of Algorithms 1 and 2 (see next Sections for details), it is necessary to translate PEP and PIP into communication instructions with the CFI monitor. As can be clearly seen from Figures 5.1, 5.2, and 5.3, the flow of information is always from the program to the monitor and never vice versa, for an obvious security issue: all information is transmitted to the PDP, and that is where decisions are made, with no need to provide any other information to the program, except a signal to interrupt its execution. Therefore, because of the memory-mapped nature of the interaction, instrumentation instructions are rendered through the use of write (`STR` in the ARMv7-M ISA). In addition to identifying the points to be instrumented and appropriately translating PIP and PEP into ARM instructions, the

[3] With proper label choice.
[4] With proper register negation.
[5] After proper memory-to-register loading.

offline phase is also concerned with removing any references to the memory area reserved for the monitor. Figure 6.1 traces the connections between the execution environment, consisting of the CPU, and the PDP, consisting of the external FPGA. There are three signals that are part of the interface:

- A 16-bit data bus;
- A 6-bit address bus, representing the low bits of the offset from the peripheral's base address;
- An interrupt signal, to signal a highest priority violation to the CPU.

In addition, reset, clock, and write-enable signal are asserted by the CPU.

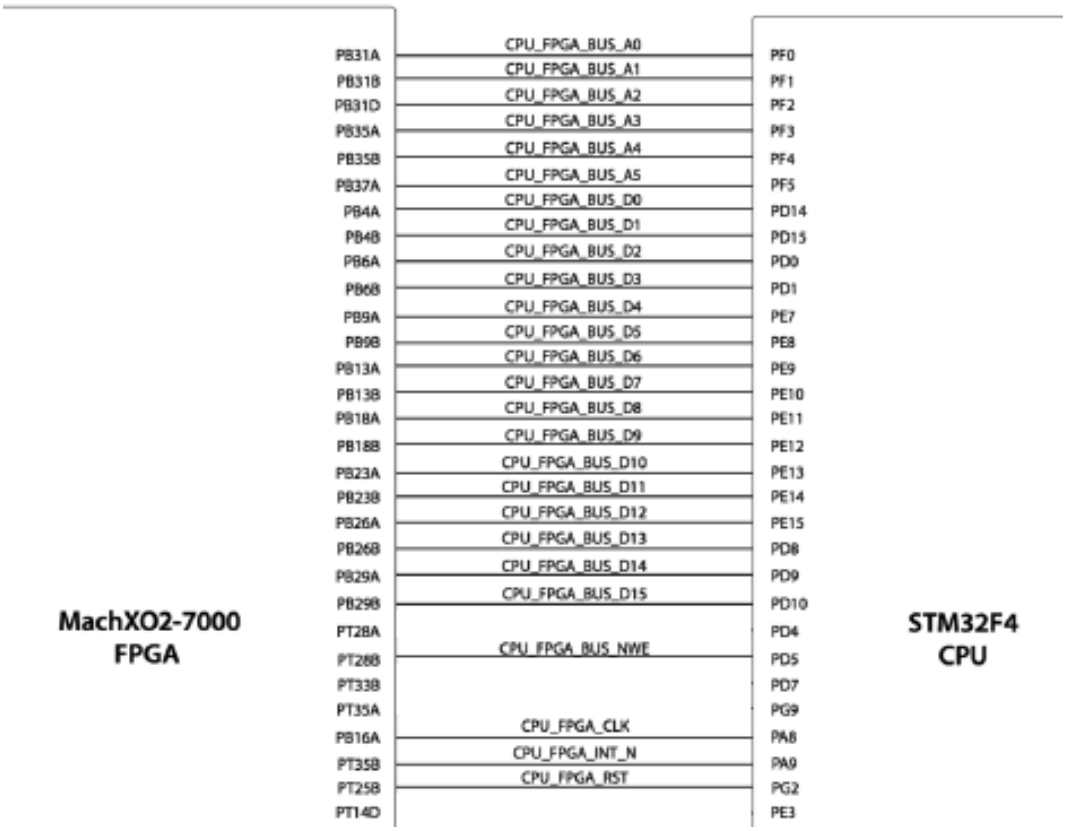


Figure 6.1 The pin-to-pin connection of the CPU-FPGA interaction.

The monitor is a High-Level State Machine (HLSM) mainly consisting of two areas: a *logic area*, responsible for checks and interpretation of commands coming from the CPU, and a *memory area*. The latter contains three memory elements:

- a *Secure Edge Table* (SET), which encodes the information about all permitted control-flow transfers, as pairs of source and destination labels (cfr. L_{prog} in Definition 10);
- a *Secure Label Stack* (SLS), where it pushes the labels received to protect backward insecure branches (cfr. S_ω in Definition 14);
- a *Secure Register Stack* (SRS), where it pushes the context of the program upon entering an ISR (cfr. R_ω in Definition 14).

The logic area, on the other hand, consists of a *Control & Check Unit* (CCU). Its task is to decode the commands coming from the CPU to generate consequent reads and writes on these three storage blocks, as well as it verifies, through a set of comparators, that the received data are the expected ones. Also contained within is then a *clock cycle counter* (cfr. t in in Definition 14) and a temporary register to store the source label of a branch to be monitored (cfr. *src* in Definition 14).

SLS and SRS are architected and behave as two standard LIFO-type data structures. SET is a read-only memory (ROM), accessed with a combination of the source label (i.e., communicated via PIP) and destination label (i.e., communicated via PEP) of the branch to be protected. Specifically, the source label is saved in a temporary register, and when the destination label arrives, a modular *sum* ($+$) is made between the two. The sum is used as the address to access SET. Two situations can occur here:

- Static analysis did not detect any coupling between the two labels, so the branch is not among those allowed: the entry contains a null value;
- The branch is allowed: the entry contains a validity flag, concatenated with the source label. This is done to uniquely identify the branch.

Figure 6.2 summarizes the monitor architecture. The data bus carries two types of information to the monitor:

- a unique label, representing a position in the code. This need not be the source address of the branch, but can be any unique identifier for that position;
- a register value.

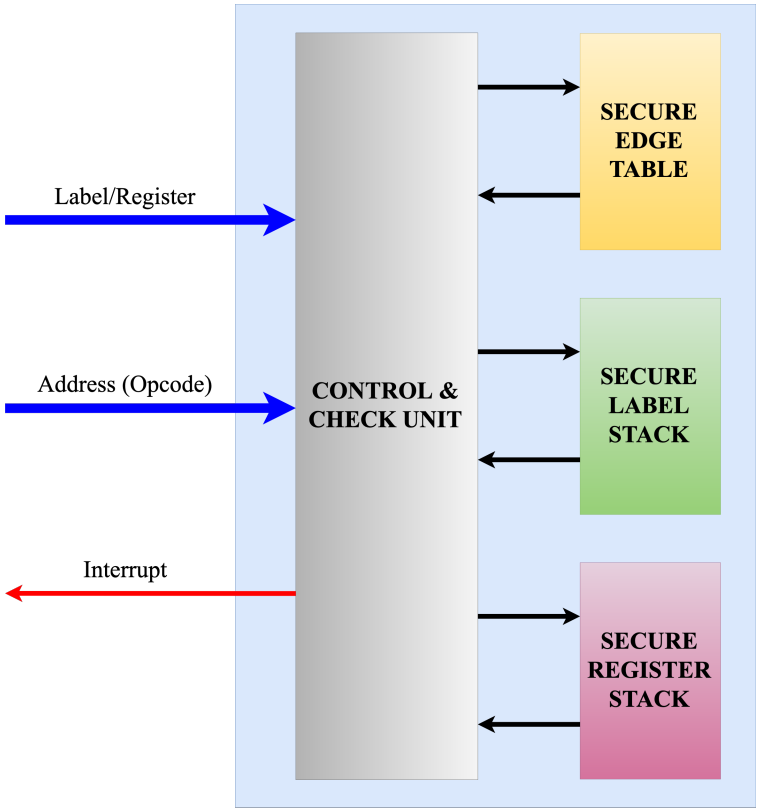


Figure 6.2 The structural hardware view of the FPGA-based CFI monitor.

Since the monitor is seen by the execution environment as a write-only memory, the arrival of each data item is associated with an *address*. The latter is not used as the actual address, but is interpreted by the CCU as an opcode to perform the proper operation on the data and transit to the next state. At the monitor boundary, there is an input detector, which is able to recognize 6 different opcodes (i.e., addresses relative to the base address of the monitor peripheral[6]):

- **0x0**: indicates that a *tainted forward branch* is to be executed next, and the transmitted data is its source label (cfr. PIP-F in Table 5.3). The monitor registers the label, activates the timer, and goes into a wait state that accepts only the branch destination as the next data item;
- **0x2**: indicates that a simple *function call* is to be executed next, and the transmitted data is the destination label of the backward branch associated with that call (cfr. PIP-B in Table 5.3). The monitor pushes the label to SLS. It

[6]Addresses are multiple of 2 because of the 16-bit parallelism of the connection.

is important to note that the site is instrumented only if the associated backward branch is tainted, as established by PROLEPSIS;

- **0x4**: indicates that a *tainted backward branch* is to be executed next, and the transmitted data is its source label (cfr. PIP-F number 4 in Figure 5.2). The monitor only activates the timer and goes into a waiting state that accepts only the branch destination as the next data item;

- **0x6**: indicates that a *function call through tainted forward branch* (e.g., `BLX`) is to be executed next, and the transmitted data is its source label (cfr. PIP-F and PIP-B together in Table 5.3). The label also simultaneously indicates the destination site of the associated backward branch[7]. The monitor performs two operations: pushes the label to SLS and registers the label into *src*, activating the timer and going in a wait state that accepts only the branch destination as the next data item;

- **0x8**: indicates that the tainted branch, backward or forward, has arrived at its destination, and the transmitted data is its destination label (cfr. PEP-F and PEP-B in Table 5.3). First, the timer is stopped. Then, depending on the last come opcode, the monitor performs several operations: if it was `0x0` or `0x6`, the destination label is added to the previously-saved source label, and SET is accessed. If the entry contains a valid value, nothing happens. If the entry contains an invalid value, an interrupt is raised. If a `0x4` opcode had previously arrived, the last SLS value is popped and compared with the arrived label. If they do not match, an interrupt is raised;

- **0xA**: indicates that the execution is in a ISR prologue, and the transmitted data is a register that has been moved, either by the architecture or by the programmer, on the stack (cfr. PIP-I in Table 5.3). In the case of ARM, the architecture performs this spill operation automatically for some fundamental registers, namely `R0`, `R1`, `R2`, `R3`, `R12` (frame pointer), `LR` (link register), `PC` (program counter), `xPSR` (processor status register). The monitor pushes the received value to SRS;

- **0xC**: indicates that the execution is in a ISR prologue, and the transmitted data is one of the registers that are intended to be filled back by automatic

[7] Since labels are decoupled from addresses, a single label can be chosen to indicate the call site and return site.

mechanism of the architecture, or have been reloaded by the programmer. The monitor extracts the last SRS value and compares it with the value that arrived. If they do not match, an interrupt is raised.

More details about the monitor architecture and functioning can be found in its Github repository[8].

Based on what presented, the instrumentation tool described later operates translations of PIP and PEP as listed in Tables 6.2, 6.3 and 6.4. The labels used are chosen randomly, as long as the aliasing that can occur when combining the source/destination labels of a branch is avoided. Their length depends on the dimension of the SET, as label combinations are used as address for this ROM. In this implementation, the maximum length is equal to the bit lenght of the data bus carrying information to the FPGA, i.e., 16 bits.

ToyRISC	**ARMv7-M**

PIP *label*, FBP
;forward branch instruction
here
	CPSID f ;disable interrupts
	PUSH {Rx, Rz}
	MOV Rx, =LABEL
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz]
	POP {Rx, Rz}
	;forward branch instruction
	here

PEP *label*, FBP
;forward branch destination
here
	PUSH {Rx, Rz}
	MOV Rx, =LABEL
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz, #8]
	POP {Rx, Rz}
	CPSIE f ;re-enable interrupts
	;forward branch destination
	here

Table 6.2 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 1).

[8] https://github.com/RHESGroup/cfi-fpga-monitor

ToyRISC	**ARMv7-M**

PIP *label*, BBP
;call instruction here
	PUSH {Rx, Rz}
	MOV Rx, =LABEL
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz, #2]
	POP {Rx, Rz}
	;call instruction here

PIP *label*, BBP
PIP *label*, FBP[7]
;indirect call instruction here
	PUSH {Rx, Rz}
	MOV Rx, =LABEL[7]
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz, #6]
	POP {Rx, Rz}
	;indirect call instruction here

PIP *label*, FBP
;backward branch instruction here
	CPSID f ;disable interrupts
	PUSH {Rx, Rz}
	MOV Rx, =LABEL
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz, #4]
	POP {Rx, Rz}
	;forward branch instruction here

;call instruction here
PEP *label*, BBP
	;call instruction here
	PUSH {Rx, Rz}
	MOV Rx, =LABEL
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH Rx, [Rz, #8]
	POP {Rx, Rz}
	CPSIE f ;re-enable interrupts

Table 6.3 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 2).

The protected execution environment is set up as follows. The static analysis and instrumentation phase produces (1) an executable binary file with the `.text` section instrumented and (2) a file listing all protected branches in the form source/destination label (S,D). The latter file is translated into a *memory initialization file* (`.mif`), where the valid entries have $S+D$ as address and a validity flag concatenated to S as value. The file is fed to the VHDL synthesizer as initializer for the Secure Edge Table as a ROM. The bitstream obtained from the synthesis is then flashed to the system's FPGA. As for the executable, in addition to the instrumentation, this

ToyRISC	ARMv7-M

;ISR entry point
PIP $[sp-1]$, IHP
	;ISR entry point
	PUSH {Rx, Ry, Rz}
	MOV Rz, =MONITOR_BASE_ADDRESS
	STRH R0, [Rz, #10] ;low part
	LSR Rx, R0, #16
	STRH Rx, [Rz, #10] ;high part
	;repeat same for R1, R2, R3, R12
	LDR Rx, [SP, #36] ;PC from stack
	STRH Rx, [Rz, #10]
	LSR Ry, Rx, #16
	STRH Rx, [Rz, #10]
	LDR Rx, [SP, #40] ;xPSR from stack
	STRH Rx, [Rz, #10]
	LSR Ry, Rx, #16
	STRH Rx, [Rz, #10]
	;repeat same for any other stack-spilled register
	POP {Rx, Ry, Rz}

PEP $[sp-1]$, IHP
IRET ; ISR exit point
	PUSH {Rx, Ry, Rz}
	MOV Rz, =MONITOR_BASE_ADDRESS
	LDR Rx, [SP, #offset] ;point
	;to stack-spilled register
	STRH Rx, [Rz, #12] ;low part
	LSR Ry, Rx, #16
	STRH Rx, [Rz, #12] ;high part
	;repeat for any other stack-
	;spilled register, following
	;the correct order and
	;incrementing #offset by 4
	POP {Rx, Ry, Rz}
	BX LR ;ISR exit point

Table 6.4 ARMv7-M machine instruction translation for PIP and PEP to the FPGA-based PDP (Part 3).

must be augmented with an `.init` section that contains the instructions needed to initialize the FPGA: set the FMC interface, set the clock, and reset the monitor. After system reset, this is done first, and then execution can jump to the main program, not before setting the `.init` section itself as `NO_ACCESS` via the MPU, which is

present in the ARM Cortex-M4. This prevents, for example, any code redirection from branching immediately to the FPGA reset instruction before the monitor itself can notice the violation. We call this portion of the executable as *Secure Initializer*.

Communication between CPU and FPGA is automatically synchronized, as the two chips share the same clock oscillator, internal to the CPU and passed to the monitor via dedicated pin. In general, in the reference architecture, the monitor runs at a speed that is submultiple of the CPU speed (at most equal), as the clock source is prescaled before sharing with the monitor. In the version implemented to test the solution, the FPGA has a frequency that is half of the CPU. In using the SET module, the monitor takes only 1 clock cycle (cc) to register the source label upon its arrival, while it takes 4 cc from the arrival of the destination label to the decision on the validity of the branch. For operations on the two shadow stacks, the duration is 3 cc for each push, as well as 3 cc for each pop plus identity check, again calculated from the arrival of the data. To these, latencies of the instructions composing the instrumentations must be added, as shown in Table 6.5.

Instruction	**Clock cycles**
`MOV`	1 cc
`STRH`	2 cc, but 1 cc if consecutive to another
`LDR`	2 cc, but 1 cc if consecutive to another
`PUSH`	1 cc + 1 cc for each register in the list
`POP`	1 cc + 1 cc for each register in the list
`CPSID`	1 cc
`CPSIE`	1 cc
branch (any)	3 cc (worst case)

Table 6.5 ARMv7-M instruction latencies for instructions used in the instrumentation [176].

For example, to calculate the latency of a practical case, in the combination of PIP-F and PEP-F, it must be taken into account a CPU overhead of 11 cc before the branch (which takes 3 in the worst case). However, during the last 3 cc of this instrumentation, 2 CPU cycles are occupied in parallel by the monitor for recording the source label. At the destination, again, 11 cc are taken, but after 7 of these, the monitor has the destination label and can begin its check, which takes 8 CPU cc. The result is that 13 cc elapse from the arrival of the source label to the arrival of the destination label (and this is the timeout value), and another 8 are used to figure out whether there has been a violation.

The danger that the attacker could perform an illicit action in that window of time exists in theory, and can be limited in practice by both optimization of the VHDL design and by the recovery procedure executed when a violation is detected. Indeed, a recovery function tailored to the specific case could undo any misdeeds. Other solutions might involve *diluting* particularly dangerous code areas with NOPs or with a lower level of optimization, so that their execution time is long enough that the monitor can sense a hijacking toward it. Furthermore, as for times, it must be said that the choice of having the SET as a ROM, accessed with complexity $O(1)$ through $S+D$ label combination, goes in the direction of reducing verification times. Precise details on synthesis results and overhead are reported later in this Chapter.

6.2 Internal Monitor

To evaluate the case where the CFI monitor is implemented internally in the processor, the AFTAB processor [177] was chosen. AFTAB is a 32-bit RISC-V computer architecture developed by the University of Tehran in collaboration with our team at Politecnico di Torino, for teaching and research purposes. The core implements the RV32IM ISA [178], which is the smallest extension of the instruction set represented by the basic arithmetic and logic operations on integers, with the addition of multiplication and division instructions. The processor is designed to be minimal in terms of area occupied, it is sequential, and does not support pipelining. AFTAB supports all the 32-bit instructions taken from the ISA, where the 7 least significant bits are reserved as *operative code* (*opcode*). The instructions are grouped in 4 different formats (R, I, U and S), as shown in Figure 6.3.

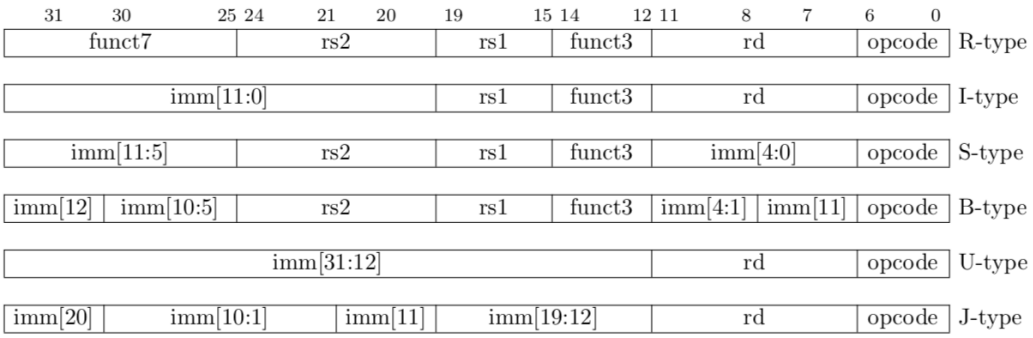


Figure 6.3 RISC-V base instruction formats showing immediate variants.

More details can be found within its Github repository[9]. The original design was modified to include internal modules to ensure control-flow integrity. The augmented CPU was synthesized and tested on a Xilinx Artix 7 FPGA.

Table 6.6 shows the branch instructions present in the RV32IM ISA, and their association with the abstraction provided by ToyRISC. Here, pseudoinstruction are listed, even if any branch instruction in RISC-V is derived from `JAL` or `JALR`, except for the conditional branches. Furthermore, the $\texttt{POP}_{pc}$ ToyRISC instructions seems to be not implemented in RV32IM. In fact, there is no actual machine instruction popping the program counter from the stack. Anyway, loading of *link register* (`X1` or `RA` in RISC-V) and subsequent `RET` can be considered equivalent. Correspondence with other logic, arithmetic, and data transfer instructions is not of interest for the purposes of this discussion and is omitted.

RV32IM	**ToyRISC**
`BEQZ Xs, label`	`CBZ` *r*, *label*
`BNEZ Xs, label`	`CBZ` *r*, *label*[10]
`CALL label`	`BL` *label*
`J label`	`B` *label*
`JAL label`	`BL` *label*
`JR Xs`	`BR` *r*
`JALR Xs`	`BLR` *r*
`RET`	`RET`
`TAIL label`	`BL` *label*

Table 6.6 RV32IM ISA branch instructions, with corresponding ToyRISC abstraction.

AFTAB has been modified to accommodate three new modules within it:

- a *Control-Flow Logic Verifier* (CFLV), which implements logic to combine the source and destination labels of branch sites and provide a validity flag (cfr. L_{prog} in Definition 10). The checker is implemented through a *Programmable Logic Array* (PLA) [44] that performs operations that are not fixed but reconfigurable depending on the contents of certain *configuration registers*;
- a *Control-Flow Address Stack* (CFAS), which automatically saves all return addresses at call times (cfr. S_ω in Definition 14);

[9] https://github.com/RHESGroup/aftab
[10] With proper register negation.

- a *Control-Flow Register Stack* (CFRS), which is to hold a secure copy of the context registers during the execution of an ISR (cfr. R_ω in Definition 14).;

Being embedded in the processor, the three modules are handled by appropriate states of the processor's internal Control Unit (CU) logic, and through appropriate datapath signals. At the critical points identified by the instrumentation tool, special instructions are needed to transform PIP and PEP into appropriate commands to trigger the modules, as we explain in the following.

The RTL view of CFAS is shown in Figure 6.4. The module reads directly from the *program counter* (`PC`) register within datapath, to use its value as input. Instead, the address is stored within a `ptr` register, which accepts increment (`push`) or decrement (`pop`) signals coming from a controller (`ctrl`). In turn, this receives two signals from the processor control unit telling whether the operation being performed is a call to function or a return from function. Upon return, data popped from the shadow stack is compared with that contained in the return address register, namely `X1`. The violation signaling is controller-mediated, and is directed to the processor control unit. The CFAS module does not need any custom instruction to operate.

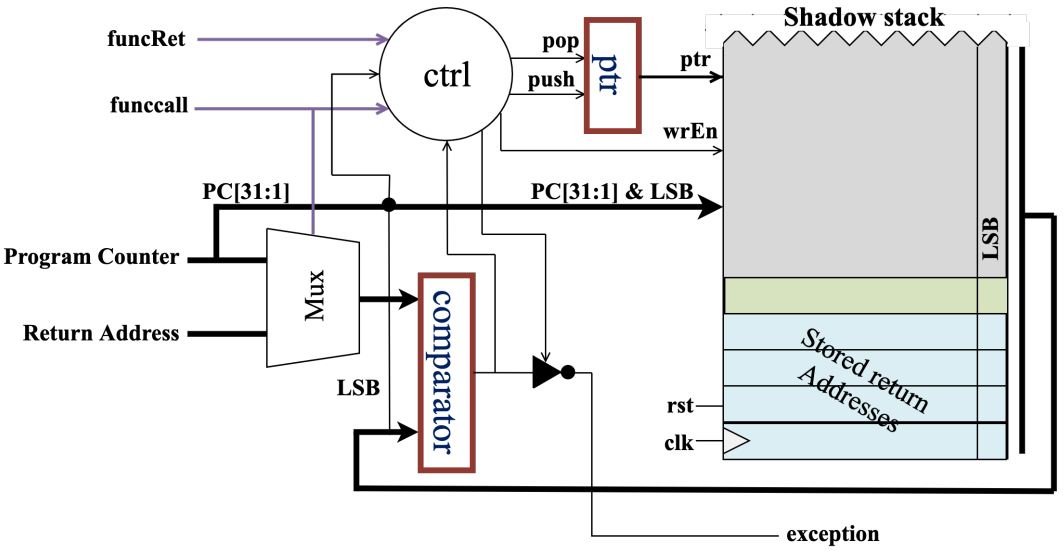


Figure 6.4 Control-Flow Address Stack (CFAS) schematics.

This version of the shadow stack is intended to solve the problem of overhead imported by recursive functions. The maximum number of nested function calls

a shadow stack can handle is indicated by its depth; however, in cases such as recursive programming, a vast number of nested function calls are needed. In this situation, after the first recursion inside the function, the later return addresses would be the same as the first one. In CFAS, instead of repeating the storage of the same address many times, the existence of a recursive function is indicated by a single bit concatenated to the address in the shadow stack: the Least Significant Bit (LSB) is set to 1, to avoid more hardware overhead for the extra bit. This bit must always be equal to 0 for normal execution, to prevent unaligned address exceptions. Checking and verifying the return address must be done as follows: if the LSB of the return address on top of the shadow stack is equal to 0, it must be popped, and comparison with the actual return address value must be made. However, in the case of recursive programming (LSB = 1), popping from the shadow stack only occurs if the return address on top of the shadow stack is different from the actual return address.

The CFRS module functions almost identically to the previous one, with some differences related to the different context in which it is involved. The recursion tag is absent, and the two signals from the control unit are driven by the execution of two explicit new instructions, `CFPUSH` (*Control-Flow Push*) and `CFPOP` (*Control-Flow Pop*). These two instructions are behaving as a normal `SW` (Store Word) and `LW` (Load Word) from ISA specification [178], but they perform in parallel the same operation from/to CFRS. The shadow stack takes the argument of `CFPUSH`, and during the execution of `CFPOP`, the comparator checks between the output of the shadow stack and the instruction. Since CFRS operations operates a double store/load on normal and shadow stack, related instruction replace normal spill/fill instruction during the instrumentation. It is important to specify that `CFPUSH` and `CFPOP` only need to operate on stack-spilled registers during ISR execution, and not for the return address. In fact, when an interrupt comes, RISC-V specification mandates to save this into `MEPC`, which is privileged and internal to the processor [178]. Therefore, there is no need for enforcing ISR returns in this context.

