POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering
Master Degree Thesis

Design of Remote Service Infrastructures for Hardware-based Capture-the-Flag Challenges

Authors: Luca Marongiu, Mauro Perra

Supervisor: Paolo Ernesto Prinetto

December, 2021

Abstract

In recent years, digital technologies have surrounded us in all aspects of everyday life. With such a large amount of data produced and exchanged, the cybersecurity topic has become fundamental, given the strong demand for protecting sensitive information about our private sphere. The rapid growth of this process led to a request for security experts which is certainly oversized if compared to the current available workforce. For this reason, the community tries to involve and train as many people as possible, sometimes experimenting with innovative learning methods that are oriented to gaming and point achievement. This is the case of Capture-the-Flag (CTF) competitions, where participants are asked to deal with real practical examples related to IT security issues, which embed the solution to be achieved through known cyber attack (or defense) techniques.

In the current panorama, most of the CTF challenges are more focused to software or network security problems, mainly because they are well supported by a consolidated research branch. Unfortunately, the same does not occur for hardware security, which has emerged in all its importance only in very recent times. In fact, the hardware is at the base of any computing system, and if the security of the electronic components of the systems is not addressed, this can result in the possible ineffectiveness of the protections applied in the overlying software layers.

To provide adequate awareness and education for the security threats to which these components are subject, the related CTF challenges should targeting participants with certain skills in the hardware domain, such as knowledge of hardware description languages, or basics of digital hardware design and synthesis. Furthermore, a knowledge of the most common hardware vulnerabilities is required to challengers for solving the challenges.

The purpose of this thesis is to help filling the hardware-based challenges gap by developing two environments capable of offering hardware-based challenges as remote services. The environments designed in this thesis exploit the usage of physical hardware devices connected to remote machines, or Electronic Design Automation (EDA) tools for simulating described hardware.

The document provides an overview of CTF competitions and the current lack of hardware challenge offerings in major competitions. After that, a description of the service architecture and its possibilities is offered. Significant practical examples

of use of the platform are also presented, together with the first experimental data related to the impact from the point of view of resources.

The thesis is the result of a joint work between Luca Marongiu who wrote Chapters 1, 2, 5 and appendices A, C, and Mauro Perra who wrote Chapters 3, 4, 6, 7 and appendices B, D.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the recent *technology boost*, human daily life has to deal with a multitude of electronic devices which produce and exchange a lot of sensitive information. In order to handle with care such a kind of data, it is clear that people must trust devices, but their increasing complexity could make them more and more vulnerable and prone to be attacked from malicious people. For this reason, computing systems must run reliable and secure software capable to manage data ensuring confidentiality, integrity and availability of the services.

With such a scenario, the issue of data security must be addressed at every level, from the awareness-raising of any citizen, up to the massive investments by public and private sectors, in order to increase the number of available experts in cybersecurity. The demand for experts in research and development job positions in cybersecurity has grown by 350% in the last 8 years [9]. Competence centers such as schools and universities are certainly making significant efforts to help fill this gap, even if very often there is an overly theoretical approach to the topic of IT security. This tends to compromise the appeal that the subject has towards young students.

A promising direction for teaching hard practical skills on cybersecurity seems to be given by the so-called *gamification* [10] [15]: students are asked to directly face security problems by solving riddles and challenges related to the breakdown or the decryption of software, communication systems or devices, or by implementing countermeasures to prevent attacks by opposing teams. *Capture-The-Flag* (CTF) challenges are definitely the corner stones, as proved by the high number of events, competitions, and training courses that rely on them. In these events, the participants are confronted directly with games and riddles related to practical problems of hacking, cyber-attack, and cyber-defense. The main goal is to extract from the challenge a unique string, the *flag*, which certifies the success.

The challenge-based learning method has been shown to be valid by several studies [3] [19] [16]. CTF competitions are especially important for their ability to make participants develop their *adversarial thinking*, which is essential to be able

to face future real threats [17].