As for forward branch protection, the CFLV module follows the classic label-based scheme already introduced by Abadi and used in the external-monitor implementation. The RTL view of the module is shown in Figure 6.5. Within the datapath, two registers of 20 bits each are reserved (i.e., the bitlength chosen for the labels, since in RISC-V, immediate values are on 20 bits at most). A new instruction called `CFES` (*Control-Flow Enforce Source*) allows its 20-bit immediate operand to be saved in the source register, while another new instruction called `CFED` (*Control-*

Flow Enforce Destination) does the same with the destination register. The two values are combined by modular sum (+), and the sum constitutes the 20-bit input (called *signature*) to the PLA, which produces a negative logical value if the label pair matches a valid branch according to static analysis, or positive if the opposite happens. Upon the arrival of the source label, a *timer* is activated, while it is stopped in the execution of the CFED. The signal indicating a control-flow violation is the logical OR between the terminal count of the timer and the output of the PLA. For proper implementation of the policies described in the previous Chapter, the CFES instruction is also responsible for disabling interrupts, while CFED is responsible for re-enabling them.

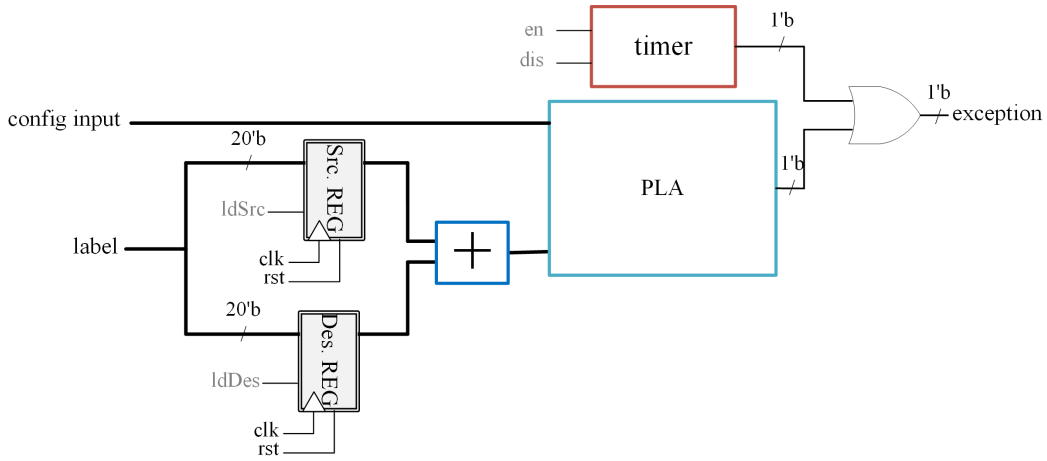


Figure 6.5 Control-Flow Logic Verifier (CFLV) schematics.

The timer value is set to only cover the execution of the branch instruction and the CFED instruction, so that after the protected branch, either a CFED is executed and the timer is stopped, or another instruction is executed and the timer reaches the limit, signaling an immediate violation. The check module is a logic designed efficiently for acting as a configurable *sum of products* (SOP) function, with a configuration depending on the application's CFG. By utilizing a new privileged instruction called CFLC (*Control-Flow Load Configuration*), it is possible to load from memory to one of the PLA configuration registers, deciding on PLA connections. Internally, the PLA is a network consisting of 8 horizontal lines and 40 vertical lines. The horizontal lines represent the products (AND), while the vertical lines represent the 20 bits of the input, each one also reported in its negated form, as per the classic PLA design [44]. All horizontal lines then are summed (OR) to produce the validity signal. Line cross connections are all controlled by an SRAM cell, which contains

a positive value if there is to be a connection at that crossing. It follows that the PLA is configured by 320 bits. For consistency with processor parallelism, these 320 bits are organized into 10 32-bit registers. Each of these is addressed by a CFLC instruction.

Since the number of OR functions cannot be unbounded in a PLA with a fixed number of rows, the instrumentation tool does not choose labels randomly, but it is fed with a list of signatures, as long as the number of allowed branches. Label generation is handled by a dedicated algorithm, which produces a set of signatures that is always within the 8 groups corresponding to the 8 products[11].

More details on the architecture and operation of the control-flow integrity modules added to AFTAB can be found in its Github repository[12].

In light of what presented, Table 6.7 lists the instructions that have been added to the RV32IM ISA base for handling the CFI modules within AFTAB.

Instruction	Type	opcode	funct3	Instruction Format
CFES	*U*	0x6B	/	CFES imm
CFED	*U*	0x7B	/	CFED imm
CFLC	*I*	0x2B	0x2	CFLC Cr, imm(Xs1)
CFPUSH	*S*	0x0B	0x2	CFPUSH Xs2, imm(SP)
CFPOP	*I*	0x5B	0x2	CFPOP Xd, imm(SP)

Table 6.7 CFI Extensions to RV32IM ISA.

Table 6.8 report the translation of PIPs and PEPs operated by the instrumentation tool.

[11] Details on the label generation algorithm at https://github.com/RHESGroup/prolepsis/blob/main/risc-v-cfi-tool/label_generator.py

[12] https://github.com/RHESGroup/aftab-cfi-pla

ToyRISC	RV32IM
PIP *label*, FBP ;forward branch instruction here	CFES LABEL ;forward branch instruction here
PEP *label*, FBP ;forward branch destination here	CFED LABEL ;forward branch destination here
PIP *label*, BBP ;call instruction here	;no instruction needed ;call instruction here
PIP *label*, FBP ;backward branch instruction here	;no instruction needed ;forward branch instruction here
;call instruction here PEP *label*, BBP	;call instruction here ;no instruction needed
;ISR entry point PIP $[sp-1]$, IHP	;ISR entry point CFPUSH Xn ;repeat this for any stack- ;spilled register substituting ; normal store
PEP $[sp-1]$, IHP IRET ;ISR exit point	CFPOP Xn ;repeat this for any stack- ;spilled register substituting ; normal load xRET ;ISR exit point

Table 6.8 RV32IM machine instruction translation for PIP and PEP to the internal PDP.

The protected execution environment is set up as follows. After the analysis and instrumentation tool has produced (1) a binary executable file with the modified `.text` section and (2) the list of permissible branches in the usual form (S,D), the latter is passed to the signature generation algorithm, which for each derives a valid signature, properly coupled with a valid source label and a valid destination label. The algorithm then also generates the 10 32-bit values needed to configure the PLA registers. All these data are then fed to the instrumentation algorithm, which uses them in the process. An `.init` section is added and programmed with a procedure that can use the `CFLC` instruction 10 times to properly configure the PLA accordingly before the main program starts.

Because they are integrated within the processor, the modules run at the same speed as the others in the datapath. The operation of none of these modules contributes overhead with respect to the execution times of the other instructions, i.e., general processor timing with no instrumentation are maintained. In particular, the operation of CFAS is run in parallel with respect to the call/return operations, and, in addition, it also does not imply any increase in execution time with respect to normal calls/returns. On the other hand, `CFES` and `CFED` instructions have a duration of 8 clock cycles (cc) each. An execution time of 7 cc has been designed for `CFPUSH` and `CFPOP`, but their use replaces that of the normal push and pop operations, which have equal latency. As far as the `CFLC` instruction is concerned, each of them has a latency of 14 cc, although their execution affects the execution environment only at bootstrap time, and not on recurringly. Unlike the previous external-monitor implementation, any window of time in which the attacker operates maliciously before the violation is as long as a single instruction, in particular as the verification instruction. For these reasons, no additional protection mechanism is provided on the execution of, e.g., the `.init` section, as was the case with the external monitor. In fact, there is no machine instruction that handles the resetting of the security modules, or the clock, since these are shared with the CPU. Even if the attacker manages to jump in the middle of the PLA configuration instructions, the timeout would be triggered immediately afterward, and an appropriate security procedure would recover the environment. On the contrary, the 10 32-bit words that make up the PLA configuration must be properly made unmodifiable. Assuming incorruptible code memory (cfr. **NWC** in Chapter 5), these can be saved directly in the `.init` section, and loaded by proper use of, e.g., the RISC-V `AUIPC` instruction for PC-relative loads.

6.3 Binary Analysis and Instrumentation

The architectures of the two monitors described above translate into practical reality the protection rules established by the theory presented in Chapter 5. Regardless of the internal or external presence of the PDP with respect to the processor, all policies are complied with in the same way in both cases. In fact, Tables 6.2, 6.3, 6.4, and 6.8 demonstrate that the same ToyRISC formal statements are being translated, with differences just with respect to two different practical architectures. Then again, the same instrumentation procedures have already been presented in formal terms in

Algorithms 1 and 2 in Chapter 5. Thus, it is possible to think to the PROLEPSIS framework as able to offer a unique tool for binary analysis and instrumentation of the firmware, eventually declined with respect to the ISA and protection instructions employed in the practical case.

In this Section, we present a software tool that implements into reality Algorithms 1 and 2, having as target binary applications written for ARMv7-M and RV32IM architectures. The tool is capable of automatically identifying tainted branches in an executable program, and then produce an instrumented version, with tainted branches protected against control-flow redirection attacks. The software is abstracted from the instrumentation code: it accepts a parametric set of protection instructions, provided by the user. In the following, the strategies adopted for implementation are discussed, as well as the organization of the code, the data structures involved, and the stages of the instrumentation process implemented by the tool: parsing of the disassembly listing, reconstruction of the disassembly code, retrieval of tainted branches, label generation, and code instrumentation. The implementation was first presented in a paper of 2021 [179].

Overview

In order to retrieve all possible tainted branch locations and destination, the first purpose of the PROLEPSIS instrumentation tool is to extract the application's control-flow graph. In this regard, the two considered instruction set architectures have some peculiarities that must be taken into account during the branch retrieval process:

- The presence of instructions that redirect the execution flow by changing the `PC` register value through operations that may involve tainted registers, which value may derive from memory locations potentially corrupted (this applies to both RISC-V and ARM);
- Unlike Intel-based architectures, the lack of proper `RET` machine instructions: by convention, ARM and RISC-V ISA perform the return with a branch instruction to the content of a register, namely *link register* (`LR`) in ARM and *return address* (`RA`) in RISC-V; or, by directly loading the return address in the `PC` register through a `POP` instruction or load (`LDR`) instruction;

- The application of multiple instructions sets inside the same binary file: executable files compiled against the ARMv7-M may contain Thumb II instructions [159], so with a mix of 16- and 32-bit instructions;

At the author's best knowledge, none of the major tools present in literature [148] [149] [150] is able to handle such features, for the reasons already discussed in Subsection 3.4.5. Hence the need to design a brand new tool for the PROLEPSIS framework.

The tool can be seen as divided into 5 different *stages*:

1. *Parsing*: it parses the file containing the disassembly of the code section (`.text`) and the disassembly of data sections (e.g., `.data`, `.rodata` and `.bss`), and are converted in two separate listing files;
2. *CFG Extraction*: the program is analyzed by applying a combination of static and dynamic analysis techniques in order to retrieve the CFG and obtain destinations of indirect branches;
3. *Edge Retrieval*: among indirect branches, tainted ones are identified, and stored in proper data structures with additional information for classification purposes;
4. *Label Generation*: from the edges, all the sources and destinations are associated to a given label, according to a given label generation policy;
5. *Instrumentation*: once the labels are available, the instrumentation code is applied to the locations previously identified in the code. Finally, all the sections are again merged in order to output a single instrumented Assembly file, containing the whole application and ready to be compiled again;

Figure 6.6 summarizes the 5 stages of the PROLEPSIS instrumentation tool.

The `CFG Extraction` and `Edge Retrieval` phases of the instrumentation process heavily rely on the reverse-engineering framework `Radare2` [180] and the Python module `r2-pipe` [181], that allows programmatic interaction with the framework. Radare is extensively used to extract control-flow graphs and retrieve multiple high-level information about components of the program (especially functions) and,

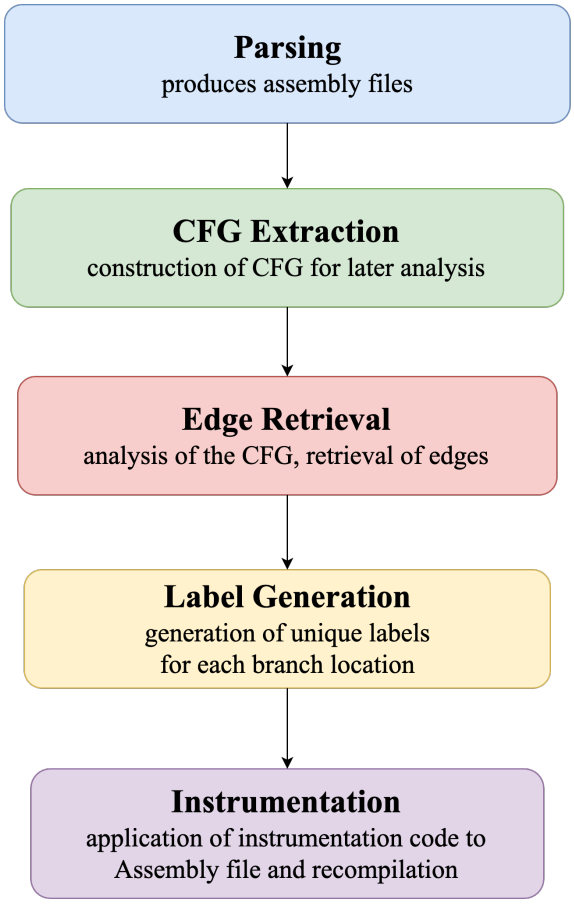


Figure 6.6 Stages of the PROLEPSIS analysis and instrumentation tool.

for each instruction, information about their type are provided, e.g., if it is a call instruction, a return instruction, or a simple branch instruction.

Both for ARM and RISC-V platforms, the scripts implementing the tool take as input:

1. the target *executable binary file*;
2. two *disassembly listings* retrieved from the binary file through `objdump`. One is the disassembly of the `.text` section, while the other one lists data sections, e.g., `.data`, `.bss` and `.rodata`;
3. the *instrumentation file*, containing the instrumentation code to be inserted in the instrumented version program, provided in the form of a JSON array.

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

1. an *instrumented Assembly source file*, retrieved from parsing the disassembly listings provided as input, and augmented with instrumentation code provided as input. Such a file is later compiled to get the final instrumented executable;
2. the *list of tainted branches*, retrieved during the *Edge Retrieval* phase. Edges are classified in terms of backward and forward edges, and for each of them, some additional information is provided, such as the *associated label*, or source/destination memory addresses;

Parsing

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

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

- the hexadecimal address of the instruction;
- the operational code of the instruction (opcode);
- the mnemonic representation of the instruction;
- a comment, useful for resolving labels and *PC-relative load operations*;

As an example, the disassembly listing of a function in ARMv7-M follows:

```
08002bec <setlocale>:
 8002bec:   b141        cbz r1, 8002c00 <setlocale+0x14>
 8002bee:   b510        push {r4, lr}
```

```
8002bf0:   460c        mov r4, r1
8002bf2:   4620        mov r0, r4
8002bf4:   490a        ldr r1, [pc, #40]   ; (8002c20 <
   setlocale+0x34>)
8002bf6:   f7fd fdd7   bl 80007a8 <strcmp>
8002bfa:   b918        cbnz r0, 8002c04 <setlocale+0x18>
8002bfc:   4809        ldr r0, [pc, #36]   ; (8002c24 <
   setlocale+0x38>)
8002bfe:   bd10        pop {r4, pc}
8002c00:   4808        ldr r0, [pc, #32]   ; (8002c24 <
   setlocale+0x38>)
8002c02:   4770        bx lr
8002c04:   4907        ldr r1, [pc, #28]   ; (8002c24 <
   setlocale+0x38>)
8002c06:   4620        mov r0, r4
8002c08:   f7fd fdce   bl 80007a8 <strcmp>
8002c0c:   2800        cmp r0, #0
8002c0e:   d0f5        beq 8002bfc <setlocale+0x10>
8002c10:   4905        ldr r1, [pc, #20]   ; (8002c28 <
   setlocale+0x3c>)
8002c12:   4620        mov r0, r4
8002c14:   f7fd fdc8   bl 80007a8 <strcmp>
8002c18:   2800        cmp r0, #0
8002c1a:   d0ef        beq 8002bfc <setlocale+0x10>
8002c1c:   2000        movs r0, #0
8002c1e:   bd10        pop {r4, pc}
8002c20:   200004f8    .word   0x200004f8
8002c24:   200004f4    .word   0x200004f4
8002c28:   2000046c    .word   0x20000610
```

From the `.text` disassembly listing, every row is parsed, and only the hexadecimal address and the mnemonic representation are extracted, along with the comment, as a string. Address and mnemonic instruction compose a entry of a Python dictionary, named `code`, where the key corresponds to the address and the value to the Assembly instruction. The comment is still kept for resolving later PC-relative load instructions. During the parsing, the direct branch instructions must be resolved. In order to do that, the comments provided by `objdump` are exploited. Taking the previ-

ous example of the `setlocale()` function, the instruction `bl 80007a8 <strcmp>` is processed and transformed into `bl strcmp`. Same happens for the instruction `beq 8002bfc <setlocale+0x10>`, that is transformed in `beq labN`, where `labN` constitutes a new label for the instruction at address `setlocale+0x10`.

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

- an hexadecimal address;
- data bytes;

Again, every data section is parsed in a way to produce a dictionary having memory addresses as keys and the corresponding values in memory as values. Depending on the processor architecture, the syntax may slightly change. In the case of ARM, the conversion looks like the one in Figure 6.7. Alongside data byte, the `objdump` produces tentative disassembly mnemonics, that are to be discarded.

```
20000610 <impure_data>:
20000610:   00000000    andeq   r0, r0, r0
20000614:   200008fc    strdcs  r0, [r0], -ip
20000618:   20000964    andcs   r0, r0, r4, ror #18
2000061c:   200009cc    andcs   r0, r0, ip, asr #19
20000620:   00000000    andeq   r0, r0, r0
```

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

```
impure_data:
    .word   0x00000000
    .word   0x200008fc
    .word   0x20000964
    .word   0x200009cc
    .word   0x00000000
```

(b) Assembly representation retrieved after parsing a `.data` fragment.

Figure 6.7 Data listing conversion within PROLEPSIS instrumentation tool.

One last step of the parsing phase is to resolve the PC-relative load instructions. To do that, the memory address is retrieved from the comment of the load instruction. Later, the dictionary representing data or code memory is accessed, depending on the address location, in order to retrieve the corresponding symbol or value (in case

the symbol is not defined). Finally, the PC-relative load is replaced with the symbol retrieved, or with the corresponding value.

The final conversion of the `setlocale()` function, previously taken as an example, can be observed below. As it is possible to notice, the `ldr r1, [pc, #40]` is a PC-relative load instruction that finally is converted with `ldr r1, =0x200004f8`. As well, `ldr r1, [pc, #20]` is resolved to `ldr r1, =impure_data`, as this is the symbol present at data address `0x20000610`. To correctly handle PC-relative load instructions, the tool inserts a `.ltorg` directive at the end of functions: this forces the assembler to place in that point a *literal pool*, such that after re-assembling, symbols can be again accessed through PC-relative loads, but with the correct offsets, possibly adjusted after instrumentation.

```
setlocale:
    cmp r1, #0
    beq lab443
    push {r4, lr}
    mov r4, r1
    mov r0, r4
    ldr r1, =0x200004f8
    bl  strcmp
    cmp r0, #0
    bne lab444
lab445: ldr r0, =0x200004f4
    pop {r4, pc}
lab443: ldr r0, =0x200004f4
    bx  lr
lab444: ldr r1, =0x200004f4
    mov r0, r4
    bl  strcmp
    cmp r0, #0
    beq lab445
    ldr r1, =impure_data
    mov r0, r4
    bl  strcmp
    cmp r0, #0
    beq lab445
    movs r0, #0
```

```
    pop {r4, pc}
    .ltorg
```

This process may suffer an important problem: some PC-relative resolved values may themselves indicate an *address*, which may change after the instrumentation process. As for data, this problem is solved by construction, as in embedded systems the data memory is divided from the code memory, and through the *linker script*, it is possible to indicate the starting address of the data memory depending on the memory map of the architecture. If the data has a starting address that is not immediately consecutive to the end of the code, but belongs to another sector, then after instrumentation, all data will continue to have the same location, even if the code increases due to instrumentation. Another issue relates the presence, within the code, of constant values indicating instruction addresses. This problem is remedied by replacing them with labels created from scratch, similarly to what happens with branch target references within the function (i.e., a proper `labN` is created).

CFG Extraction

In this phase, the executable binary is analyzed in order to retrieve the abstract representations of the program, i.e., its control-flow graph. This phase is mostly carried out directly by the reverse engineering framework *Radare2*. The framework applies several static and dynamic analysis techniques, and once finished, it allows to be queried in order to retrieve the result of the analysis. For example, it is possible to automatically retrieve the CFG of a function in a picture or in JSON format, more useful for scripting purposes.

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

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

As later described, the `codexrefs` property is particularly important in order to identify all the callers of a function. Every Python object inside the `codexrefs` field consists of the following properties:

- `addr`: provides the address of the control-flow transfer instruction (this can be a memory address belonging a different function in the code);
- `type`: the type of control-flow transfer. It can be `CALL` for identifying a call instruction, or `JUMP` for identifying simple branch instructions;
- `at`: the destination address of the control-flow transfer instruction at the corresponding address `addr` (this is a memory address belonging to the same function);

Another interesting object that can be extracted from Radare2 are Python objects representing instructions. For every instruction, Radare2 allows to retrieve its type, that can be `call` for direct call, `rcall` for indirect call to register, or `ret` for return instructions. This allows to use the same procedure across any CPU architecture.

Finally, Radare2 provides the targets of indirect call instructions. Given an instruction of type `rcall`, the list of targets is provided in the property of the object representing the indirect call instruction, named `refs`.

Edge Retrieval

This stage is the core of the script, where the information provided by Radare2 are exploited in order to extract the list of edges requiring protection. In order to retrieve tainted branches, it is necessary to first collect all the functions of the program provided by Radare2. For every function, the following operations are performed:

1. the list of instructions is retrieved and stored in a list;
2. the list of caller location for the current function is retrieved through a recursive algorithm, along with the return location for the callee;
3. from the list of instructions retrieved at Point 1, the tainted forward branch instructions are retrieved, along with their targets. Every edge is inserted in a dictionary having the source location as key and the set of destination locations as value;

4. from the list of instructions retrieved at Point 1, the tainted backward branch instructions are retrieved. For every pair of return instruction/destination instruction, a backward edge is added to the same dictionary of Point 3 (along with the caller instruction).

In order to define whether the return branch of a function is tainted or not, it is verified whether the return address is ever stored in the stack or any other writable memory region, that may be potentially corrupted. This check is dependent on the targeted ISA, as the name of the register holding the return address of course might be different.

Label Generation

During this stage, the *edges map* created during the Edge Retrieval phase is taken as input. All the memory addresses (either corresponding to a source or destination of an edge) from the map are extracted, possible duplicates are filtered out, and the remaining memory addresses are stored in a list. As a final step, the list of addresses is passed to a label generation algorithm that assigns a unique label to each memory address. A label corresponds to an integer that can be represented on a number of bits depending on the computer and the monitor architectures. Labels can be generated randomly as long as the monitor allows such a choice. For more elaborated architectures (e.g., PLA), a proper algorithm must choose them to have compatibility with the checker modules. Label generation always starts with the creation of a given number of *signatures* X, as unique identifiers for combinations of source and destination locations (S,D). X must be a function of the couple (S,D), and the reverse function is applied to X to get couples, taking into account that different signatures can have source or destination labels in common.

Instrumentation

During the last stage of the overall process, the Assembly code, retrieved during the Parsing phase, is modified by adding instrumentation instruction to the instrumentation points. First are instrumented the forward edges, and later the backward edges. Once the assembly code is augmented with the instrumentation code, the resultant Assembly code is written to a file. This file contains the entire application in a single

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

Figure 6.8 Forward and backward edges in JSON format.

Assembly source, with code and static data. The file is recompiled to get the final executable. Before that, possible setup procedures are added through an appended `.init` section or equivalent.

Since label couples are also used to feed the monitor, forward and backward edges in the edges map are converted to JSON objects and written to a file. In Figure 6.8, an example of such a file is shown.

6.4 Security Evaluation

After implementing monitor architectures and instrumentation algorithms in practice, the implementations need to be tested to quantify the validity of the proposal. Although from a formal point of view it has been proven that no control-flow attack that goes outside the CFG boundaries can be made, this needs to be tested in practice to confirm the theory. The evaluation of CFI solutions in quantitative terms is still a very open debate in the scientific literature. In fact, whether one can qualitatively evaluate a system in general, or set up custom applications that can prove an actual security level, the scientific landscape still lacks a standard and universally agreed metric to quantify the validity of an control-flow integrity solution from the point of view of its security.

The reference work described in [97] posed this problem, and attempted to provide a more quantitative assessment through the classification of the various solutions under certain levels of performance, including those presented in Section 3.4. While fundamental for a qualitative evaluation of a given solution, this proposal still fails in providing a single metric that can put the various techniques on a same level of comparison. On the contrary, it is relatively easy to make comparisons on the classic metrics by which an architectural proposal is evaluated: execution time added, evaluated on some standard benchmarks, or increase in code footprint, or increase in silicon and energy consumption, etc. Although standardization issues still exist for this type of evaluations as well, there are at least some universally accepted guidelines that give scientific strength to the results presented. All of these are in fact united by the execution of a precise *test routine* on the specific solution, architected to provide an objective output value. In this way, the same routine can be imported into another solution with appropriate changes, and the values can be put on the same level. From these best practices, one can extract the idea of doing the same for security assessments: outlining a testing procedure that is algorithmically clear, adaptable to all situations (as much as possible), and providing a precise metric that can be accepted by the reference community.