At the time of writing, it is possible to notice how the security problems mainly offered to participants are related to cryptography, software or networks. This is due to the fact that these security topics are very popular, and related issues have certainly been studied more in depth and for a longer time. Actually, this prevalence is not at all justified by a total photograph of reality, where there are security problems also related to the hardware and the physicality of the components [20]. These can have even more dramatic consequences than the classic vulnerabilities, since with a compromised hardware, security measures of the above layers may lose value and effectiveness [1].

In relation to this, it is evident that *hardware-based CTF challenges* assume a crucial role for the study of the related vulnerabilities, and the possible attacks used to exploit hardware devices. In fact, many skills can be required to solve an hardware-based challenge, from the knowledge of hardware description languages, digital hardware design and synthesis, to FPGA programming, EDA-tools commands and many more.

Universities and companies started to organize CTF competitions at different levels of resonance (national or international). In this direction, the project *CyberChallenge.IT*[1] was born. CyberChallenge.IT is organized by the CINI Cybersecurity National Laboratory[2], and is the main Italian training program for cybersecurity, targeting people from 16 to 24 years old. The project is based on a highly technical training offer of several months aimed at the main Italian universities, which focuses on all the main topics of cybersecurity, and which prepares participants for the two final CTF events that elects the winning team (see Chapter 3).

Given the importance of offering adequate preparation for challenges based on hardware devices, the Cybersecurity National Laboratory has set up *PAIDEUSIS* [2], a hybrid cyber range hosted at LINKS Foundation[3] in the Politecnico di Torino campus. PAIDEUSIS is a hybrid training environment that seeks to combine the advantages of virtualization and scalability with the realism of hardware devices physically present and connected to the cyber range, including a wide range of devices such as IoT, industrial control, and network hardware devices. The work of this thesis is part of the development of the aforementioned cyber range, and of scenarios based on vulnerable digital hardware.

What has been done in this thesis work concerns the development of two environments capable to host remote hardware CTF. The environments have been designed in such a way to embed challenges with a generic hardware description,

[1] https://cyberchallenge.it/

[2] https://cybersecnatlab.it/

[3] https://linksfoundation.com/

easily configurable from an organizational point of view. The first environment (*EDA-tool-based*) exploits a professional hardware simulation tool in order to offer the interaction between the CTF participant and the vulnerable system. Here, a software filter that wraps the entire environment is needed to just allow a predefined set of input commands. The second environment (*Hardware-device-based*) uses instead real physical devices with the vulnerable hardware flashed inside. This modality takes advantage of reprogrammable circuits to match the genericity feature of the system.

With such infrastructures, many hardware learning outcomes can be addressed, strictly depending on the environment. Many considerations also from a resources occupation point of view can be done, taking into account that, for the first environment, an instance of a hardware simulator for every participant must run in the server that hosts the competition, while for the second one, a physical instance of the device is needed. The strength of this platform resides in the possibility to host different and generic challenges, with the focus of design and solution only on the hardware. More important, it brings the advantage of avoiding the logistic problem of distributing a huge amount of devices all around the nation and abroad, since it supports remote connections from the participants, who can play challenges without having the vulnerable hardware directly on their hands.

The remainder of the document is organized as follows: Chapter 2 presents background details about *Capture-the-Flag* competitions; Chapter 3 gives context details about the PAIDEUSIS Cyber Range and the CyberChallenge.IT training platform. Then, a state-of-the-art review on hardware CTF competitions is reported in Chapter 4. Chapters 5 and 6 respectively present the two environments of the described infrastructure, outlining structure and behavior and also reporting their reachable learning outcomes and resources occupation. Finally, Chapter 7 summarizes the thesis work and exposes possible future improvement directions.