From a certain point onward, within the community, it was born the idea of devising benchmark applications that contained the most diverse case of vulnerabilities, to test how many of these do not affect the system, and based on this result, offer an accurate value. In 2003, Wilander and Kamkar stated that most protection techniques were missing around 50% of the various attack types [182], and they

contextually introduced a tool capable of doing a comparative evaluation on runtime buffer overflows. The program was designed to target 20 distinct attack combinations. The concept was expanded and enhanced in 2011, when the *Runtime Intrusion Prevention Evaluator* (RIPE) benchmark was published [183]. RIPE is a benchmark that simulates a variety of attacks based on buffer overflows in order to assess the platform's resistance. The back-end is developed in C, and the front-end is built in Python. The back-end is designed as a program allocating a series of buffers and then exploiting them via code injection or code reuse techniques, utilizing vulnerable C standard library functions as entry points. RIPE's Python front-end provides for the selection of 4 distinct memory areas, 16 target code pointers, 10 susceptible routines and 2 overflow approaches, for a total of 850 attack permutations. These include three forms of shellcode injection, Return-into-Libc, and Return-Oriented Programming. Since its publication, RIPE has been used on many occasions in the literature to test the effectiveness of protections against BOF implemented at software or hardware level, such as in binCFI [104], HCIC [129], HW-CDI [125], BBB-CFI [85] or FastCFI [143].

However, because it requires the presence of a shell on the target system in order to inject it, and because it hosts static binary code for injection or custom ROP chains, RIPE can only be compiled and executed on `i386`-compatible architecture systems that use the Linux Application Binary Interface (ABI). This is a limiting constraint since, while low-power systems can now handle light versions of Linux, most microcontroller systems do not (think of monitoring systems in industry, infrastructures, and even home automation). In most situations, the control software is built directly to execute on the bare-metal SoC, often depending on libraries known as *Real-Time Operating System* (RTOS), which are not equivalent to a traditional operating system in terms of consistency. Other than giving libraries for common functionality, the major goal here is simply to provide support for predictable scheduling of various processes. FreeRTOS [170], created by a collaboration of market-leading microcontroller firms, is a fully accessible example. Finally, an evaluator for such systems must make no assumptions about the underlying standalone OS, and must provide test cases that can be performed directly on the hardware once compiled. As it happens for Linux, RIPE is then restricted in only supporting computers that are compatible with the Intel x86-32 architecture. This architecture is rarely utilized in the embedded domain, as the ARM family dominates the market. As a result, the evaluator must employ particular attack codes (for example, shellcodes or ROP

chains) written directly in ARM machine code. A recent paper [184] described a version of RIPE that was evaluated for Raspbian OS, the operating system used on the Raspberry Pi platform[13]. The study offers data on the system's response once typical OS solutions are applied, similar to those of Ubuntu 16.04. However, once again, the work provides no support for platforms that lack a rich operating system and the ability to communicate with the Linux ABI and system calls. Furthermore, the publication does not disclose any accessible resources from which to obtain additional information about the tool.

Despite the constraints, the benefits introduced by RIPE are significant. Attacks are classified into very significant groups, and occur in a large enough number. Furthermore, the dual front-end/back-end structure is beneficial to usability and modularity, even if the two sections must be separated in an embedded device. The front-end must run on an external host system, such as a workstation linked to the chip through a protocol (e.g., UART). It may then execute the back-end, which is the real body of the evaluator operating as the chip software, and assess the results. As a consequence, the Python front-end needs no support on the system under test. In turn, he system does not need to support Python or multitasking, and can merely run the evaluator code.

On the basis of these premises, the PROLEPSIS framework proposes a new version of the security evaluator, adapting the original RIPE philosophy to microcontroller-based real-time systems. Its name is *Em-RIPE*, first presented in a paper in 2022 [185]. Em-RIPE is currently architected as a proof-of-concept for the ARM-based 32-bit microcontrollers, and has a back-end relying on FreeRTOS libraries. Em-RIPE now supports coverage evaluation against attacks on 3 different memory sections, 3 types of pointers, 5 injectable *libc* functions, using 3 different attack codes, for a total of 105 possible attack combinations.

In light of the above, the Em-RIPE testbed features a Python front-end and a back-end written in C. The front-end runs on a PC host, and is constituted by a script that iterates through all of the supported test attacks. The back-end, on the other hand, runs on the board, and consists of an attack generator that builds the attack payload and performs attacks on itself. Similarly to the original RIPE, the front-end composes all possible combinations inside the attack space, and then activates the program on the board with the appropriate parameters. The interaction is made possible through

[13] https://www.raspberrypi.com/

a UART connection, through which the front-end passes the execution trigger signal and gets from the back-end the results of all attack combinations, one by one.

The dimensions of the attack space are:

1. the *overflow location*;
2. the *target code pointer*;
3. the *vulnerable function*;
4. the *attack code*.

The result log is marked with one possible outcome for each test performed: `DONE` when the attack was successfully completed, `NOPE` when the attack encountered an error before running to completion, or `NOT POSSIBLE` when the attack is not realistically possible (e.g., attacking the return address on the BSS section).

The *overflow location* specifies the memory area containing the target buffer. Attacks on section stack, data, and BSS are all supported. To be as general as possible toward the reference domain, Em-RIPE is not supporting the heap section: not all microcontroller systems feature dynamic memory management, as it requires a non-negligible overhead in terms of consumption, speed, and additional components, which is not worth affording for the objectives of the majority of control firmware.

The address exploited by the specific attack is represented by the *target code pointer*. It directs the control flow to the right location in order to activate the payload. Em-RIPE supports attacks on:

- *Function pointers*: variables containing the address of a function, which can be called dynamically;
- *Function return addresses*: the return addresses pushed on top of the stack, used for jumping back to the caller function as the callee terminates;
- *Structures with function pointers*: function pointers being part of a C structure (`struct`) and laying adjacent to a buffer.

Function return addresses are obviously stack-specific: they can only be utilized as a target code pointer when the selected attack location is the stack. Instead, all other targets can be present in any section.

Em-RIPE employs five *vulnerable library functions* as attack entries: `strcpy`, `strncpy`, `memcpy`, `memmove`, and `homebrew`, a loop-based counterpart of `memcpy` first developed in the prior version of RIPE. Such *libc* string functions allows for transfer of a string from a source buffer to a destination buffer, without doing any boundary checks. Then, they are used to cause the target buffer overflow. Instead, `n`-versions of C string methods like `strncpy` require the target buffer size as parameter. The specification of this parameter is however upon the developer. The size is usually computed dynamically. As a result, an inaccuracy in the size computation can defeat the limit check provided by those functions.

Finally, the *attack code* provides the payload that an intruder may wish to execute via a buffer overflow vulnerability. The purpose here is to execute a code that just sends a clear indication that the vulnerability has been successfully exploited. Differently from the original version of RIPE, Em-RIPE uses an attack payload that simply relies on a supervisor call (`svc`), which is featured as a machine instruction directly in the ARM ISA. When the `svc` is successfully run, it executes the associated handler, which is responsible for writing to the UART an identification that permits the front-end to record the success of the attack (`DONE`). If the attack fails, the Em-RIPE code continues with its normal flow, writing a negative identifier (`NOPE`) to the UART port.

Em-RIPE actually features 3 SVC-based attack code variants:

- *Injected SVC*: it is a code injection attack where the payload is only composed of a `svc` instruction;
- *Injected SVC with NOP sled*: the `svc` is preceded by a number of `nop` (No-Operation) instructions in the payload. This alternative version is most likely used by attackers, since it increases their possibility to reduce the program flow within the injected code. As well, this second attack code is useful in evaluating countermeasures based on recognizing common patterns [186], such as `nop` sleds are;
- *Reused SVC*: here, there is no code injection, and the target code pointer is overrun with the address of an existing *gadget* in memory, containing the `svc` instruction of interest. This gadget is artificially inserted in the back-end code, the execution cannot reach it in any other way. Such an attack code can be

used for evaluating the specific resistance against code reuse attacks, such as Return-into-Libc and Return-Oriented Programming.

In two cases out of the three described, the payload has format like the following:

```
(NOP sled +)
attack code +
padding bytes +
address to payload start
```

where the last element is the location of the pointer to the code to be corrupted. In the third case, instead, the gadget address is just injected, with a padding needed to align it with the target pointer.

The Em-RIPE back-end is not provided as standalone firmware, with the whole source code available, but it is instead configured as a FreeRTOS application. In fact, this gives abstraction from a plethora of target-specific low-level features, which may lead the evaluator to lose generality. FreeRTOS is designed to enable Hardware Abstraction Layer (HAL) services from leading ARM-based microcontroller families, such as STMicroelectronics, NXP, and FreeScale. It is possible to download demo programs for each of the available devices and merge Em-RIPE sources into the files for the specified target from the public GitHub repository of FreeRTOS[14]. The user pays for the effort of modifying configuration-specific features such as the starting procedure and the linker script for compilation, but he gains significant modularity and portability in exchange. Furthermore, rather of a full-fledged operating system with its own life, FreeRTOS may be regarded as a set of libraries around the custom program to be executed on the board. As a result, the evaluator must not presume any form of pre-configured underlying system software that imposes usability limits, as stated above. More information about the evaluator's architecture may be obtained online. In fact, unlike prior works such as [184], Em-RIPE is accessible for public download on GitHub[15], under a free-use license.

Em-RIPE has been used to assess the security coverage of the solution implemented for ARMv7-M with the parallel FPGA described in Section 6.1, so to give experimental consistency to findings presented in Section 5.4. Our technique has

[14] https://github.com/FreeRTOS/FreeRTOS/tree/main/FreeRTOS/Demo
[15] https://github.com/RHESGroup/embedded-ripe

been compared with other compiler-based reference solution available for the ARM Cortex-M4, namely *stack canary* and *Data Execution Prevention*. The compiler used for the experiments was ARM GCC version 9 [187]. The setups tested are incremental and range from the version with no protections (i.e., with executable data and no stack guard), to one with both guards employed. Their results are finally compared with the coverage obtained by our technique.

The `-protect_stack` compilation flag was used for the evaluation, which adds a guard variable to the stack between a function's vulnerable local buffers and its stacked return address. Buffers are deemed vulnerable in a coarse sense by this protection, i.e., when they have automatic storage duratio,n or are of type `char`. The compiler saves a global reference copy of the guard value to ensure that the value on the stack has not changed before returning to the caller. If the values are changed, a buffer overflow attempt is assumed, and the routine `__stack_chk_fail()` is invoked. The programmer selects the content of this procedure. The guard's value is stored in the global variable `void* __stack_chk_guard`, which should be set to an appropriate value, such as a strong random integer.

The second configuration employs the XN (eXecute-Never) bit feature, which prohibits hardware from directly executing bytes from given memory areas. Such protection prevents the attacker from injecting arbitrary code. For our assessment, we set XN options with suitable flags on the linker script (`.ld` file) supplied to the compiler, identifying data-containing areas as non executable. Actually, data execution prevention is already enabled by default in the compiler. As a result, we had to specify the contrary, i.e., execution from data sections was explicitly permitted when we wanted to examine a vulnerable setup.

To conduct the security tests, the external monitor environment was mounted on the SE*cube*™ device[16], manufactured by Blu5 Group. The chip is actually a SiP (System-in-Package), where three different components are stacked within the same silicon die:

- The STM32F4 microcontroller referred to in Section 6.1;
- The MachX02 FPGA referred to in Section 6.1;
- An EAL5+ certified smart card, useless for the specific case;

[16] https://www.secube.blu5group.com/

The choice was made mainly for the convenience of having the FPGA already connected to the CPU, without the need to recreate the parallel bus via *jumpers*. Pin connection is according to the schematic already presented in Figure 6.1.

Table 6.9 displays the results of our evaluator measured on 5 distinct combinations of the configurations mentioned above, from the most vulnerable to the most protected. Results are given with respect to he 105 successful attack combinations: while there are in theory $3 \times 3 \times 5 \times 3 = 135$ possible combinations, the run gives `NOT POSSIBLE` in 30 cases, i.e., when the return address is taken as the target pointer in BSS and data sections, which is not possible. As a result, while the stack can contain all $3 \times 5 \times 3$ combinations, BSS and data can only have $2 \times 5 \times 3$ of them, for a total of 105 potential attacks. As can be seen, using the stack protector only provides the least protection, resulting in failure in only 14% of the examined attacks. However, the rate of unsuccessful attempts as a result of XN bit application solely is substantially greater (67%), since code injection attacks are more common in the examined attack set. Finally, combining both compiler-based methods results in 81% coverage against BOF attacks. The remaining 19% successful attacks represent code-reuse exploits with targets other than the stack. This last amount of attacks is prevented by our technique, which offers a practical proof to the theory by granting 100% of coverage.

Setup	**Overall Effectiveness**	**Successful Attacks**	**Failed Attacks**
Baseline (no protection)	**0,00%**	105 (100,00%)	0 (0,00%)
Stack guard only	**14,28%**	90 (85,71%)	15 (14,28%)
XN enabled only	**66,67%**	35 (33,33%)	70 (66,67%)
XN enabled + stack guard	**80,95%**	20 (19,05%)	85 (80,95%)
PROLEPSIS	**100,00%**	0 (0,00%)	105 (100,00%)

Table 6.9 Security evaluation of PROLEPSIS using Em-RIPE's 105 attack forms, compared to common compiler-based protections.

6.5 Performance Evaluation

This Section reports data extracted from the performance evaluation of PROLEPSIS with respect to the two implementations presented. Traditionally, in the benchmark literature for control-flow integrity, the metrics chosen for such an evaluation are in almost all cases the following three:

- *Runtime overhead*: the additional time taken by a set of standard benchmark applications to execute on the protected platform is reported, compared to the execution time on the same platform without protection;
- *Size overhead*: the amount of additional memory that is required to implement instrumentations in the program code is reported, again with reference to a set of standard benchmark applications;
- *Resource overhead*: the additional hardware cost of protection compared to an unprotected execution environment is reported; for example, in an ASIC solution, the additional consumption of gates and flip-flops is reported, or, in an FPGA solution, the occupancy of Look-Up Tables (LUTs) and global storage cells caused by the implementation is reported.

As a set of benchmark applications, Embench-IoT[17] was chosen. It is an open project for benchmarking deeply embedded systems, and it is largely derived from the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [169]. The benchmarks do not assume the presence of any operating system, require a minimal library system and no standard output. The suite is composed of 22 programs, all designed to occupy no more than 64 KB for code and data, respectively. Used algorithms are classic benchmarking algorithms (e.g., CRC32, matrix multiplication, AES, SHA, block merge sort, MD5, etc.), whose execution times are normalized to last a few seconds, and normalization is accomplished by a number of loops that depends on the frequency of the platform on which they are tested. Indeed, the frequency value is to be set in a macro defined for each benchmark, based on which the number of loops is decided. The compilation system is highly customizable, allowing the user to choose the compiler to be used, the reference architecture, and in some cases even the board and chip, if they are among those naturally supported.

To measure size, the reference to be taken is the size of the `.text` section. Runtime, on the other hand, is measured as the elapsed time between the execution of the `start_trigger()` function and the execution of `stop_trigger()`. The code for these functions is customizable and allows internal hardware timers to be triggered. Both measurements can be operated automatically thanks to the presence of special Python scripts, which, after compiling all the benchmarks, provide the code size and runtime.

[17] https://github.com/embench/embench-iot

In particular, for the latter, there is a special support that is based on a feature of ARM Cortex-M4, namely the presence of the `DWT_CNT` register, specifically designed for the measurement of execution machine cycles. The register is mapped to address `0xE0001004` in the core map, and it automatically increments at each machine cycle (i.e., clock cycle), thus allowing a difference between the instant of execution of `start_trigger()` and the instant of execution of `stop_trigger()` to be calculated. After properly flashing the board with the benchmarks, connecting it to a workstation, and starting debug execution, the Embench-IoT automated procedure opens an instance of `gdb-multiarch` (GNU Debugger for Multi-Architecture)[18] on the workstation, and runs the following commands:

```
file benchmark_name.elf
target extended-remote :4242
load
delete breakpoints
break start_trigger
break stop_trigger
continue
print /u *0xe0001004
continue
print /u *0xe0001004
quit
```

This allows execution start and end times to be displayed on the screen and the execution time to be derived.

For performance evaluation of the external-monitor implementation, the Nucleo F429ZI board from ST Microelectronics, featuring an STM32F429 as processor, was chosen. It was not necessary to create any connection with the external FPGA via jumpers, since security was not the subject of the measurement. All 22 benchmarks were compiled and linked via the GNU Arm Embedded Toolchain 10.3-2021.10, installed on Ubuntu 18.04. The flags used for compilation are:

```
-g3 -O3 -MMD -MP -mthumb --specs=nosys.specs -mcpu=cortex-
   m4 -mpure-code -nostartfiles -Wa,-mimplicit-it=thumb -
   fno-data-sections -fno-function-sections
```

[18] https://www.sourceware.org/gdb/documentation/

The maximum degree of optimization was chosen for obvious code space reasons, and `-g3` grants maximum presence of debugging symbols, useful for later debugged execution. `-nostartfiles` was employed to prevent linking of startup functions provided by the standard library. Instead, all benchmarks were linked with a custom initialization procedure placed in the `.init` section. `-mpure-code` is aimed at avoiding the proliferation of literal pools along the code, by forcing constants to be split into immediates directly encoded in the instructions. This reduces the effort made by the instrumentation tool in resolving PC-relative loadings. `-fno-data-sections` and `-fno-function-sections` are used to avoid the inclusion of unreferenced symbols in the executable image. The obtained executable files, in addition to being debugged according to the procedures described above, were processed by the instrumentation tool described in Section 6.3 to get instrumented versions according to the additions in Tables 6.2, 6.3, and 6.4. The instrumented executables then followed the same execution process. More details can be found in the Github repository containing the material used for the experiments[19].

Table 6.10 shows the total number of branches (direct or indirect, tainted or secure, forward or backward) identified for each benchmark in the case of the external monitor, with the number of branches requiring PDP intervention (*protected branches*) alongside. These are all tainted branches, plus branches before which the PDP is informed about the intended return site (see PIP-B in Table 5.3). The differences are due to the nature of the benchmarks themselves. The *protection rate* is given by the percentage of branches that are augmented with at least one PIP/PEP. The geometric mean of this value is 10.74%. This means that, in the benchmark set, only 1 branch out of 10 needs PDP intervention. This result is particularly significant when viewed in relation to the whole framework: PROLEPSIS offers protection policies that are limited to a narrow subset of the branches of an application, i.e., *only those truly risking to be corrupted*, avoiding redundant checks in a context where each added operation can have a critical cost. Thus, it is empirically shown that only by enforcing about 10% of the branches, complete security is achieved from the CFI point of view.

Table 6.11 shows the size of the `.text` section before and after the instrumentation process, in bytes (B). Differences between the benchmarks, sometimes with even very large factors (e.g., 60x between primecount and qrduino) are due to the

[19] https://github.com/RHESGroup/arm-nucleo-prolepsis

Benchmark	Total Branches	Protected Branches	Protection Rate
aha-mont64	66	8	12.12%
crc32	1405	161	11.46%
cubic	1095	384	35.07%
edn	115	13	11.30%
huffbench	1677	195	11.63%
matmult-int	101	10	9.90%
md5sum	1420	166	11.69%
minver	565	45	7.96%
nbody	487	158	32.44%
nettle-aes	95	21	22.11%
nettle-sha256	116	25	21.55%
nsichneu	792	3	0.38%
picojpeg	968	123	12.71%
primecount	37	5	13.51%
qrduino	1772	193	10.89%
sglib-combined	2476	259	10.46%
slre	315	31	9.84%
st	360	114	31.67%
statemate	371	8	2.16%
tarfind	1430	166	11.61%
ud	134	10	7.46%
wikisort	2185	265	12.13%

Table 6.10 Protection rates in the external monitor implementation measured over Embench-IoT.

library functions that each benchmark need to use. Some of them do not even require a library function. Others, which make, e.g., extensive use of mathematical or string-processing functions, have larger codebases. The geometric mean is 17.88%, which translates to an increase of about 1.6 KB on codebase size, always in geometric mean. The target device has a Flash memory of 2 MB for hosting the code, but only 1 MB is usable (the other is reserved to store static application data, to be copied to RAM at startup). The size is not so different in other architectures in the reference domain. In conclusion, we are talking about an additional +1% on the code memory occupancy. This contains all instrumentation for all protected branches, according to Tables 6.2, 6.3, and 6.4.

On the other hand, Table 6.12 reports the execution times before and after the application of protection. This is the difference between the two functions reading the `DWT_CNT` counter, first recorded as it appears in number of cycles, and then converted to milliseconds by multiplying for the device activity frequency, i.e., 168 MHz. Here, the differences between the benchmarks trace not only the nature of

Benchmark	**Original** `.text` **size**	`.text` **size with protection**	**Increase**
aha-mont64	1896 B	2096 B	+10.55%
crc32	16080 B	20504 B	+27.51%
cubic	15580 B	26424 B	+69.60%
edn	3076 B	3408 B	+10.79%
huffbench	18488 B	23836 B	+28.93%
matmult-int	1788 B	2060 B	+15.21%
md5sum	16528 B	21072 B	+27.49%
minver	7664 B	8396 B	+9.55%
nbody	8088 B	12552 B	+55.19%
nettle-aes	2924 B	3512 B	+20.11%
nettle-sha256	4936 B	5648 B	+14.42%
nsichneu	14128 B	14228 B	+0.71%
picojpeg	13932 B	17324 B	+24.35%
primecount	552 B	692 B	+25.36%
qrduino	29240 B	34516 B	+18.04%
sglib-combined	25336 B	32576 B	+28.58%
slre	3944 B	4756 B	+20.59%
st	6200 B	9396 B	+51.55%
statemate	5060 B	5380 B	+6.32%
tarfind	16464 B	21036 B	+27.77%
ud	3164 B	3436 B	+8.60%
wikisort	29536 B	36724 B	+24.34%

Table 6.11 Code size in the external monitor implementation measured over Embench-IoT, before and after the instrumentation process.

the code, but also the number and size of loops. It can be clearly distinguish how the number of protected branches contributes to the overhead (e.g., think of cubic, but also at the other benchmarks that have more than 100 protected branches). Any mismatch from this fashion can be caused by the presence of protected branches in less intensively looped code areas. Some seemingly inexplicable results, such as the negligible overheads of crc32 and primecount compared to their code size increase, can only be explained by the presence of massive instrumentation in non-looping code areas. The geometric mean of all increases is 1.10%.

Table 6.13 reports data from the synthesis of the monitor on the MachXO2 FPGA. The synthesis was done using Lattice Diamond 3.11[20] on Ubuntu 18.04. The synthesized version of the monitor was used for the security evaluation described in Section 6.4, and includes 1024 entries for the Secure Label Stack (SLS) and 512 entries for the Secure Register Stack (SRS). Plus, 8192 entries are reserved for the Secure Edge Table (SET). In other words, 13-bit labels have been chosen for the

[20] https://www.latticesemi.com/latticediamond

Benchmark	Original runtime	Runtime with protection	Increase
aha-mont64	3664.21 ms	3732.74 ms	+1.87%
crc32	4179.86 ms	4179.86 ms	+ < 0.01%
cubic	3942.53 ms	5033.19 ms	+27.66%
edn	3569.03 ms	3594.35 ms	+0.71%
huffbench	3458.49 ms	3497.81 ms	+1.14%
matmult-int	4103.72 ms	4436.74 ms	+8.12%
md5sum	2424.54 ms	2430.30 ms	+0.24%
minver	2628.31 ms	3191.60 ms	+21.43%
nbody	1902.07 ms	2560.36 ms	+34.61%
nettle-aes	3958.17 ms	4000.75 ms	+1.08%
nettle-sha256	2818.47 ms	2941.96 ms	+4.38%
nsichneu	3850.32 ms	3850.32 ms	+ < 0.01%
picojpeg	4175.55 ms	4633.86 ms	+10.98%
primecount	4227.04 ms	4227.08 ms	+ < 0.01%
qrduino	3783.53 ms	3881.63 ms	+2.59%
sglib-combined	3524.12 ms	4134.03 ms	+17.31%
slre	3586.32 ms	4286.32 ms	+19.52%
st	4060.78 ms	5498.60 ms	+35.41%
statemate	1563.24 ms	1751.78 ms	+12.06%
tarfind	1490.50 ms	1569.12 ms	+5.27%
ud	1316.81 ms	1377.41 ms	+4.60%
wikisort	1650.48 ms	1933.93 ms	+17.17%

Table 6.12 Runtime in the external monitor implementation measured over Embench-IoT, before and after the instrumentation process.

test. On the other hand, the size of SRS and SLS is actually the same, but given the 16-bit parallelism of communication, the storage capacity of 32-bit registers is half that for 13-bit labels. The size of SET was decided based on the requirements of the security evaluation in the previous Section, and in any case, to accommodate a number of labels that was a good compromise between the security provided (in terms of aliasing avoidance) and limitations given by parallelism. For SRS and SLS, the choice of 1024 entries is dictated by the fact that this is the size in bytes of a Block RAM: these blocks are in fact 1 KB blocks accessed with a 10-bit address. Smaller sizes would cause equal block consumption. 2 blocks are therefore reserved for each shadow stack, used with parallel aligned access. Halving the number of entries to use half the storage was not considered an acceptable choice in this case, since the memories are single-port, and this would have meant doubling the memory access time, which is not feasible for obvious security reasons. The utilization of LUTs and registers, totally aimed at implementing control logic, temporary registers, and the timer, is little above 2% in both cases. On the contrary, RAM utilization goes to almost 70% of the available resource, as it must accommodate all the three

storage elements described above. In any case, memory dimensions must be related to the very small capabilities of the FPGA, which was chosen right for its small size, resulting in equally small power consumption.

Resource	**Available**	**Used**	**Utilization**
LUT	6864	128	2.62%
Registers	7209	168	2.33%
Block RAM	26 KB	18 KB	69.23%

Table 6.13 Lattice MachXO2 FPGA resource usage in the external monitor implementation.

Finally, Table X shows the summary of the solution in terms of overhead on size and runtime compared to 5 other benchmark solutions in the state of the art for control-flow integrity of ARM microcontrollers. For each, the microcontroller used for implementation, the benchmark used for evaluation, the experimental data, and the granularity of the solution are reported. For Camouflage and TZmCFI, the amount of overhead on the code is not reported. Our solution has a higher impact than reported for both RECFISH and CFI CaRE. However, the latter is coarse-grained, and also has a huge overhead with respect to execution time. Our solution results to be the best of these 5 in terms of granularity and joint size and runtime overhead.