Chapter 2

Background: Capture-the-Flag Competitions

A *Capture-the-Flag* (CTF) event is a competition where the participants have to solve a computer security problem in order to retrieve a unique string chosen by the organizers, the *flag*, which is embedded into the *subject* of the challenge: a cipher, a software, a website, a network protocol, etc. [12]. For example, challengers are required to exploit SQL injection [5] or Cross-Site Scripting (XSS) [4] over a vulnerable web page, or there may be *step-based* challenges where the interaction with a command line is offered by a vulnerable system that hides the flag, for example, in the home folder of some user whose login needs to be cracked, or in some software to be attacked through code injection [6]. Challenges can be broadly categorized depending on topics as follows:

- *Binary*: challenges requiring to deal with a vulnerable software application. The abstraction level examinated during participants here is mostly the machine binary code, with a major use of disassembly and debugging tools;
- *Web*: challenges dealing with vulnerable web services, which can be cracked exploiting command or code injections to retrieve information that is originally not accessible, including the flag. Examples include challenges based on web login crack, malicious SQL query injections, tampering with cookies, etc.;
- *Crypto*: challenges consisting in thwarting an encryption scheme to decipher a message that directly or indirectly contains the flag. The encryption algorithm can be both a classic one but vulnerable, and a brand new one to be cracked. Mathematical knowledge of combinatorics, prime numbers, modular arithmetic is here required;

- *Forensics*: challenges that lead participants to mimic the typical forensic approaches adopted by law enforcement and investigation agencies. They mostly include steganography, malformed files, packet captures, .jpg or .png files modified to hide texts or executable pieces of code, etc.;
- *Networking*: challenges against vulnerable network systems, maily requiring actions such as breaking firewalls, deceiving access policies, attempting spoofing attacks and poisoning of network protocols, or reconstructing a message from individual packets;
- *Miscellaneous*: challenges based on different topics, sometimes even non-technical ones and whose resolution comes with applying the basics of logic;
- *Hardware*: challenges that represent security problems related to digital hardware, and whose resolution requires knowledge of hardware design, synthesis, test and maintenance at different levels of abstraction, as defined in [21].

The capture of the flag guarantees the participants the acquisition of an amount of points, which are used to compose a final score, and which depends on the difficulty of the challenge itself. Usually, the organizers plan to give the participants a certain number of *hints*, to be released along the execution time of the challenge. Those who decide to benefit of the proposed hints are subtracted a certain number of points from the total amount.

Mostly, challenges within a competition are many and of different topics, not to give an objective advantage to experts in one or another field. In addition to their topic, there are also different ways in which challenges are carried out. Below here, the most famous categories are highlighted:

- *Jeopardy*: Participants are alone in front of the challenge subject, which hides the flag. The flag can be taken by exploiting the vulnerabilities that have been artificially inserted into the system by the competition organizers. Participants may be grouped in teams, but there is no interaction among the teams, and the only opponent is the challenge itself;
- *Attack/Defense*: Participants are grouped in teams and each team is given an instance of a system injected with several vulnerabilities. All the instances have the same vulnerabilities and are connected to a same network. The competition includes 2 phases: in the first phase (e.g., couple of hours), each team can access its instance only, and it should identify and fix the vulnerabilities on its own instance. In this way, they prevent other teams from capturing their flag exploiting these vulnerabilities during the next phase. During the next phase, connection is opened and each team is free to access the instances of the opponent teams and capturing their flags if the vulnerabilities present in their instances have not been properly patched during the first phase. Points

are awarded based on three factors: (i) the number of flags captured on the instances of other teams (*attack points*), (ii) the number of flags stolen by other teams from your instance (*defense points*), (iii) the percentage of time the services remain up and work properly (*SLA points*). With respect to Jeopardy-style, these challenges allow participants to gain experience on both offensive and defensive skills;

Challenges can be further characterized by the *execution mode* with which they are carried out. Usually, the participants are directly put in contact with the software or the file that contains the flag. In case the subject is, for example, a vulnerable network protocol or a heterogeneous system composed of several parts, the interaction takes place with a *simulator software* that emulates its behavior.