Solution	**Processor**	**Benchmark**	**Size**	**Runtime**	**Granularity**
Kage	ARM Cortex-M4	CoreMark	+49.80%	+5.20%	coarse
RECFISH	ARM Cortex-R4F	CoreMark	+9.41%	+30.00%	fine
TZmCFI	ARM Cortex-M33	CoreMark	n/a	+21.00%	coarse
CFI CaRE	ARM Cortex-M33	Dhrystone	+15.00%	+513.00%	coarse
Camouflage	ARM Cortex-A53	Lmbench	n/a	20.00%	fine
PROLEPSIS	**ARM Cortex-M4**	**Embench-IoT**	**+17.88%**	**+1.10%**	**fine**

Table 6.14 State of the art comparison of CFI solutions for ARM microcontrollers.

Let us now analyze the same data for the internal-monitor implementation. Here, the AFTAB processor with security extensions was first synthesized using Vivado ML 2022.2[21] on Windows 10, and then a post-synthesis simulation was performed using QuestaSim Prime 2021.3.2[22] on CentOS version 7. All 22 benchmarks were compiled and linked using RISC-V GNU Compiler Toolchain version 11.1[23], used on Ubuntu 18.04. The flags used for compilation are:

[21] https://www.xilinx.com/products/design-tools/vivado.html
[22] https://eda.sw.siemens.com/en-US/ic/questa/
[23] https://github.com/riscv-collab/riscv-gnu-toolchain

```
-O3 -nostartfiles -fno-data-sections -fno-function-
   sections
```

Choice of compilation flags recalls the one already given for the external monitor case. `-g3` was not employed because there was no need for the symbols for debugging. In contrast, the `-mpure-code` option is not considered because compilation already avoids the use of PC-relative loads if a starting address for the constant data is given via the linker script. In other words, unlike the ARM compiler, the RISC-V compiler does not produce literal pools. `-nostartfiles` has the same task of preventing the linking of the standard startup library, since the benchmarks are linked to a custom procedure placed in the `.init` section. The obtained executables were first simulated to collect their execution time, then instrumented as reported in Section 6.3 and Table 6.2, and then again measured by simulation.

In particular, the used compilation and simulation environment is an adaptation of the one developed for the PULPino platform[24], i.e., based on heavy use of CMake. After being compiled, the executables are processed by a Python toolchain, that transforms them first into binary files and then into Memory Initialization Files (.mif), which are read as input directly from the VHDL files implementing the memory. AFTAB's memory was made large enough to hold all the benchmarks, i.e., 128 KB, divided into 64 KB for code and 64 KB for data. More details about the used simulation environment can be found in the Github repository containing the modified core and all the material used for the experiments[25].

Table 6.15 shows the account of total branches identified within the 22 benchmarks, along with the number of protected branches and protection rate. As for total branches, there is a high consistency with the data reported in Table 6.10 for ARM, with maxima located in the same benchmarks. The numbers are obviously not equal due to the use of a different compiler, which internally organizes machine code differently, and different libraries. More differences are found in the protection rate, which is more consistent in the cases of e.g., nbody, nettle-aes, and nettle-sha256, but less on e.g., cubic, which recorded the highest protection rate in ARM (35%) and here is just above 3%. The protection rate of st also drops significantly from 31% to 8%. This is due to the different implementation of the *math libraries*, of which these two benchmarks make extensive use. The implementation for ARMv7-M evidently

[24] https://github.com/pulp-platform/pulpino
[25] https://github.com/RHESGroup/aftab-cfi-pla

has a "worse" management of registers that are arguments for indirect branches with respect to RV32IM. Curious is the case of nsichneu, which does not even have a branch at risk. This is consistent with the case of ARM, where it stopped at 0.38%. Overall, the geometric mean of the protection rate is about -3% with respect to ARM, and stands at 7.55%. Compared with ARMv7-M, RV32IM can count on a double amount of general-purpose registers, which allows it to use less memory to make room for local data, and thus fewer tainted branches are created.

Benchmark	**Total Branches**	**Protected Branches**	**Protection Rate**
aha-mont64	96	2	2.08%
crc32	1329	78	5.87%
cubic	2643	96	3.63%
edn	92	10	10.87%
huffbench	1492	87	5.83%
matmult-int	82	23	28.05%
md5sum	1394	85	6.10%
minver	954	179	18.76%
nbody	1201	198	16.49%
nettle-aes	158	19	12.03%
nettle-sha256	157	43	27.39%
nsichneu	1075	0	0.00%
picojpeg	889	113	12.71%
primecount	38	3	7.89%
qrduino	1821	107	5.88%
sglib-combined	2406	95	3.95%
slre	416	25	6.01%
st	1126	99	8.79%
statemate	373	11	2.95%
tarfind	1355	81	5.98%
ud	117	4	3.42%
wikisort	2866	213	7.43%

Table 6.15 Protection rates in the internal monitor implementation measured over Embench-IoT.

Table 6.16 shows the differences between the sizes of the `.text` sections before and after instrumentation. From the point of view of the absolute sizes of the original benchmarks, the data are almost always consistent with ARM, with maxima located at the same points (in particular, note wikisort and qrduino). The most significant differences are again in cubic and st, due to different implementation of math library functions. The maximum size is larger than in the previous case, as a result of the fact that ARMv7-M can rely on the Thumb II extension that includes 16-bit compressed instructions. On the other hand, for RV32IM all instructions are encoded on 32 bits. In terms of overhead, the data is significantly lower than for ARM, and the

main reason is the lower number of bytes required to add instrumentations (i.e., 1 instruction versus a group of 6 instructions), also combined with the slightly lower protection rate. Combination of these two reasons bring down the geometric mean of the increase to 0.63%. It is noteworthy that there are as many as 6 cases where the increment is zero[26]. In these benchmarks, the number of protected branches is not always zero (nsichneu is the only case), although it remains very low. The reason is that there is no explicit instrumentation for backward branches in this implementation, i.e., PIP-B and PEP-B do not involve any addition of instructions. As a result, in cases where there are no tainted forward branches, the overhead on the codebase is 0. However, just as in the case of ARM, the maxima of the increments on the size are placed on the protection maxima (i.e., cubic in ARM, and matmult-int in RISC-V).

Benchmark	**Original `.text` size**	**`.text` size with protection**	**Increase**
aha-mont64	3056	3056	+0.00%
crc32	22968	23280	+1.36%
cubic	49624	50388	+1.54%
edn	4016	4016	+0.00%
huffbench	27768	28080	+1.12%
matmult-int	1904	1976	+3.78%
md5sum	23752	24064	+1.31%
minver	15680	15688	+0.05%
nbody	15244	15260	+0.10%
nettle-aes	7468	7508	+0.54%
nettle-sha256	11544	11656	+0.97%
nsichneu	19632	19632	+0.00%
picojpeg	24148	24212	+0.27%
primecount	360	360	+0.00%
qrduino	42460	42724	+0.62%
sglib-combined	37572	37972	+1.06%
slre	5524	5524	+0.00%
st	13156	13172	+0.12%
statemate	7948	8020	+0.91%
tarfind	23432	23592	+0.68%
ud	3820	3820	+0.00%
wikisort	47196	47828	+1.34%

Table 6.16 Code size in the internal monitor implementation measured over Embench-IoT, before and after the instrumentation process.

Table 6.17 shows the execution times of the benchmarks and the relative overhead. The first thing to note is the difference of a factor $\sim$10 with the ARM case. The reason

[26]Geometric mean is always computed excluding null values.

for this is the combination of a reduced frequency and an architectural difference. In fact, AFTAB is synthesized to run at 100 MHz, and is not a pipelined processor, unlike ARM Cortex-M4. Instruction execution stages occur sequentially, taking about 5 clock cycle per instruction on average[27]. To proof the above factor, let us suppose that a program consists of 100 instructions. In a pipelined processor, the 100 instructions will take about 100 clock cycles to execute at the end. If the frequency is 168 MHz, this means that the program will run in about 595 ns. The same program, running on a sequential processor with an average of 5 clock cycles per instruction, will execute in 500 clock strokes, corresponding to 5000 ns if a frequency of 100 MHz is considered. This is how the difference in execution and the factor mentioned above is explained.

Nevertheless, it is equally easy to note the consistency in execution times, with the maximum always on cubic and the minimums on the last 4 benchmarks. Where there had been zero overhead on code, this is also repeated in the overhead on runtime. Interesting results to discuss are those of matmult-int and statemate, which despite having overhead on size and 23 and 11 protected branches, respectively, they record zero overhead on runtime, adding to the other 6 benchmarks with null cost. This should not be surprising, because the protected branches count also includes branches that, although protected, require no instrumentation in this implementation, such as backward branches and their associated call branches. Moreover, examples of nsichneu and primecount maintain some consistency with the previous case, in which they were under 0.01% increment. However, significant deviations in the opposite direction from the size overhead are found in tarfind and wikisort. The reason is due to the heavy presence of tainted forward branches in the benchmark loops, which, for example, use function pointers as parameters to other functions. The difference with the previous case is due to a "better" management of the registers dealing with indirect branches by the ARM compiler. Ultimately, the geometric mean of increments stands at 0.57%. In case there was any need, this confirms the absolute convenience in terms of overhead of a solution with an internal monitor and integrated with the ISA, compared to one with an external monitor, to be managed through a dedicated interface.

Finally, Table 6.18 reports data from synthesis of AFTAB with its security extensions, and relates it to the base version of the processor and the overall occu-

[27]More details at https://github.com/RHESGroup/aftab/blob/master/doc/aftab_user_manual.pdf.

Benchmark	Original runtime	Runtime with protection	Increase
aha-mont64	37559.96 ms	37559.96 ms	+0.00%
crc32	41097.56 ms	41102.36 ms	+0.01%
cubic	75481.45 ms	80703.95 ms	+6.92%
edn	51678.94 ms	51678.94 ms	+0.00%
huffbench	20877.64 ms	20900.11 ms	+0.11%
matmult-int	54461.27 ms	54461.27 ms	+0.00%
md5sum	17058.16 ms	17149.49 ms	+0.54%
minver	26231.30 ms	27119.97 ms	+3.39%
nbody	31894.18 ms	31898.18 ms	+0.01%
nettle-aes	40327.55 ms	40327.55 ms	+0.00%
nettle-sha256	34526.04 ms	36046.10 ms	+4.40%
nsichneu	25270.10 ms	25270.10 ms	+0.00%
picojpeg	36350.93 ms	37993.36 ms	+4.52%
primecount	22015.04 ms	22015.04 ms	+0.00%
qrduino	27738.20 ms	27848.84 ms	+0.40%
sglib-combined	23825.65 ms	23830.46 ms	+0.02%
slre	24500.28 ms	24500.28 ms	+0.00%
st	40556.74 ms	40562.34 ms	+0.01%
statemate	15245.67 ms	15245.67 ms	+0.00%
tarfind	12975.27 ms	17737.30 ms	+36.70%
ud	16133.38 ms	16133.38 ms	+0.00%
wikisort	11288.34 ms	17061.18 ms	+51.14%

Table 6.17 Runtime in the internal monitor implementation measured over Embench-IoT, before and after the instrumentation process.

pancy of the FPGA on which the experiments were conducted, i.e., Xilinx Artix 7 XC7A100T[28]. Recall that the PLA of the Control-Flow Logic Verifier (CFLV) requires 320 SRAM cells for its configuration of 40 lines for each of the 8 groups, beyond the LUTs employed to implement its logic. The Control-Flow Address Stack (CFAS) and Control-Flow Register Stack (CFRS) both consist of 16 32-bit entries. Sizing of the stacks was done differently than in the previous case for different reasons. First, the implementation was not used for security measurements, and the requirements were only dictated by the execution of benchmarks. Second, by core construction, it was preferred to employ registers rather than Block RAM for the stack implementation, to reserve RAM to code and data memory alone. In fact, basing the internal shadow stacks of a processor architecture on Block RAM means increasing the routing needed for connections and making them as a part of memory, which would have decreased the maximum clock speed after extensions. The basic version of the processor occupies 4.55% of the available LUTs, with an increase

[28] https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview

of 0.42% for the logic needed for the three security modules. As for the registers, the increase is 0.72% compared with an original occupancy of 2.05%. Block RAM utilization for code and data memories is in line with the above, and is 2.63% of the available resources, with no addition after the security extensions for the reasons described above. It is also reported the relative overhead between the two versions of the processor, which is 9.19% on LUTs and 34.76% on registers. The latter is mainly due to the increases brought by the two shadow stacks. These increases are not to be considered absolutely, but are to be put in the context of a minimal processor optimized for reduced resource consumption.

Resource	**Available**	**AFTAB**	**Utilization**	**Secure AFTAB**	**Utilization**	**Increase**
LUT	63400	2884	4.55%	3149	4.97%	+0.42%
Registers	126800	2601	2.05%	3505	2.76%	+0.71%
Block RAM	4680 KB	128 KB	2.63%	128 KB	2.63%	+0.00%

Table 6.18 Xilinx XC7A100T FPGA resource usage in the internal monitor implementation.

To conclude, Table 6.19 shows the summary of the solution in terms of overhead on size and runtime, compared with 5 other reference solutions in the state of the art on CFI solutions for RISC-V cores, or anyway for soft-core, through ISA customizations and/or insertion of internal verification modules. It can be immediately seen that the 3 solutions out of 5 being coarse-grained have minimal overhead on time, while on the other hand, the fine-grained solutions import significant overhead, even more than 100% in the case of DExIE. In 4 out of 5 cases, the overhead on code size is not reported. Only Sullivan's solution reports 13.50% on the CoreMark suite benchmarks. In terms of runtime overhead, only Sullivan is more advantageous. However, the applied protection is coarse-grained. Our solution impacts for less than 1% both code size and runtime.

Solution	**Processor**	**Benchmark**	**Size**	**Runtime**	**Granularity**
HAFIX	SPARC LEON3	CoreMark	n/a	+2%	coarse
HCFI	SPARC LEON3	CoreMark	n/a	+1%	coarse
FIXER	RISC-V RocketChip	Custom	n/a	+50%	fine
DExIE	RISC-V PicoRV32	Embench-IoT	n/a	+135%	fine
Sullivan	SPARC LEON3	CoreMark	+13,50%	+0,50%	coarse
PROLEPSIS	**RISC-V AFTAB**	**Embench-IoT**	**+0.63%**	**+0.58%**	**fine**

Table 6.19 State of the art comparison of CFI solutions for soft cores.

Chapter 7

Conclusions

In the present dissertation, the scientific achievements of a research path that has taken place over the past three years have been presented, which had as main topic the protection of special-purpose and low-power computer microsystems, more commonly known as *embedded systems*. The research has focused on how to design the architectures of these systems in a secure manner, or rather, in such a way as to resist the vulnerabilities that along their lifecycle afflict the software components they execute. In fact, this is the concept of *vulnerability-tolerant architectures*, a scientific term coined from the already pre-existing fault-tolerant architecture, and transferred from the safety to the security domain. At the beginning of the dissertation, it has been presented what is meant by *computer security* within the reference scientific school that inspired the research work. It has been defined what is meant by *threat*, *danger*, *weakness*, and *attack* in computer security, and what are the pillar properties on which security is based. The concept of *vulnerability* has been detailed, and what are the categories within which to classify one of them. The *zero-trust* principle has been discussed, and how a design that wants to be secure from the ground up and provide security to other domains must assume the presence of vulnerabilities and threats in the surrounding environment as certain. In the context of computer security, the role of hardware has been discussed, as the physical substrate of the systems from which they cannot disregard, and on which the design of a vulnerability-tolerant architecture is necessarily based.

The dissertation then delved into an excursus on the reference application domain. Embedded systems' nature has been analyze in detail, together with their main

characteristics, their deployment, their pervasiveness quantified in numbers. Based on these parameters, the most worrisome security issues related to these devices have then been highlighted, with the help of numbers and metrics released by leading international institutes in the security field. From the discussion, it emerged that security in embedded systems cannot be considered as a second-rate property, but should be instead aligned with the other parameters whereby the very mission of the devices depends on it, equally with reduced resource consumption, longevity, or hard real-time constraints.

Among all the security issues, the study focused on issues originating from the software these devices run, and opening the possibility for the attacker to corrupt the memory of these devices in such a way as to force them to execute sequences of unintended code. Thus, the so-called *binary attacks* have been discussed, their history, the classical defenses that the community has devised to defend itself, and also how attackers have engineered the attack to escape them. The concept of *control-flow integrity* (CFI) was introduced as a security property, ensuring that a program only follows execution paths as originally designed for the mission of that system. The original philosophy of this concept, together its advancement and conjugation throughout the recent history of the scientific literature, have been discussed discussed. A taxonomy was given to the various CFI proposals, which distinguishes the various techniques based on the domain in which they are applied, the static or dynamic nature of their policies, the type of software instrumentation applied to the programs, and most importantly, the *granularity* of the checks that are performed to verify compliance with the flow integrity.

Control-flow integrity solutions specifically designed for resource-constrained systems have been detailed, and arguments were offered to demonstrate how the general techniques offered in the literature may not always be feasible or may lack fundamental requirements in the embedded domain. In particular, it has been highlighted the need for architect solutions that would continue to ensure adherence to the tight constraints on real-time execution, through *schedulability analysis* and employment of parallel, non-blocking computation with respect to the main execution. It has been noted the poor consideration that almost all presented solutions have in considering the presence of *interrupts* as sudden events that break the execution of embedded programs and for which no static analysis can be implemented. Rather than for a solution, it has been pointed out the need for *a reference framework*,

with formal rules to describe both an *offline instrumentation* process and an *online monitoring* process, that can then be adapted to any architecture.

This has led to the presentation of the formal theory behind PROLEPSIS, the reference framework that is the main scientific output of this dissertation. Using a formal language derived from some distinct precedents in the literature, the behavior of an embedded program has been described, and how it can be hijacked to *illegal executions* through the intervention of an attacker. The concept of *tainted branch* was then introduced and defined, as a branch from which execution can escape from predefined paths. An algorithm for finding a *minimal* number of such branches within a program was defined, as well as the behavior of a monitor that performs detection and inhibition of hijacking attempts at these points has been described. Such a monitor is based on the classical *Policy Decision Point* (PDP) architecture, whereby *Policy Information Points* (PIP) and *Policy Enforcement Points* (PEP) are inserted within the program to be supervised. *Theorems* on which the operation of the monitor is based have been formally demonstrated. A protected execution environment according to the rules of such a monitor corresponds to a *vulnerability-tolerant architecture*, at least for vulnerabilities originating control-flow attacks.

To give empirical strength to the formal findings, the monitor has been implemented targeting two examples of vulnerability-tolerant architecture. In the former case case, the monitor is on an external hardware with respect to an ARM-based processor, and receives control information necessary for verification; in the latter, the monitor represents an extension of a RISC-V processor, and is implemented internally. The technical details of the two architectures have been presented, and based on scientifically established practices, performance parameters have been evaluated, in terms of binary size increase, additional time required for execution, and cost in terms of hardware resources. In the case of the external monitor, a 17.88% additional code base size has been measured, and a 1.10% additional execution time over a standard set of benchmarks, with a cost on the external reconfigurable hardware hosting the monitor logic below 3%. In the case of the internal monitor, we have got even better results, i.e., around 0.4-0.6% additional cost on all evaluation parameters. These results significantly outperform previous solutions, and provide evidence that the framework implementation can lead to solutions falling within the real-time requirements of the target domain. In both cases, such results are better than a group of reference solution identified for each type of monitor, where usually fine-grained solutions have double-digit percentage costs in all parameters,

while coarse-grained solutions are more efficient but highly ineffective. Regarding practical security evaluation, the framework includes a new runtime evaluation benchmark based on established practices in the literature, but specifically designed for embedded architectures. The experimental results have shown 100% coverage on all considered attack cases, empirically demonstrating the theoretical conclusions reached by the elaboration of the formal theory.

The results of this dissertation can lead to architectural solutions with a clear degree of novelty compared to what has been scientifically elaborated so far. In addition to the abovementioned results, it is important to note that the framework abstracts from the analysis of software vulnerabilities that the architecture is then charged with executing. Many CFI solutions presented in the literature require some degree of *branch regulation*, i.e., to change the nature of control-flow transfers, or even the high-level code style itself, to reduce vulnerabilities. On the contrary, the advantage of PROLEPSIS is not to require any security analysis, but offering solutions that can be resilient to a precise attack pattern in all its variants.

Among other advantages, it is also worth mentioning how it does not require any *pre-existing architectural support*. Everything required to implement protection is imported, without requiring anything from the architecture. On the contrary, many proposals require cryptographic extensions or modules such as the Last Branch Record (LBR) from which to read program activity. Above all, PROLEPSIS is not based on any secret: even when the attacker gets the control-flow graph information, he cannot exploit it in any way to his advantage. The communication with the monitor has a precise protocol, and excluding the possibility for the attacker to inject or modify the code, there are no combinations of using instrumentation instructions to escape the security checks.

The formal theory that underpins PROLEPSIS can be an important starting point for general research activities in the area of software security. While keeping its underlying concepts unchanged, the model can be extended and adapted to describe applications even outside the domain of reference discussed here, e.g., more pertinent to classic products such as personal devices or servers. In this sense, an important adaptation work has to be done in the context of *multitasking*, so that one can shift from the context of one main application to that of many applications running in parallel, with the consequent need to introduce a specific formalism for the context switch and its security treatment. Then, valuable results can still be achieved in

the area of security evaluation, and in particular in the establishment of a clear and universal standard, which can be applied in all contexts to quantify the coverage of a security solution.

Last but not least, PROLEPSIS offers protection that works at the level of the binary code, since it involves translating policy information and enforcement points into machine instructions. Such translation can take place in one of the compilation steps, if the compiler is duly integrated, but also after compile time, as it has been presented earlier, through a mix of decompilation and instrumented recompilation. This can provide protection even in *legacy systems*, i.e., for which developer support may have ceased or intellectual property rights to the firmware may not be owned. In this regard, PROLEPSIS was included in the EU project SPARTA[1], funded by the European Commission and participated by more than 40 academic excellence centers in cybersecurity across the Union. In particular, the framework has been presented within one of the tasks of the project, entitled "*Hardening Legacy Components*". The implementation based on the ARM microcontroller and the external FPGA monitor described in Section 6.1 was placed within a *cyber range* designed by the SPARTA community, that traces an *intellingent infrastructure* in the smart building domain [188]. In this environment, the protected device according to PROLEPSIS is a sensor of physical accesses to the building, which despite having a vulnerability in the firmware that allows memory corruption via a Bluetooth channel, manages to behave as a vulnerability-tolerant architecture and block attack attempts. The participation of the framework in the European project has been of paramount importance for its development.

Moreover, the development by the SPARTA community of a hybrid cyber range, i.e., with physical hardware components integrated within a simulated infrastructure, fits into the context of another major research issue that has been addressed along the three years of the present research path, namely the one related to education and training in the field of hardware security. Indeed, if an intelligent infrastructure simulator can be used as a testing ground for the effectiveness of certain distributed security solutions, e.g., in the IoT domain, it can also be employed as a training polygon for security professionals, for example tasked with testing the same security solutions outlined by the researchers, or responding to a simulated attack situation on the infrastructure itself. Placed in the reference theory on hardware security

[1] https://www.sparta.eu/

presented in the Introduction of this dissertation, this topic has led to some scientific achievements over the past three years. Its detailed discussion is included in the Appendix of this thesis.

Appendix A

Hardware Security Education and Training

A.1 Introduction

Throughout this dissertation, from the introduction to the scientific proposals presented, it has made clear how computer security is now a topic of primary importance that touches all aspects of society and the lives of individuals. In particular, Chapter 2 offers a glimpse into a reality that seemed unimaginable to us only a few years ago, whereby the domain of computers has merged with the various domains of everyday structures and applications. It has been discussed how the real phenomenological dimension of the modern days is *data*, and how this necessarily entails the ubiquity of the threat, with attackers being everywhere just as everywhere is data. As stated earlier, and as some interesting cited analyses also show [42], the process has taken on incredible speed in recent years, probably too much for the cognitive capacities of many of the professionals and citizens who have seen their business and private spheres rapidly transformed into a digitized reality, without having a full awareness of the possibilities, but especially the risks, involved in all this process.

The $(ISC)^2$ Association published its Cybersecurity Workforce Study last year [189], estimating that the global lack of qualified personnel on cybersecurity reaches almost 3 million. Also interesting is what comes out from a study conducted by the ISACA Institute [190], which reports that 50% of human resource managers believe that the personnel hired in cybersecurity do not actually have the right knowledge

for that role, and another 16% believe that they are not even able to accurately assess the cyber risks related to their business. The reasons for this are among the most diverse, but certainly a big negative contributor is the shortcomings present in the *training programs* for university students choosing a cybersecurity study path, but also in the training and update courses for professionals who have to work in the field. Courses are often taught by educators who, due to the complex hierarchy mechanisms present in academies and companies, are in turn not fully aware of the landscape out there. This is reflected in inadequate programs, mainly based on theoretical and notional training, that while on the one hand at least help to frame the problem, on the other hand leave the practical part uncovered. Thus, on the one hand, we have cyberdefenders who study and try to get up to speed to come up with new protections with their focus in books and documents, while on the other hand, we have attackers who have the ability to think *out of the box* and anticipate all defenses [191].

For these reasons, it becomes essential to train a generation of *white hackers*, i.e., professionals who share with the attackers anticipation attitude, the way of thinking out of the box, the creativity, but above all the technical skills, despite being on the right side of the fence. To solve this problem for some years now, even in the scientific literature, people have begun to talk about the concept of *education through gamification* [192]. From static lectures and seminars, there is a shift to a learning model putting the learner directly in front of problems, and through the import of game rules, the learner gets points in relation to as many problems as he can solve. Perhaps, the most successful example of gamification in cybersecurity is the *Capture-the-Flag* (CTF) games [193]. These games are based on solving challenges in which the goal is usually to breach one or more vulnerable components of a computer system (such as websites, files, databases, or network devices). Challenges are architected in such a way that, through the use of an appropriate exploit, it is possible to get a *flag*, which is a unique alphanumeric string generated by the game organizers, certifying the achievement of objectives. Therefore, in CTF the participants impersonate the attacker, and are enticed to emulate his techniques in order to win the game. This stimulates them to enter into out-of-the-box thinking mood, in the hope that this will then help in a future professional career to anticipate the attacker's moves.