This can be more articulated if talking about hardware challenges, which are the subject of this thesis. In fact, the concept of execution mode can be further formalized here, by distinguishing some categories as follows [21]:

- **By-hand**: Participants are given a static representation of a digital hardware (such as HDL code or RTL/gate-level schematics), and the challenge can be solved manually with analyzing the provided description, without mandating the use of any particular tool. An example follows: participants are given the HDL code of a Control Unit where one *opcode* of an instruction is not documented, and the participants have to identify it. The flag can be the opcode itself. This modality gives the opportunity to learn the target HDL code, and the skill of reverse-engineering in a digital circuit description, from a structural (better in terms of learning) or behavioral description to the real behavior of the circuit.

- **EDA-tool-based**: this second modality resorts to some major *Electronic Design Automation* tools to simulate a model of an hardware design described in some HDL code. This type of challenge allows the participant to solve it even remotely, e.g., by connecting with a TCP connection to the service hosted in a server. An example can be a design simulated with *ModelSim*, that if stimulated with certain inputs, can give as output the flag in binary representation. The EDA-tool-based modality teaches the target HDL code of the design, reverse-engineering skills, but also the most important commands of an EDA tool. Participants may use scripting languages (e.g., *Python*[1]) in order to automate the sequence of commands sent to the service. Is important to underline that this modality may need to develop custom *wrappers* designed in order to prevent the participant to use some powerful commands of the EDA tool, that may allow him to retrieve the flag in a easier way (i.e., out of the scope of the challenge);

[1] https://www.python.org/

- **Hardware-device-based**: the third modality is the most realistic one, but it differs from the previous in the learning outcomes and many other characteristics, as it resorts to physical hardware devices to host the hardware design of the challenge, e.g., on FPGA circuits or other configurable hardware devices. Participants use a custom software written by the organizers in order to communicate with the hardware device, and use its synthesizable HDL version to study its behavior. Then, they may try to individuate the vulnerabilities of the system by attempting some interaction with it. Despite a physical hardware device is needed, the challenges of this type can be executed also in remote, giving the possibility to connect to the service by an online connection. The learning outcomes here range from studying the target HDL code of the design, reverse-engineering skills to synthesis of digital hardware and knowledge of FPGA technology. Nevertheless, this mode introduces severe issues in term of scalability, since during the competition each participant (or team of participants) must be given a different instance of the hardware device, regardless the competition type.

In this thesis, the last two modalities (that have the highest development requirements) have been explored extensively. As for the EDA-tool-based mode, a complete environment that uses the ModelSim hardware simulation software has been designed and developed. It supports the offering of generic VHDL designs, and the participants can interact with the subject by submitting a subset of ModelSim commands filtered with a Python wrapper. With this modality, the Jeopardy challenges are fully supported. The VHDL description of the hardware design is provided to all the participants, except for the flag information, which is masked.

The second mode explored is the hardware-device-based, for which two versions of interaction platforms have been developed: one for Jeopardy challenges and one for Attack/Defense challenges. Both versions uses the SE*cube*™ boards[2] that hosts the vulnerable hardware design inside the FPGA of the chip. The decision of developing Attack/Defense CTFs only for this second type of environment has been taken in order to provide a more realistic scenario to the game involving real hardware devices. Furthermore, the patch phase consists here not only in writing the VHDL of the design, but also to synthesize it with respecting the timing constraints. This last concept is an added value to the learning outcomes that the challenge has, and can be addressed only with the hardware-device-based challenges.

[2] https://www.secube.blu5group.com/

Chapter 3

The Scenario: CyberChallenge.IT and PAIDEUSIS Hybrid Cyber Range

To date, the Italian panorama related to cyber education sees the CINI Consortium[1] (namely "*Consorzio Interuniversitario Nazionale per l'Informatica*") as a major contributor, with its *Cybersecurity National Laboratory*[2]. The Laboratory acts as a collector for the main cybersecurity competence centers in Italy, with the main goal of offering its knowledge to public companies and institutional decision makers.