CTF challenges form the basis for training programs, national and international significant events, and even conferences around the world. There are different modes in which a CTF can be held, depending on the role and number of participants.

- In *Jeopardy* challenges, participants are asked to play unopposed, individually or in groups, in front of the vulnerable system that hides the flag. The game is thus played on the individual or collective ability to detect the vulnerability and use the necessary exploit to capture the flag. In Jeopardy competitions, a maximum time for solving challenges may be given, and participants are generally given a set of different miscellaneous challenges. The player or team that wins the game is the one who succeeded in solving the highest number of challenges.

- In *Attack/Defense* challenges, on the other hand, participants are usually opposed into teams, and each team is given an instance of the same vulnerable system. The teams' task is to attack the opponents' system while protecting their own from attacks. Usually, an initial time is given in which the systems are not reachable to each other, and the teams' task is to analyze the system to discover vulnerabilities in advance, for which exploits and patches can then already be prepared to apply to their own instance. In a second phase, the real game opens up, and each team can collect points of 3 different types. Points are in fact granted based on: (i) the number of flags captured from the other teams (in positive), i.e., *Attack Points*, (ii) the number of flags stolen by the other teams (in negative), i.e., *Defense Points*, and (iii) the percentage of total game time in which the services offered by the system remain reachable, i.e., *Service Level Agreement Points*. This last mechanism was introduced as a trade-off against a possible disconnection strategy, that would guarantee immunity from opponents' attacks.

- In a third mode, called *King of the Hill*, participants divided into teams compete for possession of a single vulnerable system, i.e., they try to break into the system, and once they gain nor control, they struggle to leave everyone else out. The game is won by whoever has maintained possession of the system for the longest time.

In addition to the classification by type, CTFs can also be classified according to the cybersecurity topic on which they are based. The most famous are:

- *Binary*. This is the category of challenges requiring the exploitation of a vulnerable software application, including the problems described in this dissertation. They take this name because the participant play at the level of the binary machine code, folding tools such as disassembly, decompiling or debugging. Some challenges in this category may require reconstructing the behavior of the program, to directly get hold of the flag, or use this knowledge for a later exploit (*Reversing*). Other challenges may require to use code injection and code reuse techniques, such as those described in Section 3.2 (*Pwning*);

- *Web*. This category of challenge requires breaching inside vulnerable web services, for example through command or code injection weaknesses on online forms or login pages [194]. Widespread examples include those involving the exploitation of SQL language vulnerabilities to inject malicious queries [195], or the use of malicious scripts hidden within pages (Cross-Site Scripting, XSS [196]);

- *Crypto*. These challenges deal with problems related to the encryption of messages or files, based for example on classic but vulnerably implemented encryption schemes, or brand new protocols devised by the game organizers and to be reversed. Some knowledge of the mathematical elements commonly found in cryptography is required, including modular arithmetic and prime numbers;

- *Forensics*. These challenges are named like this because they retrace problems that investigators or law enforcement agents often face when analyzing computer equipment. Very often steganography is employed, for example within malformed files or packet captures, or images that hide text or pieces of executable code;

- *Network*. In these challenges, participants are asked to use techniques for attacking network protocols, such as circumventing or corrupting rules imposed by a firewall, spoofing and poisoning attacks [197], or reconstructing a message from packets extracted from a plaintext conversation;

- *Miscellaneous*. Even if based on computers, these challenges are not very specific to a topic, and are mainly aimed at evaluating deduction skills of the

participants (e.g., solve a stepwise riddle, or cracking a passphrase by clues hidden in the surrounding environment, etc.);

A.2 Hardware-Based Capture-the-Flag Challenges

As can be seen from the above list, CTFs usually focus on widely known and covered aspects of cybersecurity, which we could almost call *mainstream*. The same news we are used to are reporting data breaches or intrusions into databases of large corporations, or DDoS attacks shutting down important web pages and services. Being organized by academic or industry specialist in the cybersecurity field, CTFs trace the topics that are more widely treated by these organizations, and in particular, on which there is a greater level of expertise. Among these topics, *hardware security* is not very often included, although we have seen how its importance takes on a central role in the defense of cyber systems (see Section 1.3). This is due to several concauses, the first of which is certainly a lower diffusion of expertise on the topic in academic and industrial domains. Hardware security issues are very often relegated to sector-specific industries, and although the trend is reversing after the recent discoveries of hardware vulnerabilities plaguing a significant portion of consumer processors used in desktops and servers [16] [15], there is still resistance to bringing this topic within cybersecurity training programs, and even worse in gamification events. A *cliché* that is very common to be heard is that hardware security issues are too complicated for the average tenor of participants in CTF events, such that it would take too long for people to really learn while having fun as they do with challenges on the other topics. This is a fallacious argument, that is mainly due to an absence of hardware security culture within the backgrounds of the organizers of these events. Believing that exploiting a hardware trojan has a greater degree of difficulty than mounting a ROP, XSS, or PHP code injection attack is purely arbitrary, and totally dependent on what a given individual may have previously known. As for the argument for which this specific training should be left to the companies in the field, this is equally fallacious, because it is unclear whoever should be training the next generation of hardware designers, who will be hired by those companies, if not the university and specialized training programs. A valid argument to contend with and to take up as a challenge in introducing the culture of hardware security within CTF training programs and events, is instead that hardware is in general too

complicated to manage, difficult to access, expensive, and challenging from the reachability and deployment point of view. On this, it is impossible not to agree that assembling a simulated environment for a vulnerable web page, rather than deploying Python scripts with the source code of vulnerable sites and programs, is far more comfortable than preparing challenges where, e.g., electronic boards, FPGAs, waveform measurement instrumentation, device test equipment, or physical wirings are used. On the other hand, reducing everything to a simulation, whereby even hardware security issues are rendered via software emulators, largely degrades the very nature of the issue, which is also based the physicality of the devices (think of the case of side-channel attacks, or fault injection). Moreover, some studies show how attackers use instead realistic testbeds to prove their attacks [198].

The above considerations regarding the low presence of hardware security in CTFs are supported by a study we conducted looking for any reference to the word "hardware" in the contexts of platforms and events that globally host challenges. In general, two types of instances have been mainly encountered:

1. specific events proposing an advanced CTF format which targets an audience of experts and professionals in the field;
2. generic platforms that have introduced challenges dealing with or involving hardware for some time, but where often no or poor knowledge about actual hardware security issues is required.

For example, the *HackAT* events[1], organized within some of the major hardware design and test conferences such as DAC (Design Automation Conference)[2], USENIX Security Symposium[3], and CHES (Conference on Cryptographic Hardware and Embedded Systems)[4], involve since 2018 a special competition that is held in two phases: before the conference, participants are given a description of a vulnerable System-on-a-Chip, and they must be able to discover as many vulnerabilities as possible, using any tool. Then, at the conference, a second vulnerable SoC is delivered to participants, and the finalists compete in a live time-limited CTF on that, always based on disvealing the highest number of vulnerabilities. The Hardwear.io

[1] https://hackatevent.org/
[2] https://www.dac.com/
[3] https://www.usenix.org/conference/usenixsecurity22
[4] https://ches.iacr.org/

platform[5] runs a similar CTF event focused on radio protocols, side-channel analysis, automotive, and even physical invasive attacks on chips. The internal track called Capture-the-Signal (CTS) [6] is especially based on reverse engineering of radio signals: participants face Jeopardy challenges of increasing difficulty, ranging from simple signal reading to demodulation and decryption. Another example is the CTF RHme event[7], run since 2015 by the Riscure company, which specializes in the security of automotive hardware. This competition features a multi-level challenge built on the Arduino platform[8]. Participants are given firmware to flash (or that has already been flashed) onto the board, and the score is increased by executing exploits involving binary reversing and pwning, as well as some related to cryptography, side-channel analysis, or fault injection attacks.

As for the most famous CTF international events, a sample of 40 challenges tagged as "hardware" has been collected, which have been offered in the last 3 years (2019, 2020, 2021) editions of 18 major competitions, i.e.:

- *Aero CTF*
- *ALLES! CTF*
- *Chujowy CTF*
- *CONFidence CTF*
- *CTF Internacional MetaRed*
- *Cyber Apocalypse*
- *Google Capture The Flag*
- *HTB Uni CTF*
- *ICHSA CTF*
- *Insomni'hack*
- *KnightCTF*

[5] https://hardwear.io/
[6] https://hardwear.io/usa-2021/ctf.php
[7] https://www.riscure.com/challenge
[8] https://www.arduino.cc/

- *Ledger Donjon CTF*
- *pbctf*
- *ph0wn*
- *Pwn2Win CTF*
- *UIUCTF*
- *WPICTF*
- *X-MAS CTF*

The tool employed for the survey was *CTFTime.org* portal[9], which is the main portal collecting CTF proposed at global events and continuously updated by teams from all over the world. The same sample can be collected by anyone by simply connecting to the website, searching entries for the "hardware" tag only, and discarding results that are previous than 2019. All challenges have been classified on several parameters, among which:

- the need for software reversing and pwning techniques to solve the challenge;
- the presence of any *Hardware Description Language* (HDL) in the files provided to participants;
- the abstraction level of such hardware descriptions;
- the presence of a local or remote electronic board;
- the involvement of software-based hardware emulation techniques to host the challenge;

The analysis revealed that 12.5% of the total (5 cases) involve the use of some physical board by ARM or Atmel, of which 2 in local and 3 in remote. Any of these challenges is actually requiring binary reversing and pwning activities over the software running on the board, and there is no case in which some knowledge of the CPU architecture is required, or any hardware description or schematic is given to

[9] https://ctftime.org/

participants. 22.5% (9 cases) of challenges is based on remote hardware emulation. In 8 cases of them, a known processor architecture (RISC-V, Intel or Atmel) is simulated, and in one case only the challenge is based on a program written in C++ interacting with a SystemVerilog module "executed" by Verilator[10]. Apart from this last case, no details are provided on how the hardware is simulated, even if this is probably done through a fully-software instruction set simulator (such as Unicorn[11]), and not through the simulation of the processor HDL netlist. As a matter of fact, this would have also been useless, since vulnerabilities are located in hardware modules that are externally connected to the processor. In 5 of these 9 challenges, some hardware description code is given to participants, and the flag can be captured by executing some software on the platform to trigger the vulnerability in the external module. Therefore, these challenges can be categorized as a mix between binary and hardware. In the other 4 challenges of this subset, the only required activities are reversing and pwning on the binary code. The Verilator-based challenge is therefore the only one requiring some digital hardware knowledge.

In total, the 17.5% of the 40 challenges involves the presence of HDL code, and another 20% require the presence of schematics, for a total of 37.5%. Beyond the 5 challenges involving remote hardware emulation and described above, the other 10 are just solvable by inspection. A global 25% of the sample is based on binary reversing and pwning, and albeit involving the presence of local or remote hardware, physical or emulated, the knowledge about their hardware details is in any case irrelevant for the resolution. The 37.5% left is made by challenges based on the extrapolation of information from a waveform capture, e.g., relating to communications on physical protocols such as CAN, I2C, SPI, for which no specific knowledge of hardware is needed in advance, and that we can probably classify as miscellaneous challenges (see Section A.1). Figure X summarizes findings of the described survey.

In light of what we have seen, it is clear that associating the word hardware with CTFs is still rather a dark field. Therefore, in addition to the topics identified in the list of Section A.1, we define a new category of challenges, the *hardware-based CTF challenges*, as those "*in which the challenger must exploit her/his knowledge about digital hardware (including methodologies and technologies related to design, validation, verification, testing, maintenance, etc., at all the abstraction levels), in*

[10] https://www.veripool.org/verilator/
[11] https://www.unicorn-engine.org/

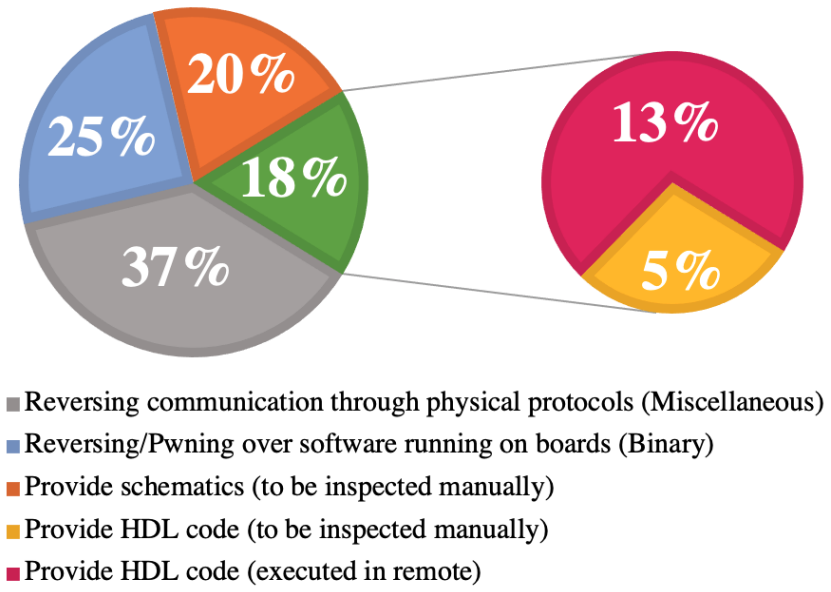


Figure A.1 Topics and typology distribution from a sample of 40 challenges marked as "hardware" in CTFTime.org (percentages are integer-rounded).

order to capture a flag consisting in identifying, remediating, or exploiting vulnerabilities artificially introduced either in the design or in the actual implementation of the hardware structure of a digital system". The definition is copied by [199], a paper written in 2020 in which we first speak of a distinct and clear category for such challenges.

It is worth highlighting the following practical implications of the aforementioned definition:

1. A challenge may not necessarily qualify as a hardware-based CTF challenge just because it appears to "rely" on hardware. Each software ultimately executes on hardware, hence the condition of "running on a hardware device" cannot be used as qualifiers;

2. In a true hardware-based CTF challenge, successfully capturing the flag must need background knowledge of digital hardware, and cannot be solved by only relying on other lateral knowledge;

3. In a true hardware-based CTF challenge, the flag could not be captured by simply taking advantage of artificially introduced vulnerabilities in the software

applications running on the hardware device, or in the communication or security protocols adopted by that device;

4. A true hardware-based CTF challenge can be implemented and proposed in actual competitions even without using any physical hardware support, as it can rely on some specific design elements or device functionalities, that may be disclosed to participants via description files or appropriate EDA environments and tools.

The paper also introduce a new comprehensive *taxonomy* for hardware-based CTF challenges, by individuating some orthogonal dimensions:

1. **Challenge Purpose**: challenges must be planned differently depending on their overall goal, differentiating between:
   - *Training*: in this case, the challenge should be structured to smoothly guide the students through the various learning steps, which are typically characterized by an increasing level of complexity;
   - *Competition in Jeopardy style*: (cfr. Section A.1): these challenges are typically more hard-to-solve versions of the ones used for training, with extra tricks purposefully incorporated in accordance with the difficulty level of the competition;
   - *Competition in Attack/Defense style*: (cfr. Section A.1): here, each team gets a same copy of the challenge object, and each instance may be accessed concurrently by all the teams and by the game server, which makes these challenges the most complex ones. This essentially implies that, in order to correctly queue, manage, and service all the incoming concurrent requests when a physical hardware device is involved, each instance must be equipped with a custom *software wrapper*.
2. **Challenge Execution Mode**: this dimension mainly focuses on the resources offered to the participants to solve the challenge. Usually, three possibilities are taken advantage of:
   - *Inspection-based*: the challenge object is the description of a digital hardware (often schematics and/or HDL files, relating to devices defined at the gate or register-transfer level, but also reports from its synthesis

or operation). Static examination of these files and/or the use of local emulation tools on users' PCs, as necessary, lead to solve the challenge. The flag is often an encoded response to a target-related question (e.g., the input combination generating a given condition, or a unique code that can only be recovered from the precise position of the vulnerability, etc.);

- *Simulation-based*: here, the target is offered to participants through a hardware logic simulator, through which it is typically possible to write input, read output, and, depending on the difficulty, control internal signals as well. Participants are provided with a description of the target (such as HDL, schematics, or another), which they use for a preliminary understanding and perhaps local simulations, to then send an exploit to the server hosting the operating target. In this situation, a remote service must be used to block access to a hardcoded flag in the design: obviously, participants receive a cleaned copy, i.e., with flag hidden;
- *Device-based*: participants here deal with actual hardware, that has a vulnerability allowing the flag to be captured. In this mode, an application with the primary responsibility of reading inputs, transmitting them to the device, and enabling the exit of outputs, always initiates communication with the device. The participants may receive a physical copy of the device, or they may connect to a remote service to reach it. The interaction program must be installed on each participant's computer in the first case, but in the second case, it can operate behind a TCP socket that is open to participants. As for the previous mode, participants are given a sanitized representation of the target in both situations.

3. **Challenge Topic**: there are many topics that may be explored by a hardware-based CTF challenge, the most relevant of which are:

   - *Hardware Trojans*: participants get digital hardware that has had a hardware trojan [200] intentionally injected into it. The players are able to capture the flag after correctly identifying and/or exploiting the trojan;
   - *Unprotected test infrastructures*: the flag in these challenges may be earned by deftly utilizing a test infrastructure present in the physical device. These include basic scan chains [201], the IEEE standard 1149.1 Test Access Port and Boundary-Scan Architecture (a.k.a. JTAG) [202],

the IEEE standard 1500 for Embedded Core Test [203] and others, all of which are left to unrestricted access;

- *Undocumented functions and features*: participants are given associated material in addition to hardware specifications or implementations, some of which odd characteristics are purposefully left out. For instance, machine instructions, parts, or undocumented consequences of the combined use of numerous described components may be among them [204]. Only using these secrets grants the flag capture;
- *Design bugs and flaws*: participants must cope with hardware that has some design defects or vulnerabilities [12], through which the flag can be accessed. Other examples include internal race conditions for which sensitive information might be disclosed, or improperly implemented machine instructions, etc;
- *Physical Attacks*: participants get a hardware device that is susceptible to timing or power side-channel attacks [14], or that must be attacked through some invasive attack techniques [205]. Participants may have a set of physical equipment at their disposal, so they can carry out the attack and capture the flag;
- *Weak implementations of hardware-based security modules*: The hardware provided to the participants is an instance of a *Hardware Security Module* (HSM) (e.g., hardware ciphers, random number generators, authenticators, etc.), but it has been built with some vulnerabilities that may be leveraged against it to access the flag.

4. **Hardware Device Description**: *Abstraction Level* and the *Representation Domain* are two orthogonal subdimensions that must be taken into account when describing a hardware design:

   - **Abstraction Level**: it indicates the granularity of information included in the system description, which generally varies from the *System Level* to the *Register-transfer level* (RTL) and the *Gate level*. Lowest abstraction layers are rarely ever employed;
   - **Representation Domain**: the offered descriptions might fall within the *Behavioral* or *Structural* domain. In the first, the behavior of the target hardware system is described in terms of properties (both functional

and non-functional ones), which specify what the system does and the environment in which it operates; in the second, the structure (i.e., the topology) of the target system is described in terms of a set of properly interconnected functional building blocks.

A.3 Cyber Ranges

As already introduced, the shortcomings present today within education programs for cybersecurity professionals concern not only the lack of a practical imprint, but also the little coverage with respect to central topics such as hardware security. As can be imagined, the combination of these two problems necessarily leads to some considerations about the training environments that should be used by learners. We saw above how only 13% of the sample of hands-on cybersecurity exercises give the learner the opportunity to interact with a live running hardware. This means that in 87% of challenges deemed as "hardware", these either do not deal with hardware security topics, or when they do deal with them, they do not employ any environment that involves a running instance of a piece of hardware. The rise of SDN (Software-Defined Network) [206] proves the ability to abstract a significant portion of the hardware from the physical main to construct a network for corporate use, and today's virtualization options are practically limitless. Even though the community may be intrigued by these possibilities, it is important to remember that *not everything can be done with software*. For example, many corporate networks still have physical firewalls, routers, and switches in their IT and OT infrastructures, especially when they need to interface with ICS and SCADA. In addition, IoT networks that are present across our cities, companies, homes, and bodies are unquestionably real. Because attackers themselves employ incredibly realistic testbeds to create sophisticated attacks, virtualizing these realities for teaching purposes may be limiting.

The community answered to this issue by conceiving platforms where it is possible to simulate realistic situations, replicating a variety of cyber-attacks, to improve operators' and users' abilities in spotting cybersecurity breaches and devise mitigation measures. This is what is commonly referred to as *cyber ranges*. According to NIST, they are "*interactive, simulated representations of an organization's local network, system, tools, and applications that are connected to a simulated Internet*

level environment" [207]. A valid and comprehensive taxonomy for cyber ranges was created and supplied by Yamin *et al.* in [208], which categorized them according to their purposes, the structure and design of the testbed and scenarios, the technology they employed, and other metrics. A schematic representation of the suggested classification system is shown in Figure A.2. The following are some of the taxonomy's dimensions:

- **Scenario**: it describes the storyline and the virtual environment for the test, competition, or exercise session that is conducted. It also lists the actions to be taken and how the described events are interlaced, to help users in identifying the simulation. A scenario is further defined by:
  - a *Purpose*, such as an experiment, a test of new components or a defense mechanisms, a competition, a training session, or a lesson;
  - an *Environment*, i.e., *physical, virtual,* or *hybrid*, as we will describe better below;
  - a *Storyline*, which describes how the activity should be done:
  - a set of *Objectives*;
  - a *Type*, i.e., *static* or *dynamic* depending on whether it may change while running
  - a *Domain*, e.g., IoT, industrial, etc.;
  - the set of *tools* employed for designing and developing the given scenario;
- **Teaming**: it defines how to classify the various groups or users or maintainers that engage with the cyber range. There are two different kinds of user teams: *Red Teams*, tasked with identifying and exploiting a scenario's vulnerabilities, and *Blue Teams*, tasked with resolving the vulnerabilities and problems identified inside the scenario. There are two types of maintenance teams: *White Teams*, which are tasked with putting up the scenarios and adding the vulnerabilities, and *Green Teams*, which are in charge of designing and building the infrastructure hosting the scenarios;
- **Monitoring**: it defines how management teams supervise user teams' actions in order to guarantee adherence to set guidelines and determine score. It is made by the *Methods* and *Tools* used as well as the abstraction *Layer* at which control actions are done;

- **Scoring**: it defines the procedures for determining the score, including objectives (such as finding a rated flag within the scenario) or the evaluation of participant activity logs;
- **Management**: it establishes the ways for managing the cyber range, through defining *Roles* and *Resources*; it also establishes the methods for providing scenarios via portals or dashboards (*Range Management*).

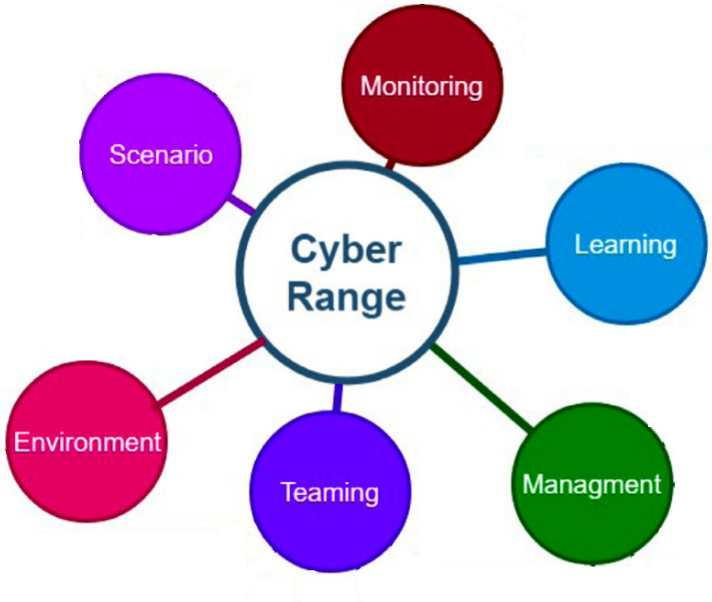


Figure A.2 Taxonomy of cyber ranges [208].

In the following of the Section, the most notable examples of such existing environments are given, classified according to the physical, virtual, or hybrid nature of their environment.

Physical Cyber Ranges

The goal of physical cyber ranges is to offer an exact replica of the target infrastructure in a secure and isolated environment. The present devices are all real, physical ones, and no aspect of the reference infrastructure is mimicked. This technique delivers the most loyalty to a target infrastructure or target devices, but it sacrifies

the capacity to readily scale, offers minimal flexibility, and has a significant impact on setup costs.

The National SCADA Testbed (NSTB) [209] by the US Department of Energy is one significant example. The project's goal is to investigate how an energy supply system might respond to cyber assaults on the one hand, and to create a hyper-realistic testbed for vulnerability discovery and mitigation strategies on the other. The testbed consists of a full-scale electrical power grid and many substations, all of which are built with industry-standard technologies such as SCADA and different network and industrial protocols, such as IP and MODBUS. Morris *et al.* [210] present another example, developing a critical infrastructure testbed at Mississippi State University. There are seven scenarios outlined, including petrochemical, steel industry, and electrical power system. The range contains a control system comprised of PLCs, sensors, actuators, interface and management software widely found in the industrial sector, as well as many physical processes that simulate components of the target infrastructure, such as water and oil containers and smart grid transmission. The technology is utilized not just for security assessments and research, but also for teaching activities inside the university's own courses.

Virtual Cyber Ranges

Virtual cyber ranges simulate all scenario components with the use of virtualization technology. This strategy gives the most flexibility and scalability, but it compromise realism. Because of the significantly higher capacity to maintain a completely virtual component architecture compared to physical cyber ranges, the category of virtual cyber ranges is undoubtedly the richest in examples.

The progress and aggregation of competence that the Horizon 2020 European project promotes through the projects CONCORDIA[12], SPIDER[13], and CYBER-WISER.eu[14] is one of the most significant works in this field. Several cyber ranges are suggested as part of the CONCORDIA project, each with different aims and domains. KYPO [211], developed by Mosaryk University and published as an open-source platform, is one of the most important. The KYPO cyber range is provided under the *Platform-as-a-Service* paradigm [212] and is meant to be modular,

[12] https://www.concordia-h2020.eu/
[13] https://spider-h2020.eu/
[14] https://www.cyberwiser.eu/

adaptable, and scalable. It is designed to run on cloud infrastructure. Virtualization methods enable the construction of sophisticated network infrastructures with numerous interacting devices and entities to reflect the desired architecture. The cyber range is entirely available via a web interface, with various features based on the user role inside the range, ranging from scenario planning to active execution.

The ICE&T Cyber Range[15] is an entirely virtual platform created by Munich University's CODE research institute. So far, it has implemented three separate scenarios with 80 individual exercises from various academic areas that are utilized for experimentation as well as instruction in the university's master degree program. The RISE Cyber Range[16], implemented by RISE Sweden's national research institute, allows users to emulate their networks or design complicated scenarios using virtualization and emulation. The project's goal is to create a cybersecurity training platform testbed for the public and commercial sectors, especially small and medium enterprises, which sometimes lack the resources to carry out such operations. It is important to emphasize the cyber range's usage as a test and validation facility for cybersecurity certification in Sweden. The aforementioned CYBERWISER.eu project focuses on creating a platform for cybersecurity professional training. All courses and materials given include lecture content, as well as hands-on practical experience on the remote cyber range platform. In order to enable for long-term platform maintenance, special emphasis is being placed on defining exact guidelines for scenario creation, from definition through requirements and deployment. In the end, The SPIDER European project is working on upcoming telecommunication technologies, namely 5G networks. They propose a *Cyber-Range-as-a-Service* (CRaaS) platform, that provides specialized training to both professionals and novices in the sector. To replicate 5G features, the platform makes use of standard virtualization technologies. The planned cyber range's objective is not just to give training support, but also to provide decision support for investments and metrics for evaluating the economic effect of a successful attack by adding econometric models.

FITS [213] is another important example of testbed envisioned as an infrastructure for network experiments, utilizing virtualization software to reproduce the network environment required by users to test their solutions. Several nodes are located across the world to support the infrastructure, not just in Brazilian institutions but also in European ones. Later, the FITS infrastructure was augmented with the

[15] https://www.unibw.de/code/forschung/zentrallabore/cyber-range
[16] https://www.ri.se/en/cyberrange

RIO platform [214], an orchestration architecture, to facilitate the deployment of experimental scenarios. In RIO, the features of the experiment may be defined using a descriptive language stored in a configuration file. The configuration file is interpreted to apply the desired characteristics during the creation of virtual machines and all virtual components of the scenario.

Then, recently, virtual cyber ranges have begun to be specifically thought to be employed in cybersecurity academic courses. The University of Potsdam's Tele-lab platform [215] is a laboratory based on the usage of virtual machines allocated to students for their activities. The content of virtual machines varies depending on the learning aim for which they are designed, and various machines are built from a pool of templates. The authors of the aforementioned work focused on the dynamic deployment of exercise scenarios: it is possible to adjust a set of parameters for the virtual machines (e.g., scripts for executing programs, network connections, etc.) to simply construct new scenarios while avoiding the reuse of previous ones.

Hybrid Cyber Ranges

As the name implies, hybrid cyber ranges combine the best of both physical and virtual methods, employing virtualization techniques to promote scalability and flexibility, while employing real hardware devices as targets of the activity carried out to create a high level of realism.

Among the most notable works, Mallouhi *et al.* offered a testbed to assess security solutions for SCADA systems [216]. They propose a hybrid testbed made up of a real SCADA controller and many virtual components that may be used to simulate Smart Grid infrastructure. Several macro blocks can be identified: (i) a control center that provides means to interact with the simulated model's components, (ii) a power distribution grid, (iii) a network of simulated components such as PLU, RTU, and various devices, and (iv) a simulation system of the electric grid controlled by the SCADA system. Hong *et al.* give another example of a power grid hybrid testbed in [217]. A actual SCADA system is used, while the power supply and different electronic components are mimicked using software and virtualization. Furfaro *et al.* [218] propose a hybrid cyber range in which real and emulated smart devices (such as surveillance cameras and Android smartphones) cohabit and collaborate inside a virtual environment intended for testing security for IoT applications. SmallWorld [219], the original program used to deploy the scenarios, is the most intriguing aspect

of this study. The scenarios in SmallWorld are not considered as monolithic blocks, but as a collection of interrelated elements that promote reusability and scalability. As a result, developers may deploy virtualized components according to the unique environment, from the operating system to the devices. Emulation of active entities' behavior, such as malevolent users or software, is another key part of the work done. The evolution of the scenario during execution might be different each time, even though the starting conditions are the same, significantly boosting its reusability.

Tsai *et al.* presented Testbed@TWISC [220], a network emulation testbed designed for research purposes in Taiwan. Users are given with a web-based interface, through which scenarios are developed using the available hardware and virtual resources, as well as a number of security analysis , such as exploit tools for the attack and firewalls or control access devices for the defense. Maintainers may design the scenario using a variety of hardware components such as switches, IoT devices, and routers. To increase the system's scalability and flexibility, virtualization techniques may be used to replicate other components, network infrastructures, and devices.

The hybrid cyber range category is represented by two cyber ranges in the CONCORDIA European project. The TELECOM Nancy Cyber Range, for example, is powered by the DIATEAM HNS (Hybrid Network Simulation) cyber range server[17]. The platform is utilized to assist educational activities and experiments at TELECOM Nancy University. While it runs scenarios using normal virtualization, it is possible to connect various physical networking platforms or devices for testing and experiments.

In light of the above, among all types, hybrid cyber ranges prove to be the best examples for implementing specialized training on concrete problems related to hardware security involving real use cases, but without losing the advantages that are offered by virtualization, especially in the area of internal network organization and the possibility of offering the environment to learners who are also far away from the physical location of the devices. In the next two Sections, two examples of hybrid cyber ranges will be presented, which originated in the context of Italian and European major projects, respectively, and whose implementation was participated along the three-year research path described in this dissertation.

[17] https://www.diateam.net/what-is-a-cyber-range/#hybrid_cyber_ranges

A.4 The PAIDEUSIS Hybrid Cyber Range

The present Section describes PAIDEUSIS, a cyber training camp located at IMT Lucca (Italy) that combines virtualized and physical components, including a wide range of IoT, industrial control, and network devices. The term παιδευσις (read /paidèusis/) was used in Ancient Greece to refer to the education of young people for life and war, as well as the location where this instruction was imparted. Our hybrid cyber range project was named with the transcription of this term, as to recall its mission. In fact, PAIDEUSIS's goal is to provide multiple scenarios and testbeds through the combination of devices and services, ranging from guided programming training of devices, to actual training sessions in VAPT (Vulnerability Assessment & Penetration Testing), with gaming sessions in the CTF (Capture-the-Flag) domain, up to Cyber-Defense Excercises (CDX) [221] for experts employed in cybersecurity company teams. The cyber range was first presented in a paper of 2020 [222].

PAIDEUSIS is a hybrid cyber range:

- completely useable from a remote, with the ability to interact with actual equipment from any location;
- that has non-fixed but changeable scenarios, based on demands;
- remotely reconfigurable, featuring procedures and languages for creating new scenarios.

Among the offered ones, it is possible to find scenarios for:

- training in the programming and exploitation of Hardware Security Modules (HSM), such as the SE*cube*™ platform[18];
- training on configuration issues and artificially introduced vulnerabilities in communication devices, such as Wi-Fi Routers;
- training on side-channel attacks over real processors, such as the ChipWhisperer-Nano devices[19];

[18] https://www.secube.blu5group.com/
[19] https://www.newae.com/products/NAE-CW1101

- training and competitions based on vulnerable hardware devices, both synthesized on FPGAs such as Xilinx PYNQ[20] and emulated through hardware simulator engines such as GHDL[21];
- simulating IoT networks and ICSs with SCADA controllers and actual sensors, for example employing devices such as the arancino.cc board[22].

PAIDEUSIS is initially conceived as part of the Italian project CyberChallenge.IT[23]. This is an initiative of the Italian Cybersecurity National Laboratory[24], and is the major Italian track for cybersecurity education of young people aged 16 to 24 from Italian high schools and universities since 2017. The project involves a selection step, as well as an initial training phase that combines more traditional education with training on a very large collection of CTF Jeopardy, which is posted on a unique online repository and produced by a community of specialists from all across Italy. Training activities are arranged by *venues* located around the country, with each venue selecting a team of approximately 20 members. The course concludes with two final events: a local Jeopardy tournament among the students of a venue, and a national Attack/Defense competition in which local teams compete against each other for the title of winner venue. The 2022 edition has received 5344 requests for participation from 34 different places, including the military academies of the Italian Army and the Air Forces.

As previously stated, the primary goal of PAIDEUSIS is to serve as a training platform for security elements associated to the hardware domain, which frequently do not play major roles in the current cyber range offer. The largest issue is thus adapting established virtualization technologies to enable scenarios to host physical devices as their main subjects. On the one hand, virtualization approaches are used to (i) create interfaces with hardware devices, (ii) appropriately export their services, and (iii) offer the necessary connections inside the scenario. On the other hand, to allow physical device to participate in the scenarios, wired or wireless connections are employed between them, together with other supporting devices (e.g., network infrastructure).

[20] http://www.pynq.io/
[21] http://ghdl.free.fr/
[22] https://arancino.cc/
[23] https://cyberchallenge.it/
[24] https://cybersecnatlab.it/

PAIDEUSIS is internally structured as follows. The fundamental entity is the *component*, which is any virtual or physical hardware or software part that is within the cyber range. A *subnet* is a collection of components that are physically linked by LAN (Wi-Fi, Ethernet) or PAN (USB, Bluetooth) interfaces (in the case of real hardware devices) or virtual interfaces (when devices are virtualized). The *range* (also called *theater*) is a collection of one or more subnets designed to host scenarios that are compatible and coherent. In principle, the subnet/range relationship is *many-to-many*: a range may require numerous subnets to be set up, and many ranges may use the same subnet of components. The abstraction is outlined in Figure A.3.

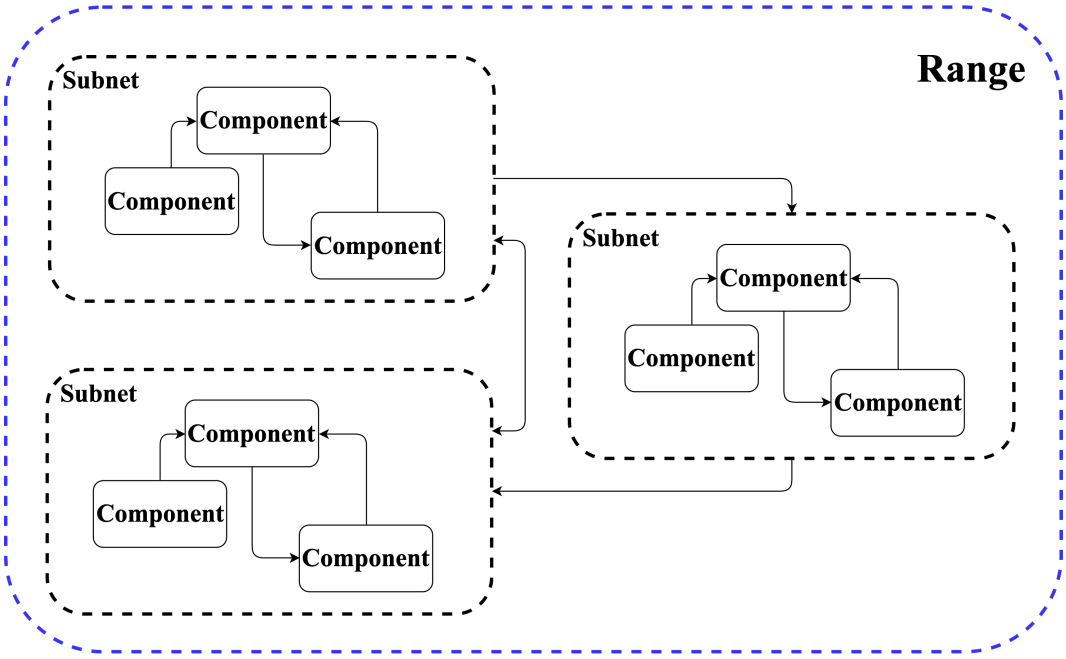


Figure A.3 PAIDEUSIS internal organization.

A *scenario* is defined here as a specific setup of the range, and it represents what was already defined in the taxonomy of Section A.3. Finally, a *session* is a single instance of a timed interaction between a user team and a specific scenario, where user teams (Red or Blue) are defined as in Section A.3. Sessions are stateless, in the sense that once they are gone, all information about how users interacted with the scenario is lost.

Figure A.4 depicts the physical topology of PAIDEUSIS. From the outside, two primary services are accessible via two virtual servers: one for user teams (in yellow) and one for maintainer teams (in green). These are followed by *infrastructure*

servers, to which the target physical devices are connected. Wi-Fi routers, SCADA controllers, microcontrollers, FPGAs, general-purpose IoT boards coupled to sensors or actuator devices, and boards for side-channel attacks are among the components found. Depending on the scenarios, infrastructure servers are in charge of either virtualizing the terminals that interface with the , or directly exposing their services. Alternatively, the machines can be used to virtualize the devices themselves, enabling scenarios in which just emulated hardware, or emulated hardware in conjunction with actual hardware, is present.

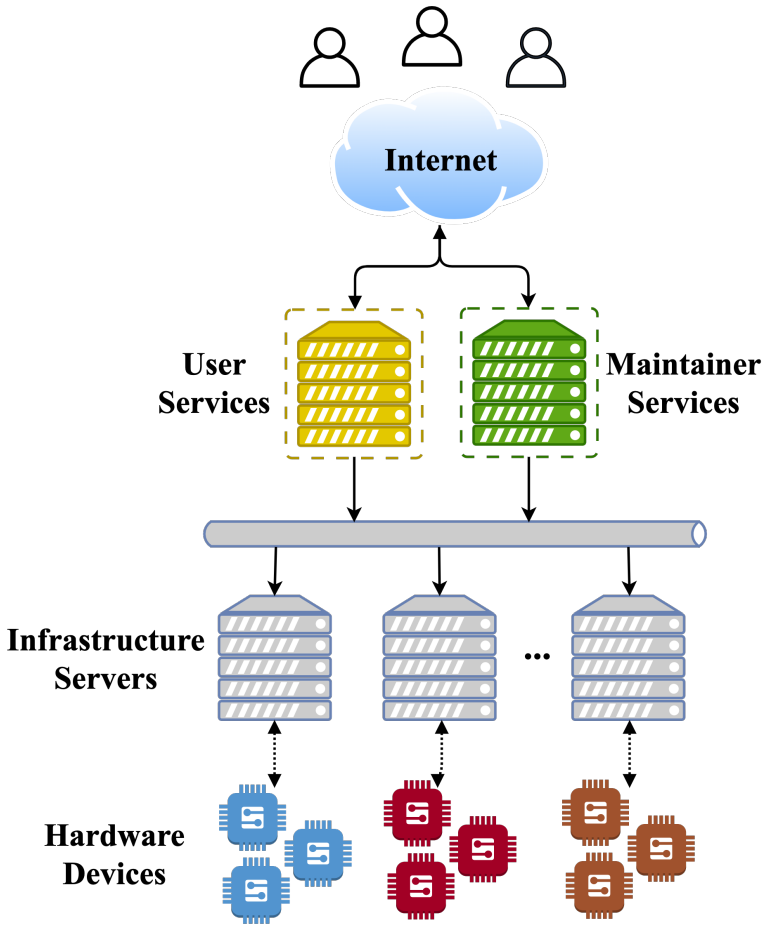


Figure A.4 PAIDEUSIS topology.

Groups of identical or similar devices are directly connected to a server that has been configured for them, e.g., scaled according to the specificities of the conceivable scenarios associated to these devices. The combination of one of these "thematic" servers and its cluster of devices may or may not create a range in itself.

Thanks to network virtualization technology, a range may be created by grouping any combination of components inside the cyber range, in any location.

PAIDEUSIS currently features 4 infrastructure servers, each one having 40 Intel Xeon E5-2650 CPUs and 128 GB of RAM, and running Ubuntu Server 21.10. Libvirt[25] is the open-source framework used to create virtual machines inside the infrastructure, based on command-line interaction and XML file parsing. WireGuard[26], an open-source VPN tool, is instead used to configure the virtual network. WireGuard employs strong encryption and performs better than comparable technologies such as OpenVPN. WireGuard allows users to connect to the scenario transparently with respect to the infrastructure, going through the user services abstracting from the infrastructural details. In other words, WireGuard *configuration files* are provided to users, who are then projected into the scenario after the configuration is applied to their terminals. Once connected, users begin the session by authenticating themselves, or through other techniques specified in the scenario requirements. WireGuard offers the benefit of abstracting the service from the infrastructure, as well as permitting an isolation-by-design that functions as a security mechanism for the platform itself: devices not included in the scenario are not included in the VPN configuration, and thus they are not reachable. As for service hosting, the infrastructure makes large use of Docker containers[27]. Docker enables the creation of lightweight Linux images that include all of the requirements required to run a certain program, with a high degree of repeatability. As a consequence, applications are deployed quickly and reliably, and environments are isolated from the hosting machine.

PAIDEUSIS does not define any *a priori* criterion for monitoring and scoring, and instead delegates mechanisms for user activity monitoring and score assignment to the White Team in charge of building up the scenario. These elements may be communicated via the maintainer services, which stay active during the sessions, and are obviously connected to the infrastructure in which the scenario is being played. For example, PAIDEUSIS has been used in combination with external CTF competition platforms, such as CyberChallenge.IT, which featured a separate flag submission mechanism, hosted elsewhere.

[25] https://www.libvirt.org/index.html
[26] https://www.wireguard.com/
[27] https://www.docker.com/resources/what-container/

In the following, some of the ranges that are currently hosted under PAIDEUSIS are presented.

- *Secure Device Training Range*: this range is primarily intended for hosting training scenarios for programming Hardware Security Modules (HSM), such as the SE*cube*™ device. Users can interact with a Xubuntu virtual desktop to which the development board is connected. The VM is equipped with the tools required to program the chip components. To accommodate all of the required USB connections, the infrastructure server has been upgraded with PCI-Express expansion cards. To isolate the IOMMU groups, each USB port has its own controller. Other than such Blue Team activities, the range may also be used for Red Teams, e.g., in exercises focused on exploiting flaws in the HSM firmware.

- *Digital Hardware Emulation Range*: here, scenarios based on emulated digital hardware designs that contains vulnerabilities [199] are mainly hosted. The current simulation environment is based on GHDL, which allows simulation of digital circuit descriptions in VHDL, ranging from small size devices, such as simple OTP authenticators or random number generators, up to real microprocessors, microcontrollers or DSPs. Hardware trojans [200] or other logical bugs and flaws are purposely implanted into circuits. More details are given in the following of this Section.

- *Digital Hardware Device Range*: the security issues stressed by the scenarios that can be set in this range are the same compared to the previous case, with the difference that in this theater, the vulnerable hardware is physically synthesized on FPGA devices, such as the PYNQ board by Xilinx. Compared with the previous case, in addition to offering a greater degree of realism, the range offers more possibilities in terms of expressiveness on the part of maintainers and interactivity on the part of users, allows for greater scale, and guarantees better performance. This range is discussed more thoroughly in the following.

- *ChipWhisperer Range*: this is the theater devoted to NewAE Technology Inc.'s ChipWhisperer-Nano devices, a board that permits physical attacks on a victim chip on the board, such as power side-channel analysis [14] or fault attacks [57]. The setup is similar to the Secure Device Training Range, even though

the interaction machines use far fewer resources, lack a graphical interface, and do not require the use of particular programming or synthesis tools. In addition to training scenarios for beginners in physical attacks, there is the option of hosting more difficult game scenarios, enabling a category of challenges that is seldom existent in the international CTF landscape, as we have seen above. More details are provided in the following.

- *Network Security Range*: this range contains network hardware devices such as routers, switches, and firewalls, on which one may practice security-oriented setup and programming abilities, as well as construct competitive situations. For example, one can think of scenarios in which participants are granted access to some network equipment, and the task may be breach into others starting from them, even exploiting weaknesses in physical LAN communication protocols, or to prevent others from taking possession of their equipment by setting access policies. Compared to a fully virtualized environment, the range has the advantage of allowing the use of physical protocols that cannot be virtualized, as well as using legacy technologies developed by specialized companies that have no virtual counterparts or cannot be rendered through software emulation.

- *Industrial Control System Range*: this theater contains devices for replicating industrial automation systems, and for rendering at scale IoT networks used to control smart cities and smart buildings. It includes real SCADA controllers such as the JACE 8000[28] by Tridium, and small control units are rendered by the use of boards such as Raspberry Pi[29] or arancino.cc, controlling e.g., smart cameras or quality air stations. An example of a scenario here implemented is that of a real acqueduct simulator, called EVA [223]. Here, the SCADA controller exchange data with IoT devices connected to sensors and actuators using protocols such as MODBUS over Wi-Fi, activating valves or indicating tank levels. Such a theater can be leveraged, for example, as reference infrastructure for hosting training scenarios for Blue Teams such as Incident Response Teams from the industrial sector.

PAIDEUSIS is in activity since 2020, the first year in which hardware-based CTF challenges have started to be submitted to the students of CyberChallenge.IT both in

[28] https://www.tridium.com/us/en/Products/niagara/jace-8000
[29] https://www.raspberrypi.com/

the training and in the final competition phase. Other than CyberChallenge.IT, in the same year, the topic was also faced during specialized training of TeamItaly[30], the team of young cyber-defenders that represent Italy in European and global competitions training, as well as within University courses, such as "Cybersecurity for Embedded Systems", held as Master Degree course in Politecnico di Torino. The Network Security range has hosted a scenario that has been featured in the CyberChallenge.IT Attack/Defense final event of that year, participated by 28 teams of ethical hackers from Italian academies. The challenge was called home r00ter, and it comprised of 28 Wi-Fi routers by Tiesse[31], configured with custom built software and linked to the same switch but isolated in terms of network because of the usage of a distinct VLAN for each device. Each team's router was only accessible via a separate VPN. First, participants had to discover an entry point and breach their own router. Second, while inside their device, the purpose was to exploit the routers of the other teamsusing the physical Wi-Fi connection. Participants in the local finals of the 2021 edition of CyberChallenge.IT were given simulation-based challenges that could be accessed remotely via the competition webpage. In the 2021 edition of the Italian CTF competition m0leCon[32], held in Turin in December of that year, included a device-based challenge built on the FPGA of the SE*cube*™ board. Teams received a copy of the gadget, while faraway teams received a virtual computer accessible via the competition's VPN and physically linked to the device. PAIDEUSIS continues to be the base where the hardware-based CTF challenges of the most important competitions at the national level are hosted, and it is also increasing its appeal at the international level, since also ENISA has expressed interest in the project and in hosting the challenges on the topic in the competitions it organizes, including the European Cyber Security Challenge (ECSC)[33], in which national teams of EU participate and where TeamItaly had the best placement in 2019 reaching second place.

Since the focus of the research work described in this Appendix is education and training on hardware security problems, 3 of the theaters presented above are analyzed more in details, i.e., the Digital Hardware Emulation Range, the Digital Hardware Device Range, and the ChipWhisperer Range. Our specific work done on these has been first presented in a paper of 2022 [224].

[30] https://teamitaly.eu/
[31] https://www.tiesse.com/en/products/routers/
[32] https://m0lecon.it/
[33] https://ecsc.eu/

Digital Hardware Emulation Range

This environment is built on GHDL, an open framework for simulating logic circuits described in VHDL language. The whole HDL compilation and simulation toolchain is included, as well as a set of preset instructions for a set of fundamental action for interacting with the circuit. The service is a traditional character-based TCP service, designed to read commands in a specified encoding to (i) set inputs, (ii) observe outputs (and optionally internal states), and (iii) advancing simulation time. For this purpose, it was required to embed GHDL commands within a *Python wrapper*, in order to filter instructions and signal names as needed. The wrapper is run on a virtual machine among those provided by PAIDEUSIS, and hosted by the Linux program `socat`, which accepts Internet connections. Every user that connects to the service receives a Python instance. Using GHDL commands, the instance starts the circuit simulation from the beginning of the main stimulation procedure of a *testbench* object, enclosing the challenge target. Indeed, the target is not directly emulated as a top entity, but must be encased in a portless top entity (exactly, a testbench), which initializes the inputs, resets the circuit, and starts the clock cycles automatically, when a timing signal is present. From connection onward, the player interacts with the circuit using the following commands (Figure A.5):

- `force <signal_name> <value>`, for setting a signal value;
- `examine <signal_name>`, for displaying a signal value;
- `run <time>`, for advancing the simulation[34].

A *filter* is required between the socket and the GHDL instance in order to examine and execute the user commands. The filter evaluates user inputs and discards those that are prohibited, using *configuration files* given by the challenge designers. The power of the `force` command, in particular, must be carefully regulated, as it may be used to set the value of internal signals, attempting to escape from certain circuit logic and discover the flag more easily. The presence of three configuration files is recognized by the system:

- `whitelist.conf`: includes a list of GHDL commands that are permitted. Any other command is denied and the user is notified;

[34]For simplicity reasons, clock advancing is automatically managed by the testbench.

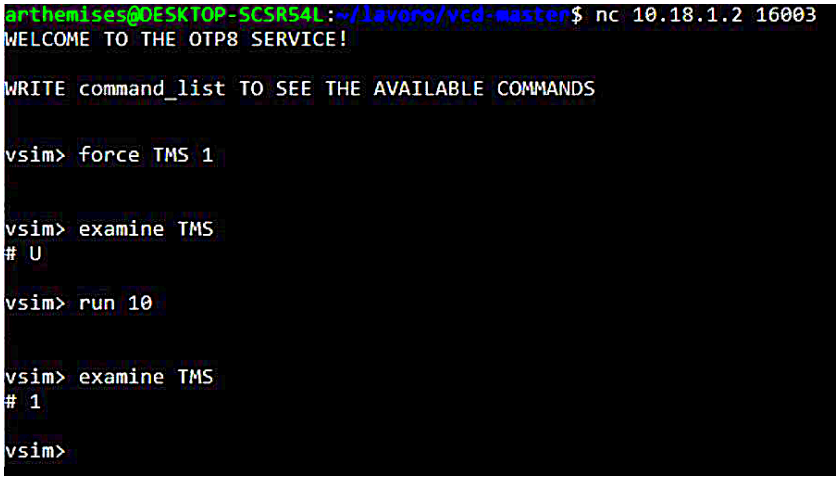


Figure A.5 User interface of the Digital Hardware Emulation Range.