Beyond its specific consultancy activity for the public sector, since 2017 the Laboratory is the creator and maintainer of the *CyberChallenge.IT* platform, which represents the main Italian training program in cybersecurity for high-school and undergraduate students from main Italian universities.

3.1 CyberChallenge.IT

CyberChallenge.IT is a training program for young talents with great passion, commitment and curiosity in the scientific-technological disciplines. It is the leading Italian initiative to identify, attract, recruit and place the next generation of cybersecurity professionals.

The program combines a traditional training activity with a gamification-oriented

[1] https://www.consorzio-cini.it/index.php/it/

[2] https://cybersecnatlab.it/?lang=en

approach that translates into participation in online competitions that simulate scenarios of networks and real working environments where logical, problem-solving, communication skills and, above all, a great desire to learn are also needed [8].

The training phase aims to provide the methodological and practical basis required to analyze vulnerabilities and possible attacks, identifying the most suitable solutions to prevent them, in different areas of cybersecurity. In particular, it is organized in the following thematic areas: Ethics, Cryptography, Web Security, Software Security, Hardware Security, Network Security, Malware Analysis, and Attack/Defense. All the module inserted inside *CyberChallenge.IT* cover all the threats that come from cyberspace.

Inside the project, Hardware Security area has been recently added, due to the growing attention it is creating in community. In the recent past years, such a topic has been treated with superficiality, but it plays a fundamental role, even with respect to the national security. An example could be done considering the fact that most of microchips produced in the world is made by few companies in different countries. It might be realistic that some of those companies perform malicious actions like inserting backdoor inside circuits, with the aim to damage nations that include such devices inside their critical environments.

To include the hardware security in the *CyberChallenge.IT* gamification-oriented training, the PAIDEUSIS Hybrid Cyber Range [2] has been set up in Turin to host challenges based on simulated or real hardware.

3.2 PAIDEUSIS Hybrid Cyber Range

According to NIST, a cyber range is a "*interactive, simulated representation of an organization's local network, system, tools, and applications that are connected to a simulated internet level environments*" [7].

PAIDEUSIS is a *hybrid* cyber range, as it is made up of more than just virtualized components. Its main objective is to offer users the possibility to interact with real devices, real hardware components that spans from IoT boards to networking appliances, even without the physical proximity to the hardware or its possession. Despite is difficult to entirely replace the interaction with real hardware devices, it is impossible to offer a sufficient variety of cybersecurity training scenarios resorting exclusively to real hardware, this because of the huge costs related to the will of building an infrastructure based on real hardware components.

Because of these limitations, PAIDEUSIS is not just hardware. It also offers cybersecurity training scenarios that are implemented using the endless possibilities of hardware emulation to virtualize different devices and components.

Considering fully hardware-based and fully virtualized environments, PAIDEUSIS offers the best of both worlds. The Cybersecurity National Laboratory is committed to offer as many scenarios as possible that are backed up by real hardware,

real components. Whenever this is not feasible, PAIDEUSIS can still achieve the goal by resorting to components virtualization and emulation.

PAIDEUSIS architecture acknowledges some fundamental entities, which are listed below [2]:

- The smallest entity is the **component**: an element, that could be virtual (software or simulated hardware) or real (hardware-devices), that makes up the cyber range;
- The set of component is called **subnet**, where a group of components are interconnected using LAN (Wi-fi or Ethernet), or PAN (USB, Bluetooth), or through virtual interfaces in case the devices are virtualized;
- The set of one or more subnets is called **range** properly (or **theater**), that is aimed to host compatible and coherent scenarios. A range may need more subnets to be set up, and several ranges can use same subnet of components;
- A **scenario** is a particular setting of a range, that represents a story-telling which players are into;
- When a team (or a user) interacts with a particular scenario, he or she creates a **session** (i.e., a single instance of an interaction). Sessions are usually *stateless* over time, meaning that no progress from users is kept between two different sessions.