- `blacklist.conf`: provides a list of words that must not occur in the challenge participant's input string. For example, the name of the HDL signal or constant containing the flag must be unquestionably banned. Unless an internal state is required for challenge resolution, the same holds for all internal signals and variables. It also contains all illegal punctuation symbols, which may allow access outside the interface or cause the environment to crash;
- `blacklist-force.conf`: is a custom blacklist for the `force` command exclusively, comprising the set of signals and variables that the participant cannot directly set.

When a service disconnection is detected, the program cleans temporary files and directories produced by the simulation instance and terminates it to free memory and CPU use. Alternatively, as one might expect, the service includes a timeout of a few seconds to prevent congestion. In fact, it is expected that challenges is resolved by automating the sequence of commands to be sent to the system.

It is also worth highlighting that any challenge designer can include their challenge in this environment, regardless of the characteristics of their target, as long as they supply a testbench in compliance with the features specified above and this triple of files.

Digital Hardware Device Range

This range leverages the use of the PYNQ-Z1 board by Xilinx (Figure A.6). This internally hosts a dual-core ARM Cortex-A9 processor and a Xilinx XC7Z020 FPGA, with 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops, and 630 KB of fast Block RAM. The processor runs PYNQ Linux, an operating system which is based on Ubuntu. Therefore, boards can independently reach the Internet without the need for a specialized infrastructure service. The set of boards connects to a switch that, through a router, reaches the outside of PAIDEUSIS. For the purposes of this project, the FPGA is used to host the challenge target, while the CPU is used to handle communication with the user by one side, and with the vulnerable hardware by the other. Having a real hardware running on FPGA rather than a simulated one obviously increases the efficancy of the challenge, which can fit into a more realistic storyline, since there is a tangible physical device to interact with, even if remotely.

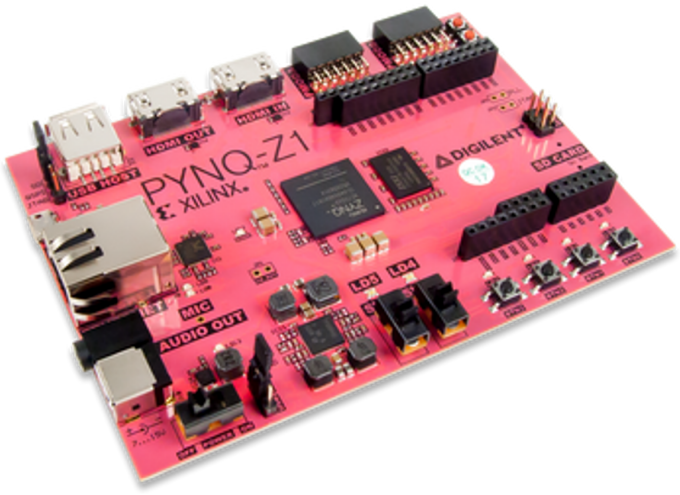


Figure A.6 The Xilinx PYNQ-Z1 board.

For the CPU-FPGA communication, the PYNQ leverages a library called `Overlay`. Communication with the hardware that is synthesized on the FPGA is virtualized by a Python library that allows for multi-level interaction, ranging from the most basic set of digital inputs and get of digital outputs, to more advanced communication protocols. Similarly to the previous case, Python scripts that make use of this library can be hosted via `socat` directly from the PYNQ processor itself. For example, let

us suppose that the challenge target is a silly multi-bit adder, like the one described with the VHDL code below:

```
library ieee;
use ieee.std_logic_1164.all;

entity adder is
  generic
    (len : integer := 32);
  port
  (
    clk : in std_logic;
    rst : in std_logic;
    a   : in std_logic_vector(len-1 downto 0);
    b   : in std_logic_vector(len-1 downto 0);
    c   : in std_logic_vector(len-1 downto 0);
  );
end entity;
```

The Python script hosting the challenge should contain an import of the synthesize bitstream of the target, stored into a file present in the PYNQ file system:

```
from pynq import Overlay
challenge_target = Overlay("adder.bit", download="False")
```

After this, the challenge target become available for the script, and inputs can be set, as well as output can be read:

```
adder.a = 43
adder.b = 80
# other statements...
print(adder.c) # 123
```

The user may not even be directly offered the ability to interact with the inputs and outputs of the digital system, but several levels of abstractions can be created to simulate any kind of protocol. Thanks to the considerable capabilities of PYNQ in terms of hardware resources, even large systems can be synthesized on the FPGAs, to make the Challenges even more adherent to realistic cases. For example, the reconfigurable part could host a custom architecture of a processor, and the service for interaction could allow the upload of a binary program to be executed on that

platform, with the purpose of triggering, e.g., a hardware trojan or flaw inserted into the datapath of the synthesized soft architecture. Another example is offered by a vulnerable peripheral that, through the software running on the PYNQ itself, could be exploited. A major advantage that is undoubtedly imported by this range over the previous one is the comfortable hosting of Attack/Defense scenarios, since each team can control its own PYNQ via SSH (which is more difficult to render if the hardware is simulated), and can patch vulnerabilities identified on the target by uploading a new bitstream to the board itself, which is then uploaded by the service script when it is back online.

With respect to the previous environment, various distinct infrastructure-side elements must be examined, including service containerization and request enqueuing. When a user connects, it is not merely a question of creating an execution instance; it is also necessary to connect to actual hardware, which must be ready and not occupied. The use of `socat` on the host PYNQ may help, but due to limited resources of the CPU, it might be not sufficient. Boards connection with the outside world may be viated through an infrastructure server rather than through a simple router. On such a server, an instance of HAProxy[35] may be launched during the environment setup on one of the infrastructure servers, to coordinate access to the challenge. The task of HAProxy is to remap all sockets on a single TCP port accessible over the Internet, enqueue all requests, and appropriately balance the load on the boards. The connection between a user and a board is maintained until the user exits or a timeout expires, depending also on the context in which the challenge is executed (training, Jeopardy, Attack/Defense, etc.)

ChipWhisperer Range

This environment is intended for training on hardware security vulnerabilities that are directly tied to the physicality of the device, such as conducting real power side-channel analysis [225] on actual chips. The ChipWhisperer platform from NewAE Technology is used in the environment (Figure A.7).

As it is also notable from the Figure, the platform is made up of two parts: an STM32F0 microcontroller that serves as a “victim”, on which firmware that executes cryptographic algorithms is flashed. The fundamental pins of this device, such as the

[35] http://www.haproxy.org/

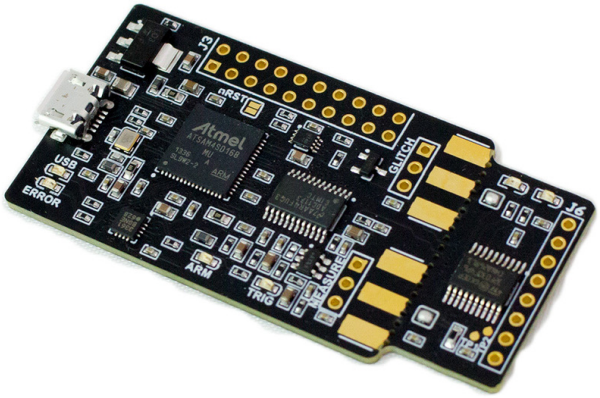


Figure A.7 The ChipWhisperer.

clock and power supply, are connected to a second "attacker" system, which has a more advanced Atmel CPU managing user interaction, power measurements, and fault injections, as well as data collecting from the victim. The platform is fully built on Python libraries, allowing for a simple and abstract interface in comparison to the complexity of the attack procedures. The environment includes a dispatcher service on the infrastructure machine to process requests from users, as well as a Docker container that users may download and start to create a local environment connecting to the remote one. In fact, the users analyze and visualize the captured data locally, and the remote service is just responsible for fulfilling the capture requests and provide users with data.

Each exercise is presented to the user as a Jupyter Notebook file[36]. In a nutshell, a Jupyter Notebook file is a Markdown file containing interactive graphs and executable code snippets. The file is shown to users as a website, and the code snippets are executed by an underlying Python kernel. The webpage includes editable code boxes, that allow users to test and re-iterate numerous attack strategies and acquire results in both graphical and numerical formats (Figure A.8). All of the tools required to execute and show the Notebook are already included in the container that users download (e.g., Jupyter engine, ChipWhisperer library, Python's `matplotlib`, and so on).

The remote service is in charge of receiving any incoming capture operation requests provided by users. Each request contains a collection of configuration data,

[36] https://docs.jupyter.org/en/latest/

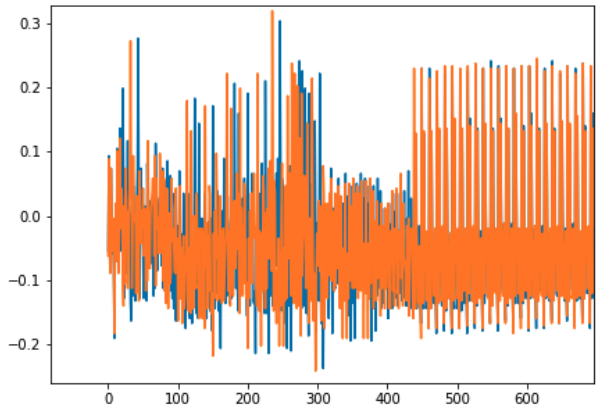
```
In [9]: #Example - capture 'h' - end with newline '\n' as serial protocol expects that
trace_h = cap_pass_trace("h\n")

print(trace_h)

%matplotlib notebook
import matplotlib.pyplot as plt
plt.figure(figsize=[7,5])
plt.plot(cap_pass_trace("h\n"))
plt.plot(cap_pass_trace("0\n"))
plt.show()
```

```
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.

[-0.05859375  0.09765625 -0.05078125 ... -0.1171875  -0.00390625
 -0.11328125]
```




```
In [7]: attack = cwa.cpa(project, cwa.leakage_models.sbox_output)
results = attack.run()
print(results)
```

```
Subkey KGuess Correlation
  00    0x2B    0.76781
  01    0x7E    0.75389
  02    0x15    0.80293
  03    0x16    0.74749
  04    0x28    0.78206
  05    0xAE    0.70145
  06    0xD2    0.72779
  07    0xA6    0.73236
  08    0xAB    0.75005
```

Figure A.8 Screenshots from Jupyter Notebooks performing offline side-channel analysis.

such as those reserved for the ChipWhisperer scope board and the capture configuration. The service must then tie the incoming request to the relevant ChipWhisperer, employing a “direct mapping”-like technique in which, given m available boards, all requests are evenly partitioned on m queues. In contrast to the PYNQ-based environ-

ment, a board only reply to a single capture request, after which the connection is instantly terminated. The remainder of the calculation required for the progression of the exercise is performed within the users' local container.

The various back-end components are also contained within a Docker container. Each of them is capable of writing informative logs, as well as reporting the most serious failures to the system maintainer, e.g., through Telegram alerts, allowing for rapid problem resolution when necessary.

A.5 HArMoNICS: Cyber Range for Intelligent Infrastructures

This Section is intended to present HArMoNICS, a cyber infrastructure located in the University Campus of Savona (Italy), that is inspired by a zero-emission building scenario within the domain of smart microgrids. All of the component in such an infrastructure are derived from (or inspired by) real systems that exist inside the original infrastructure. The primary objective for HArMoNICS is the strategic importance of intentionally vulnerable environments as a common setting for security research and development, other than training, to serve as cyber range for the Intelligent Infrastructure (II) domain. Indeed, such environments can be used not only as a classical cyber range, but also to foster and promote innovation by serving as a testing ground for new security design approaches. Although the concepts underlying Security-by-Design (cfr. Section 1.4) and its potential benefits are evident, prototyping solutions in a distributed and heterogeneous environment remains a significant issue, and all measures should be executed and populated with real security tools and processes to be effectively assessed. As a matter of fact, most of these processes might require stopping the infrastructure's operations, which is usually impossible. As a result, the actual effectiveness of the processes is frequently examined only after a real security event has occurred, and if they fail, then it might be too late.

HArMoNICS is presented as a hybrid cyber range that can be used by the community as a *playground*, e.g., to conduct training exercises and organize CTF or CDX sessions, as well as a *benchmark*, e.g., to systematically examine the efficiency of a certain security measure in a realistic environment that takes into account

numerous circumstances. These actions may be carried out by utilizing the built-in *use cases* (cfr. *ranges* in Section A.4). Moreover, use cases can be added to mimic unique security problems, such as those associated with new technology, thanks to the modular nature of HArMoNICS. Also, physical devices may be plugged in through a VPN-based integration mechanism, and HArMoNICS can be used to augment existing environments such as simulators or digital twins. The infrastructure was first presented in a paper of 2022 [188].

In summary, the following HArMoNICS characteristics are deemed as of paramount importance to the security community.

- Because the cyber range is intended and developed to resemble a real II, it incorporates a variety of technologies that may exist within the perimeter the reference domain.
- HArMoNICS includes 8 security and privacy use cases centered on specific threats and weaknesses common to most II, including but not limited to: (i) end-point device software integrity and updates; (ii) privacy-preserving data management and processing; (iii) intrusion detection; (iv) protocol verification; and (v) fog computing orchestration and hardening.
- HArMoNICS design and implementation are open source and publicly available, so it may be extended and modified.

HArMoNICS has been designed and developed within the context of the European project SPARTA[37], one of the pilot project on cybersecurity funded by the European Commission within the Horizon 2020 framework. The goal of SPARTA is to create and maintain an European roadmap in cybersecurity research, involving more than 40 relevant cybersecurity competence centers (within academies and research institutions) in the Union, from Greece, Spain, Italy, Estonia, Czech Republic, Finland, Germany, France and others. SPARTA is mainly divided into *programs*, each one having a precise mission. HArMoNICS has been a conception of the HAII-T (High-Assurance Intelligent Infrastructure Toolkit), tasked with creating an integrated framework and toolkit to aid in the design, development, and verification of large-scale distributed II security systems.

[37] https://www.sparta.eu/

The HArMoNICS network infrastructure is divided into three segments. `dmz`, `intranet`, and `iot` are the three parts. Servers hosting public services, i.e. those accessible from outside the network boundary, such as DNS, web and proxy, are linked to `dmz`. Instead, other servers and hosts are linked to the `intranet`, such as access control, intrusion detection systems, and data collection. Finally, `iot` is used to link field devices such sensors for light, air quality, temperature, physical access, parking, plus actuators, and fog nodes as intermediate data collectors. The smart building network is protected by a router firewall, which connects the perimeter to the public Internet. From the outside, accesses from and to the infrastructure might be done by servers for software update or smart parking, or by software developers or entities responsible for power distribution. Docker Compose[38] has been chosen as range deployment tool, and Docker containers are used to implement the above described network topology. Container connectivity is provided via virtual networks that are imitated by the host system. HArMoNICS has been developed through a standard *Infrastructure-as-Code* (IaC) called TOSCA [226], which is employed as orchestrator of the Docker containers. The infrastructure code is all available on GitHub[39], so that continuous integration is enabled. In fact, to import flexibility, a virtual private network (VPN) server provides direct access to each network. The VPN may thus be accessed from applications operating on the host machine, as well as from the Internet. The primary goal of the VPN is to allow the integration of physical devices, such as a wireless access point or an IoT device.

In the following, some of the use cases hosted within HArMoNICS are presented.

- *Software Integrity for Smart Sensors*. One of the most serious dangers to II security is the inadequate, or perhaps non-existent, protection of sensing devices, which frequently comprise a considerable portion of the perimeter of the whole infrastructure. A *smart door sensor*, for example, put inside the `iot` network, may be responsible for monitoring the entrance of persons within the building. To do this, it connects via Bluetooth to the user's mobile device and asks the transmission of a unique ID. On the personal device, an app is loaded that can connect with the door sensor and pass the person's ID to it. In the current scenario, the firmware may incorrectly judge the mobile app as a trustworthy actor. The sensor may just read bytes until the string termination

[38] https://docs.docker.com/compose/
[39] https://github.com/enricorusso/spartawp6

symbol is received, rather than sanitizing the incoming message. An attacker can take advantage of this vulnerability in a variety of ways, e.g., mounting a ROP attack (see Section 3.2). This use case can serve as playground, e.g., for exercises in the field of binary exploit and protection of IoT firmware. Here, it has been possible to employ PROLEPSIS, described throughout the course of this dissertation, in a realistic context, e.g., by equipping the ARM-based sensor with an external FPGA to act as a CFI monitor as described in Section 6.1.

- *Software Updates for IoT End Devices*. Low-power indoor sensors continuously gather data about the interior air quality, e.g., in order to detect and avoid poor air quality conditions. In the use case, such sensors are equipped with a COTS SoC featuring an ARM microcontroller communicating with low-power radio protocols. The sensor firmware is built on RIOT [227], an open-source general-purpose operating system popular in low-power IoT devices. The goal of this use case is to provide a testbed for software update security investigation, or for exercise exploiting update flaws.

- *Privacy-Preserving Data Processing*. The access control mechanism in HArMoNICS is administered by a dedicated server in the `intranet` subnetwork. The system controls who has access to the smart building. It also serves as an access control mechanism for cars that wish to utilize the college parking lot. There are several risks associated with such a scenario, including (i) *consent unawareness*, which refers to a user being unaware of the information disclosed to the system, e.g., allowing a malicious agent to retrieve his identity, or (ii) *policy and consent non-compliance*, which means that, while the system displays its privacy policies to users, there is no guarantee it actually complies with the advertised policies [228]. The use case can serve for exercises on expoliting privacy flaws in the reference scenario, e.g., for tracking and profiling users, or for assessing new authenticated protocols that do not fully reveal the user's identity to a verifier.

- *Intrusion Detection*. A sophisticated II backed by a large-scale network, comprised of diverse and disparate devices, is unlikely to be totally secure from instrusions. The goal of an Intrusion Detection System (IDS) is to identify such assaults on computer systems and networks. To cope with latent threats, an IDS continually monitors the running system and analyzes the collected

data to determine whether or not an attack occurs. HArMoNICS can be used to train corporate Blue Teams to effectively employ the information coming from the IDS to develop ready responses to attacks that could affect such a critical infrastructure. From a research perspective, instead, it can be used to evaluate the effectiveness of datasets against real-world attack scenarios.

- *IoT Communication Protocol Flaws*. The EnOcean protocol [229] is utilized inside HArMoNICS to provide communications between IoT devices interacting with a smart HVAC (Heating, Ventilation, and Air-Conditioning) system. Because IoT devices are manufactured by many companies, design flaws in the EnOcean protocol may have a significant impact on the correct operation of the HVAC system. An attacker might use such weaknesses in the EnOcean protocol to perform (i) *eavesdropping*, a.k.a. spying the system, (ii) *replay attacks*, in which (parts of) communications are captured for subsequent use, (iii) *man-in-the-middle* attacks, in which communication between two communication partners is intercepted and perhaps modified during transmission, (iv) *Denial-of-Service attacks*, which prevents genuine users from gaining access to the system. The use case can be helpful for training sessions for Red Teams interested in learning how to carry on such attacks, or for the assessment of verification solutions for protocols. For example, *formal verification* allows for mathematically proving the correctness of a target protocol in relation to a certain specification or property. Possible tests include functional correctness, verification or falsification of security characteristics, qualitative and quantitative analysis of protocol specifications or implementations [230].

- *Fog Orchestration Security*. The basic goal behind this use case is to examine the location of Fog and Edge devices and services for potential QoS and security concerns, as well as to identify non-optimal service distribution amongst Fog nodes. Two fog nodes are physically located in two distinct places and are both linked to the `iot` network. These Fog nodes can perform communication services in order to link with associated edge devices. Fog nodes contain services that monitor lighting qualities in rooms using light sensors, and can modify lighting based on human preferences, e.g., by activating smart bulbs. It is therefore provided a testbed for Fog orchestrators, to assess their ability to control services based on QoS and security requirements. Each orchestrator decision on starting/stopping/suspending/moving Fog services

should be checked for compliance with the minimal requirements imposed by various hardware and software constraints of the involved physical devices, as well as requirements arising from the specifics of the application area (e.g., health-related data should be protected better than environment monitoring data).

- *Fog Hardening*. Fog nodes themselves play a vital role in a IoT network such as the one featured in HArMoNICS. Indeed, they offer edge computing capabilities for end-users nearby, sharing the strain on cloud servers' resources. However, this network topology optimization exposes the security of the user- and kernel-space applications operating on Fog nodes, as they directly interface with potentially malicious end-user devices. This may relate to the exploitation of security vulnerabilities that jeopardize the confidentiality of data processed by user-space Fog apps, the integrity of the kernel-space Fog operating system, and, inherently, the entire Fog layer of the building. Again, this use case can be employed as playground for training and research activities on binary attacks, but on embedded systems that are more powerful than end devices, and that may feature full-fledged operating systems and even virtual machines running on it.

- *Managing Personal Data in Vehicle Recharge Process*. HArMoNICS features a car charging station. The user's personal device, the smart parking lot, and the power distribution infrastructure are its primary components. The charging procedure in the smart parking lot is started by the driver using his own device. Once the charging procedure is completed, the power distributor provides the charged energy amount to be paid (based on the smart meter). The scenario focuses on payment details, as well as the associated security threats. Payment information may comprise (i) the driver's identity, (ii) bank account and/or credit card information, and (iii) permission to debit/credit the bank account for the service. The use case can serve to develop verification tools and methodologies to implement compilance with, for example, GDPR principles [231]. For example, techniques based on the use of TLS can be developed, and public key encryption can be employed to encrypt payment details before sending them to the parking lot. From the perspective of security exercises, on the other hand, the use case can serve as an environment where it

is required to identify privacy issues, e.g., through the use of assessment tools such as DPO [232].

HArMoNICS is a case study that collects technologies linked to important security challenges in real-world contexts inside a single infrastructure. In fact, the design strategy incorporates both simulated and physical components that duplicate some of the attack vectors of relevance. In many cases, the scenarios that can be emulated through HArMoNICS replicate situations in which are involved critical infrastructures that, for example, may be of primary importance for the cybersecurity of an entire nation. In addition to the more common purposes of theater for training sessions, or for evaluating security research solutions, HArMoNICS can be of inspiration for the deployment of simulated infrastructures operating in strategic sectors. National agencies in charge of regulating security measures and procedures for such infrastructures could require strategic companies to own a similar environment, or equivalently, to integrate their use cases within a reference infrastructure for the country. Once this is done, agencies could require them to provide results of an assessment conducted on the vulnerabilities present in that company's ICT assets, e.g., after employing formal tools for modeling threats, weaknesses and attacks. A very interesting example is provided by ThreMA [233], a threat modeling tool that aims to automate the process of identifying threats to a digital infrastructure through the use of semantic models and formal rules. ThreMA is based on the use of an *ontology* that provides the vocabulary, or *metamodel*, for describing ICT infrastructures, while threat modeling logic is defined through rules that ontology reasoners can interpret and use to identify threats to which the infrastructure under analysis may be subject.

Bibliography

[1] R. Ross, M. McEvilley, and J. Oren, "Systems security engineering: Considerations for a multidisciplinary approach in the engineering of trustworthy secure systems," National Institute of Standards and Technology, Tech. Rep. 800-160, 2016.

[2] Gartner Glossary, "Definition of Operational Technology (OT)," https://www.gartner.com/en/information-technology/glossary/operational-technology-ot, 2022, [Online; accessed October 3, 2022].

[3] "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer," https://cwe.mitre.org/data/definitions/119.html, 2022, [Online; accessed October 3, 2022].

[4] "CVE-2019-6778," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6778, 2019, [Online; accessed October 3, 2022].

[5] "CWE-127: Buffer Under-read," https://cwe.mitre.org/data/definitions/127.html, 2022, [Online; accessed October 4, 2022].

[6] "CWE-209: Generation of Error Message Containing Sensitive Information," https://cwe.mitre.org/data/definitions/209.html, 2022, [Online; accessed October 4, 2022].

[7] "CWE-912: Hidden Functionality," https://cwe.mitre.org/data/definitions/912.html, 2022, [Online; accessed October 4, 2022].

[8] "CWE-327: Use of a Broken or Risky Cryptographic Algorithm," https://cwe.mitre.org/data/definitions/327.html, 2022, [Online; accessed October 4, 2022].

[9] "CWE-335: Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)," https://cwe.mitre.org/data/definitions/335.html, 2022, [Online; accessed October 4, 2022].

[10] M. Alsharif, S. Mishra, and M. Alshehri, "Impact of Human Vulnerabilities on Cybersecurity." *Comput. Syst. Sci. Eng.*, vol. 40, no. 3, pp. 1153–1166, 2022.

[11] F. Salahdine and N. Kaabouch, "Social engineering attacks: A survey," *Future Internet*, vol. 11, no. 4, p. 89, 2019.

[12] P. Prinetto and G. Roascio, "Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy," in *ITASEC - Italian Conference on Cybersecurity*, 2020, pp. 177–189. [Online]. Available: http://ceur-ws.org/Vol-2597/paper-16.pdf

[13] R. Anderson, *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020.

[14] F.-X. Standaert, "Introduction to side-channel attacks," in *Secure integrated circuits and systems*. Springer, 2010, pp. 27–42.

[15] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[17] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[18] J. Kindervag *et al.*, "No more chewy centers: Introducing the zero trust model of information security," https://media.paloaltonetworks.com/documents/Forrester-No-More-Chewy-Centers.pdf, 2010, [Online; accessed October 6, 2022].

[19] UK National Cybersecurity Centre, "Device Security Guidance - Guidance for organisations on how to choose, configure and use devices securely," https://www.ncsc.gov.uk/collection/device-security-guidance, 2019, [Online; accessed October 6, 2022].

[20] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," National Institute of Standards and Technology, Tech. Rep. 800-207, 2020.

[21] Berkeley University of California, "Introduction to Formal Verification," https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html, 2013, [Online; accessed October 6, 2022].

[22] J. C. S. Santos, K. Tarrit, and M. Mirakhorli, "A Catalog of Security Architecture Weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 220–223.

[23] I. Kant, "Groundwork of the Metaphysic of Morals (1785)," in *Immanuel Kant: Groundwork of the Metaphysic of Morals in focus*. Routledge, 2020, pp. 17–98.