The present thesis aims to develop relevant parts of 2 of the PAIDEUSIS ranges, which are:

- **Device-Based Hardware Security Range**: designed to host training and gaming scenarios related to security issues of hardware devices. Exercises are thus based on the presence of physical devices with vulnerabilities. These vulnerable devices are installed inside the infrastructure of PAIDEUSIS and, by definition, they are not reachable from users who are outside the infrastructure. In order to communicate with the vulnerable devices, users connect to the cyber range public services, working as proxies for exploiting the vulnerable devices. At the moment, the vulnerable devices are represented exclusively by instances of the SE*cube*™ hardware security platform[3]. Despite having only one type of device, the internal hardware of the SE*cube*™ can be leveraged to inject several vulnerabilities, such as:
  - *Hardware vulnerabilities* inherent to the logical domain, where the circuit is physically implemented on the SE*cube*™ FPGA;

[3] https://www.secube.blu5group.com/

  - *Software vulnerabilities* affecting the code that is physically executed on the microcontroller located inside the device;
  - *Hybrid vulnerabilities* without a precise location but concerning the device in its entirety;

- **Hardware Simulation Security Range**: offers scenarios for training and gaming about security issues of fully-emulated hardware devices. Simulation is done using the professional industrial tool *ModelSim*;

The basic idea behind PAIDEUSIS is to work with a system that is able to host multiple scenario and change it at need, using a *scenario orchestrator*.

Orchestrating a scenario means building it from basic blocks, connected to create the final result. The orchestration also includes the possibility to modify, backup and delete scenarios. Ideally, this should lead to the implementation of a GUI and of a repository of configurations. The GUI is used to manage the available scenarios and to create new ones, while the repository is used to fetch configurations for specific devices (hardware or emulated) that are needed in a given scenario.

This is a clever way to speed up the creation of scenarios and to reduce the learning curve of the personnel of PAIDEUSIS. The orchestration of a scenario includes many operations, such as:

- flashing a specific firmware on a hardware device;
- setting up dedicated networking rules on physical and emulated network appliances;
- setting up dedicated networking rules on physical and emulated network appliances;
- setting up VMs with operating systems configured in a very specific way, according to the needs of the users;
- etc.

In this thesis, the design of two different environments have been developed with the aim to host generic vulnerable hardware component for CTF competitions inside the *Device-Based Hardware Security* Range and *Hardware Simulation Security* Range.

Chapter 4

Related Work

For the reasons already discussed, software and network security issues have mostly been addressed by the CTF competitions, leaving the hardware security topic apart. Very often, hardware is believed to be just subject to *failures*, and not to *attacks* in the common sense of the term [20]. On the contrary, it is widely known that hardware is the base of every computing system, and a lack of attention to its security problems can lead up to the invalidation of the protections applied to the software layers running above, although extensively tried or formally verified [1].

To the authors' best knowledge, the main CTF events currently offer a limited range of hardware CTF. Some of the most important organizations that worldwide acknowledge hardware-related challenges are:

- The *Hardwear.io*[1] platform, which organize courses, conferences, webinars related to hardware security and organize CTF competition since 2017. The challenges cover several arguments like RFID, automotive Bluetooth components, side-channel analysis, (de)soldering, and radio;
- *Riscure*[2], an important security evaluation laboratory specialized on embedded system and IoT security. From the hardware CTF point of view, this laboratory organized the *RHme* ("Riscure Hack me") event from 2015 to 2018, focused on the use of Arduino™[3] products for the implementation of the challenges [23] [24] [25];
- The *Hack*@DAC [14] hardware security contest, held within the Design Automation Conference (DAC)[4] since 2017. It is a competition focused on the