[24] HORIBA MIRA Ltd., "MISRA Compliance:2020 - Achieving compliance with MISRA Coding Guidelines," https://www.misra.org.uk/app/uploads/2021/06/MISRA-Compliance-2020.pdf, 2020, [Online; accessed October 6, 2022].

[25] Carnegie Mellon University, "SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems," https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf, 2016, [Online; accessed October 6, 2022].

[26] E. Raymond, "The Cathedral and the Bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[27] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[28] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual," https://cdrdv2.intel.com/v1/dl/getContent/671200, 2022, [Online; accessed October 6, 2022].

[29] ETSI, "TS 103 645 - Cyber Security for Consumer Internet of Things," https://www.etsi.org/deliver/etsi_ts/103600_103699/103645/01.01.01_60/ts_103645v010101p.pdf, 2019, [Online; accessed October 6, 2022].

[30] International Organization for Standardization (ISO) & International Electrotechnical Commission (IEC), "ISMS Family of Standards," https://standards.iso.org/ittf/PubliclyAvailableStandards/, 2022, [Online; accessed October 6, 2022].

[31] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[32] B. Johnson, "Fault-Tolerant Microprocessor-Based Systems," *IEEE Micro*, vol. 4, no. 06, pp. 6–21, nov 1984.

[33] D. Siewiorek, "Fault tolerance in commercial computers," *Computer*, vol. 23, no. 7, pp. 26–37, 1990.

[34] E. Dubrova, "Hardware redundancy," in *Fault-Tolerant Design*. Springer, 2013, pp. 55–86.

[35] A. Martin, A. Rashid, H. Chivers, G. Danezis, S. Schneider, and E. Lupu, "The Cyber Security Body of Knowledge. Version 1.1.0, University of Bristol," https://www.cybok.org/, 2021, [Online; accessed October 7, 2022].

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[37] J. Ménétrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, "An exploratory study of attestation mechanisms for trusted execution environments," *arXiv preprint arXiv:2204.06790*, 2022.

[38] P. England, B. Lampson, J. Manferdelli, and B. Willman, "A trusted open platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[39] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 699–716.

[40] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[41] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[42] Statista, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030," https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, 2022, [Online; accessed October 10, 2022].

[43] Gartner Inc., "Gartner Forecasts Global Devices Installed Base to Reach 6.2 Billion Units in 2021," https://www.gartner.com/en/newsroom/press-releases/2021-04-01-gartner-forecasts-global-devices-installed-base-to-reach-6-2-billion-units-in-2021, 2021, [Online; accessed October 10, 2022].

[44] C. Passerone, *Analog and Digital Electronics for Embedded Systems*. Turin, Italy: Edizioni C.L.U.T., 2015.

[45] IEEE Spectrum, "Top Programming Languages 2022," https://spectrum.ieee.org/top-programming-languages-2022, 2022, [Online; accessed October 10, 2022].

[46] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 2017, pp. 256–267.

[47] C. Bai, P. Dallasega, G. Orzes, and J. Sarkis, "Industry 4.0 technologies assessment: A sustainability perspective," *International journal of production economics*, vol. 229, p. 107776, 2020.

[48] Carnegie Mellon University, "The "Only" Coke Machine on the Internet," https://www.cs.cmu.edu/~coke/history_long.txt, [Online; accessed October 10, 2022].

[49] RSM UK Consulting LLP, "Evidencing the cost of the UK Government's proposed regulatory interventions for consumer IoT," https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/900330/Evidencing_the_cost_of_the_UK_government_s_proposed_regulatory_interventions_for_consumer_internet_of_things__IoT__products.pdf, 2020, [Online; accessed October 18, 2022].

[50] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the Mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[51] Kaspersky USA, "What is a DDoS Attack? - DDoS Meaning," https://usa.kaspersky.com/resource-center/threats/ddos-attacks, 2022, [Online; accessed October 18, 2022].

[52] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[53] Open Web Application Security Project (OWASP), "Internet of Things (IoT) Top 10 2018," https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf, 2018, [Online; accessed October 18, 2022].

[54] Carnegie Mellon University, "On Board Diagnostics: Risks and Vulnerabilities of the Connected Vehicle," https://resources.sei.cmu.edu/asset_files/WhitePaper/2016_019_001_453877.pdf, 2016, [Online; accessed October 18, 2022].

[55] N. Apthorpe, D. Reisman, and N. Feamster, "A smart home is no castle: Privacy vulnerabilities of encrypted IoT traffic," *arXiv preprint arXiv:1705.06805*, 2017.

[56] G. Vishwakarma and W. Lee, "Exploiting JTAG and its mitigation in IoT: a survey," *Future Internet*, vol. 10, no. 12, p. 121, 2018.

[57] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[58] Microsoft Learn, "The STRIDE Threat Model," https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN, 2009, [Online; accessed October 18, 2022].

[59] W. Stallings, L. Brown, M. D. Bauer, and M. Howard, *Computer security: Principles and practice*. Pearson Upper Saddle River, 2012, vol. 2.

[60] A. Djellid-Ouar, G. Cathebras, and F. Bancel, "Supply voltage glitches effects on CMOS circuits," in *International Conference on Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006.*, 2006, pp. 257–261.

[61] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[62] CWE - Common Weakness Enumeration, "2022 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2009, [Online; accessed October 19, 2022].

[63] PaX Team, "PaX Non-Executable Pages Design and Implementation," https://pax.grsecurity.net/docs/noexec.txt, 2003, [Online; accessed October 19, 2022].

[64] Microsoft Learn, "Data Execution Prevention," https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2022, [Online; accessed October 19, 2022].

[65] ARM Developer, "Execute never bits," https://developer.arm.com/documentation/ddi0360/f/memory-management-unit/memory-access-control/execute-never-bits, 2022, [Online; accessed October 19, 2022].

[66] PaX Team, "The advanced return-into-lib(c) exploits: PaX case study," http://www.phrack.org/archives/issues/58/4.txt, 2001, [Online; October 19, 2022].

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

[68] 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, 2011, pp. 121–141.

[69] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[70] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, 2012.

[71] 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.

[72] T. Kornau *et al.*, "Return oriented programming for the ARM architecture," Ph.D. dissertation, Ruhr-Universität Bochum, 2010.

[73] G. Gu and H. Shacham, "Return-oriented programming in RISC-V," *arXiv preprint arXiv:2007.14995*, 2020.

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

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

[76] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 2, pp. 139–156, 2018.

[77] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762.

[78] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[79] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[80] T. Newsham, "Exploiting Format String Vulnerabilities," http://www.madchat.fr/coding/c/c.seku/format_string/formatstring.pdf, 2000, [Online; accessed October 10, 2022].

[81] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with PACed canaries," in *Proceedings of the 4th Workshop on System Software for Trusted Execution*, 2019, pp. 1–6.

[82] Tzi-Cker Chiueh and Fu-Hau Hsu, "RAD: a compile-time solution to buffer overflow attacks," in *Proceedings 21st International Conference on Distributed Computing Systems*, April 2001, pp. 409–417.

[83] C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hely, "A red team blue team approach towards a secure processor design with hardware shadow stack," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, July 2017, pp. 57–62.

[84] 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, Sep. 2018.

[85] W. He, S. Das, W. Zhang, and Y. Liu, "BBB-CFI: lightweight CFI approach against code-reuse attacks using basic block information," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, 2020.

[86] S. Bhatkar, D. D. C, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *USENIX Security Symposium*, vol. 12, no. 2, 2003, pp. 291–301.

[87] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically Returning to Randomized lib(c)," in *2009 Annual Computer Security Applications Conference*, Dec 2009, pp. 60–69.

[88] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *International Conference on Information Systems Security*. Springer, 2009, pp. 163–177.

[89] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 447–462.

[90] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *21st Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[91] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "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.

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

[93] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[94] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size Does Matter: Why Using Gadget-Chain$Length to Prevent Code-Reuse$ Attacks is Hard," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[95] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[96] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," *Journal of Automated Reasoning*, vol. 31, no. 3, pp. 191–229, 2003.

[97] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[98] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1805–1821.

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

[100] L. Chen, J. Jiang, and D. Zhang, "Code Reuse Prevention through Control Flow Lazily Check," in *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*, Nov 2012, pp. 51–60.

[101] J. Tan, H. J. Tay, U. Drolia, R. Gandhi, and P. Narasimhan, "PCFIRE: Towards provable preventative Control-Flow Integrity enforcement for realistic embedded software," in *2016 International Conference on Embedded Software (EMSOFT)*, Oct 2016, pp. 1–10.

[102] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[103] V. v. d. Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 934–953.

[104] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[105] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

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

[107] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[108] L. Pike, P. C. Hickey, T. Elliott, E. Mertens, and A. Tomb, "TrackOS: A Security-Aware Real-Time Operating System," in *16th International Conference on Runtime Verification*. Springer, 2016.

[109] X. Hao, M. Lv, J. Zheng, Z. Zhang, and W. Yi, "Integrating Cyber-Attack Defense Techniques into Real-Time Cyber-Physical Systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 237–245.

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

[111] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon, and S. Etalle, "μShield," in *International Conference on Network and System Security*. Springer, 2017, pp. 694–709.

[112] Y. Yang, S. Zhu, W. Shen, Y. Zhou, J. Sun, and K. Ren, "ARM pointer authentication based forward-edge and backward-edge control flow integrity for kernels," *arXiv preprint arXiv:1912.10666*, 2019.

[113] Q. Technologies, "Pointer Authentication on ARMv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017, [Online; Accessed October 21, 2022].

[114] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J.-E. Ekberg, "Camouflage: Hardware-assisted CFI for the ARM Linux kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[115] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 89–106.

[116] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security analysis and enhancements of computer operating systems," National Bureau of Standards Publications (NBS), NIST, Tech. Rep. 76-1041, 1976.

[117] N. Maunero, P. Prinetto, and G. Roascio, "CFI: Control Flow Integrity or Control Flow Interruption?" in *2019 IEEE East-West Design & Test Symposium (EWDTS)*, 2019, pp. 1–6.

[118] ARM Developer, "TrustZone technology for Armv8-M Architecture," https://developer.arm.com/documentation/100690/0201/Arm-TrustZone-technology, 2022, [Online; Accessed October 21, 2022].

[119] T. Nyman, J. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 259–284.

[120] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-aware control-flow integrity using trustzone for Armv8-M," *International Journal of Parallel Programming*, vol. 49, no. 2, pp. 216–236, 2021.

[121] 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," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[122] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1219–1236.

[123] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2281–2298.

[124] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, "Control Flow Integrity Based on Lightweight Encryption Architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1358–1369, July 2018.

[125] Y. Lee and G. Lee, "HW-CDI: Hard-Wired Control Data Integrity," *IEEE Access*, vol. 7, pp. 10 811–10 822, 2019.

[126] Y. Li, Z. Dai, and J. Li, "A Control Flow Integrity Checking Technique Based on Hardware Support," in *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, Oct 2018, pp. 2617–2621.

[127] B. Gassend, D. Clarke, M. V. Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 148–160.

[128] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[129] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-Assisted Control-Flow Integrity Checking," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 458–471, Feb 2019.

[130] R. d. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. d. Bosschere, B. Preneel, B. d. Sutter, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1172–1177.

[131] A. Fu, W. Ding, B. Kuang, Q. Li, W. Susilo, and Y. Zhang, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Computers & Security*, vol. 116, p. 102666, 2022.

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

[133] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *3rd RISCV Workshop*, 2016.

[134] W. Wang, M. Liu, P. Du, Z. Zhao, Y. Tian, Q. Hao, and X. Wang, "An Architectural-Enhanced Secure Embedded System with a Novel Hybrid Search Scheme," in *2017 International Conference on Software Security and Assurance (ICSSA)*, July 2017, pp. 116–120.

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

[136] A. Chaudhari and J. A. Abraham, "Effective Control Flow Integrity Checks for Intrusion Detection," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, July 2018, pp. 1–6.

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

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

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

[140] A. De, A. Basu, S. Ghosh, and T. Jaeger, "FIXER: Flow Integrity Extensions for Embedded RISC-V," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 348–353.

[141] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The RocketChip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[142] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow checking for intrusion detection via a real-time debug interface," in *2014 International Conference on Smart Computing Workshops*, Nov 2014, pp. 87–92.

[143] L. Feng, J. Huang, J. Hu, and A. Reddy, "FastCFI: Real-time control-flow integrity using FPGA without code instrumentation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–39, 2021.

[144] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 94–105.

[145] F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, "On-chip control flow integrity check for real time embedded systems," in *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, Aug 2013, pp. 26–31.

[146] 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*, vol. 24, no. 11, pp. 3193–3207, Nov 2016.

[147] C. Spang, Y. Lavan, M. Hartmann, F. Meisel, and A. Koch, "DExIE-an IoT-class hardware monitor for real-time fine-grained control-flow integrity," in *Workshop on Design and Architectures for Signal and Image Processing (14th edition)*, 2021, pp. 26–34.

[148] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 175–183.

[149] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[150] A. Bernat and B. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011, pp. 9–16.

[151] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 575–589.

[152] Y. Guo, L. Chen, and G. Shi, "Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications," in *2018 IEEE Conference on Communications and Network Security (CNS)*, May 2018, pp. 1–9.

[153] Intel Corporation, "Control-flow Enforcement Technology Specification," https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf, 2019, [Online; accessed October 22, 2022].

[154] 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*, pp. 4–44, 2017.

[155] A. Fanti, C. Chinea Perez, R. Denis-Courmont, G. Roascio, and J.-E. Ekberg, "Towards Register Spilling Security Using LLVM and ARM Pointer Authentication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3757–3766, 2022.

[156] Microsoft Learn, "Control Flow Guard for platform security," https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard?redirectedfrom=MSDN, 2022, [Online; accessed October 22, 2022].

[157] David Weston, "Advancing Windows Security," https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE37dMC, 2019, [Online; accessed October 22, 2022].

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

[159] ARM Limited, "THUMB Instruction Set," http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf, [Online; accessed October 25, 2022].

[160] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-Oriented Programming on a Cortex-M Processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*, Aug 2017, pp. 823–832.

[161] Texas Instrument, "Tiva™ C Series TM4C123GH6PM Microcontroller Data Sheet," https://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf, [Online; accessed October 25, 2022].

[162] ——, "SW-TM4C TivaWare™ for C Series (Complete)," https://www.ti.com/tool/SW-TM4C, [Online; accessed October 25, 2022].

[163] V. Eftekhari Moghadam, P. Prinetto, and G. Roascio, "Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems," in *2022 IEEE European Test Symposium (ETS)*, 2022, pp. 1–4.

[164] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[165] Google Code Archive, "Project ardupilot-mega," https://code.google.com/archive/p/ardupilot-mega/, [Online; accessed October 26, 2022].

[166] OSTechNix, "UnixBench – A Benchmark Suite For Unix-like Systems," https://ostechnix.com/unixbench-benchmark-suite-unix-like-systems/, [Online; accessed October 26, 2022].

[167] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[168] Embedded Microprocessor Benchmark Consortium, "CoreMark-PRO CPU Benchmarks," https://www.eembc.org/coremark-pro/, [Online; accessed October 26, 2022].

[169] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[170] "FreeRTOS - Real-time operating system for microcontrollers," https://www.freertos.org/, 2022, [Online; accessed October 26, 2022].

[171] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, p. 30–50, feb 2000. [Online]. Available: https://doi.org/10.1145/353323.353382

[172] I. E. T. F. (IETF), "Rfc 2753 - a framework for policy-based admission control," https://datatracker.ietf.org/doc/html/rfc2753, 2000, [Online, accessed April 28th, 2022].

[173] A. De Lucia, "Program slicing: methods and applications," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 142–149.

[174] U. Erlingsson, *The inlined reference monitor approach to security policy enforcement*. Cornell University, 2004.

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

[176] A. Developer, "Arm Cortex-M4 Processor Technical Reference Manual Revision r0p1," https://developer.arm.com/documentation/100166/0001/?lang=en, 2020, [Online, accessed November 3, 2022].

[177] M. Rajabalipanah, M. S. Roodsari, Z. Jahanpeima, G. Roascio, P. Prinetto, and Z. Navabi, "AFTAB: A RISC-V Implementation with Configurable Gateways for Security," in *2021 IEEE East-West Design & Test Symposium (EWDTS)*, 2021, pp. 1–6.

[178] R.-V. International, "RISC-V ISA - Volume I Unprivileged ISA," https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, 2019, [Online, accessed November 3, 2022].

[179] V. Forte, N. Maunero, P. Prinetto, and G. Roascio, "PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity," in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2021, pp. 1–6.

[180] Radare2 Team, "Radare2 GitHub repository," https://github.com/radareorg/radare2, 2017, [Online, accessed November 4, 2022].

[181] ——, "R2pipe repository," https://github.com/radareorg/radare2-r2pipe, 2022, [Online, accessed November 4, 2022].

[182] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *The 10th Network & Distributed System Security Symposium 2003 (NDSS), San Diego, California, USA*. Internet Society, 2003, p. 149.

[183] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[184] S. Zhou and J. Chen, "Experimental Evaluation of the Defense Capability of ARM-based Systems against Buffer Overflow Attacks in Wireless Networks," in *2020 IEEE 10th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2020, pp. 375–378.

[185] G. Roascio, G. Serra, and V. Eftekhari Moghadam, "Em-RIPE: Runtime Intrusion Prevention Evaluator for ARM Microcontroller Systems," in *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2022, pp. 1–6.

[186] D. J. Day, Z. Zhao, and M. Ma, "Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems," in *2010 Fourth International Conference on Digital Society*, 2010, pp. 172–177.

[187] A. Developer, "ARM Compiler for Embedded User Guide Version 6.18," https://developer.arm.com/documentation/100748/0618/, 2022, [Online; Accessed November 5, 2022].

[188] G. Roascio, G. Costa, E. Baccelli, L. Malina, R. Matulevičius, M. Momeu, N. Morkevičius, E. Russo, B. Stojanović, and A. Tasidou, "HArMoNICS: High-Assurance Microgrid Network Infrastructure Case Study," *IEEE Access*, vol. 10, pp. 115 372–115 383, 2022.

[189] I. C. W. Study, "A Resilient Cybersecurity Profession Charts the Path Forward," https://www.isc2.org//-/media/ISC2/Research/2021/ISC2-Cybersecurity-Workforce-Study-2021.ashx, 2021, [Online; accessed November 10, 2022].

[190] ISACA, "State of Cybersecurity 2021 - Part 1: Global Update on Workforce Efforts, Resources and Budgets," https://securitydelta.nl/media/com_hsd/report/424/document/state-of-cybersecurity-2021-part-1.pdf, 2021, [Online; accessed November 10, 2022].

[191] S. Bratus, "What Hackers Learn that the Rest of Us Don't: Notes on Hacker Curriculum," *IEEE Security & Privacy*, vol. 5, no. 4, pp. 72–75, 2007.

[192] A. Anil Yasin and A. Abbas, "Role of gamification in engineering education: A systematic literature review," in *2021 IEEE Global Engineering Education Conference (EDUCON)*, 2021, pp. 210–213.

[193] C. Eagle, "Computer security competitions: Expanding educational outcomes," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 69–71, 2013.

[194] "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')," https://cwe.mitre.org/data/definitions/78.html, 2022, [Online; accessed November 10, 2022].

[195] "CWE-89: Neutralization of Special Elements used in an SQL Command ('SQL Injection')," https://cwe.mitre.org/data/definitions/89.html, 2022, [Online; accessed November 10, 2022].

[196] "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')," https://cwe.mitre.org/data/definitions/79.html, 2022, [Online; accessed November 10, 2022].

[197] "CWE-345: Insufficient Verification of Data Authenticity," https://cwe.mitre.org/data/definitions/345.html, 2022, [Online; accessed November 10, 2022].

[198] KasperskyLab, "Threat landscape for industrial automation systems," https://ics-cert.kaspersky.com/reports/2019/03/27/threat-landscape-for-industrial-automation-systems-h2-2018/, [Online; accessed November 10, 2022].

[199] P. Prinetto, G. Roascio, and A. Varriale, "Hardware-based capture-the-flag challenges," in *2020 IEEE East-West Design & Test Symposium (EWDTS)*, 2020, pp. 1–8.

[200] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, p. 6, 2016.

[201] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2013, ch. 14.

[202] "1149.1-2013 - IEEE Standard for Test Access Port and Boundary-Scan Architecture," https://standards.ieee.org/standard/1149_1-2013.html, 2013, [Online; accessed November 10, 2022].

[203] "IEEE 1500 Standard for Embedded Core Test (SECT)," http://grouper.ieee.org/groups/1500/index.html, 2005, [Online; accessed November 10, 2022].

[204] C. Domas, "Hardware backdoors in x86 CPUs," *Black Hat*, pp. 1–14, 2018.

[205] Q. Shi, H. Wang, N. Asadizanjani, M. M. Tehranipoor, and D. Forte, "A comprehensive analysis on vulnerability of active shields to tilted microprobing attacks," in *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, Dec 2018, pp. 98–103.

[206] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[207] National Institute of Standards and Technology, "Cyber Ranges," https://www.nist.gov/system/files/documents/2018/02/13/cyber_ranges.pdf, [Online; accessed November 11, 2022].

[208] M. M. Yamin, B. Katt, and V. Gkioulos, "Cyber ranges and security testbeds: Scenarios, functions, tools and architecture," *Computers & Security*, vol. 88, p. 101636, 2020.

[209] K. Barnes and B. Johnson, "National scada test bed substation automation evaluation report," Idaho National Laboratory (INL), Tech. Rep. No. INL/EXT-09-15321, 2009.

[210] T. Morris, A. Srivastava, B. Reaves, W. Gao, K. Pavurapu, and R. Reddi, "A control system testbed to validate critical infrastructure protection concepts," *International Journal of Critical Infrastructure Protection*, vol. 4, no. 2, pp. 88–103, 2011.

[211] P. Čeleda, J. Čegan, J. Vykopal, D. Tovarňák *et al.*, "Kypo–a platform for cyber defence exercises," *M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence. NATO Science and Technology Organization*, 2015.

[212] E. Keller and J. Rexford, "The "Platform as a Service" Model for Networking," *INM/WREN*, vol. 10, pp. 95–108, 2010.

[213] I. M. Moraes, D. Mattos, L. H. Ferraz, M. E. Campista, M. Rubinstein, L. .Costa, de M. Amorim, P. V. O. C. Duarte, and G. Pujolle, "FITS: A flexible virtual network testbed architecture," *Computer Networks*, vol. 63, pp. 221–237, 2014.

[214] I. D. Alvarenga and O. C. Duarte, "RIO: A denial of service experimentation platform in a Future Internet Testbed," in *2016 7th International Conference on the Network of the Future (NOF)*. IEEE, 2016, pp. 1–5.

[215] C. Willems and C. Meinel, "Online assessment for hands-on cyber security training in a virtual lab," in *Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2012, pp. 1–10.

[216] M. Mallouhi, Y. Al-Nashif, D. Cox, T. Chadaga, and S. Hariri, "A testbed for analyzing security of SCADA control systems (TASSCS)," in *ISGT 2011*, 2011, pp. 1–7.

[217] J. Hong, S. Wu, A. Stefanov, A. Fshosha, C. Liu, P. Gladyshev, and M. Govindarasu, "An intrusion and defense testbed in a cyber-power system environment," in *2011 IEEE Power and Energy Society General Meeting*, 2011, pp. 1–5.

[218] A. Furfaro, L. Argento, A. Parise, and A. Piccolo, "Using virtual environments for the assessment of cybersecurity issues in IoT scenarios," *Simulation Modelling Practice and Theory*, vol. 73, pp. 43–54, 2017, smart Cities and Internet of Things. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X16302374

[219] A. Furfaro, A. Piccolo, D. Saccà, and A. A. Parise, "A virtual environment for the enactment of realistic cyber security scenarios," in *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*. IEEE, 2016, pp. 351–358.

[220] P. Tsai and C. Yang, "Testbed TWISC: A network security experiment platform," *International Journal of Communication Systems*, vol. 31, no. 2, p. e3446, 2018.

[221] J. A. Mattson, "Cyber defense exercise: A service provider model," in *IFIP World Conference on Information Security Education*. Springer, 2007, pp. 81–86.

[222] G. Berra, G. Ferraro, M. Fornero, N. Maunero, P. Prinetto, and G. Roascio, "PAIDEUSIS: A Remote Hybrid Cyber Range for Hardware, Network, and IoT Security Training," in *ITASEC - Italian Conference on Cybersecurity*, 2021, pp. 284–297. [Online]. Available: http://ceur-ws.org/Vol-2940/paper24.pdf

[223] S. Ahmad, N. Maunero, and P. Prinetto, "EVA: A Hybrid Cyber Range," *CEUR Workshop Proceedings*, vol. 2597, pp. 12–23, 2020.

[224] G. Roascio, S. Y. Cerini, and P. Prinetto, "Remotizing and Virtualizing Chips and Circuits for Hardware-based Capture-the-Flag Challenges," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 477–485.

[225] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman," *Cryptography*, vol. 4, no. 2, p. 15, 2020.

[226] P. Lipton, C. Lauwers, M. Rutkowski, C. Noshpitz, and C. Curescu, "TOSCA Simple Profile in YAML Version 1.3," https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.pdf, OASIS, Tech. Rep., February 2020.

[227] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[228] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements," *Requirements Engineering*, vol. 16, no. 1, pp. 3–32, Mar 2011. [Online]. Available: https://doi.org/10.1007/s00766-010-0115-7

[229] S. Marksteiner, V. J. Exposito Jimenez, H. Valiant, and H. Zeiner, "An overview of wireless IoT protocol security in the smart home domain," in *2017 Internet of Things Business Models, Users, and Networks*, 2017, pp. 1–8.

[230] K. Hofer-Schmitz and B. Stojanović, "Towards formal verification of IoT protocols: A Review," *Computer Networks*, vol. 174, p. 107233, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128619317116

[231] European Union, "Regulation (EU) 2016/679 of the European Parliament (General Data Protection Regulation)," https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016, [Online; accessed November 12, 2022].

[232] "DPO Tool - Evaluate Business Process," https://dpotool.cs.ut.ee/, 2020, [Online; accessed November 12, 2022].

[233] F. De Rosa, N. Maunero, P. Prinetto, F. Talentino, and M. Trussoni, "ThreMA: Ontology-Based Automated Threat Modeling for ICT Infrastructures," *IEEE Access*, vol. 10, pp. 116 514–116 526, 2022.