[1] https://hardwear.io/

[2] https://www.riscure.com/

[3] https://www.arduino.cc/

[4] https://www.dac.com/

topic of micro-architectural and side-channel flaws in chips. Participating teams (student and industrial as well) are given a design of a vulnerable chip to be studied before competition. The aim is to identify the greatest number of security problems. The winners of this first phase then participate in the CTF competition held live at the conference: here, the teams are assigned a new design of a vulnerable SoC. At the end, the winner is the team that reported the greatest number of problems in the design, under the format of flags;

- The *Google Capture The Flag* [13] event, which introduced some hardware-oriented challenges as well. In the 2017 edition, a challenge which consisted in cracking a slot machine was proposed, requiring to physically connect to the pins of the Arduino™board which controlled the machine in order to extract the flag. Other challenges that required to reverse HDL code or schematic hardware components were included in the 2018, 2019 and 2020 editions;

- The *Reply Challenges* [22] from Reply[5]. In 2021 edition, there has been a miscellaneous section of challenges, in which one of those was including a logic net generating an output that could be used to access the flag. This challenge could be resolved by hand or writing the HDL code and simulating the hardware model.

From this state of the art review, it can be noted that there is a limited number of events that hosts also challenges that consider *strict* hardware security problems, i.e., where the design of an hardware is analyzed and then the vulnerability could be exploited simulating the hardware or using a real device with a vulnerable component inside.

The problem may be due to the difficulty of getting all the participants to obtain hardware devices or to find a system capable of hosting hardware-based challenges that do not need to be redesigned every time the scenario is changed. In fact, producing a standard and safe system capable of supporting generic hardware components is itself challenging.

Furthermore, in major CTF competitions, it seems very often that great importance is given to the ability of participants in exploiting an interface to break into the system and get the control, even when it comes to hardware systems. In the authors' opinion, there is a gap to fill on how to consider the participants' ability to address the security problems within the challenge target, independently of the used interface. In this regard, the authors of this work refer to the definition of hardware-based challenge given in [21], and the thesis work aims to offer the generic participant this type of experience, no matter whether it is a Jeopardy challenge or inside an Attack/Defense context.

[5] https://www.reply.com/en/

Chapter 5

EDA-tool-based Environment

5.1 Motivations

A hardware-based CTF is defined as a challenge in which the participant can reach the solution just through their knowledge in the hardware technology domain [21]. For this reason, a well structured and hard-to-break environment is needed: the challenger must be unable to attack the software that surrounds and "offers" the hardware, but he must be focused on the hardware design whose description is given by the organizers. Thus, the challenge subject is the vulnerable hardware design, and within the challenge itself it can take any form.

As for the challenges solvable by hand, their essence is the hardware description language (HDL) code or schematic, delivered to participants by the challenge creators. As already described, this type of challenges is the one with lowest development requirements, as there is no environment to be used by the participants: they must analyze the material and find the flag by inspection or by simulating the target. This modality is also the least interactive: as the challengers are in front of a static representation of the vulnerable system, and the most of the effort is put on HDL understanding and reverse-engineering.

Another type of challenges can be introduced to get a higher degree of interaction, even without resorting to real hardware devices. The solution is given by exploiting *Electronic Design Automation* (EDA) tools to get the challenge target running inside a machine and "live" for interaction with users. In other words, the circuit is given "life" through a notion of time and the automated control of its clock pin, which allows it to advance through its internal states. The user is essentially given 3 main faculties:

1. the ability to set the value of the inputs of the target;

2. the ability to read the value of the outputs of the target (and in some cases also of the internal states);

3. the ability to advance the simulation for a certain (simulated) time.

These faculties are made possible through the use of the specific commands of the hardware simulation tool, which therefore must be carefully filtered to avoid not only a platform crack, but also to reach the solution through trivial paths (e.g., directly reading the value of the registers containing the key, instead of exploiting the vulnerability to get it out on the output).

An EDA-tool-based environment is very suitable to host *Jeopardy* challenges, where the participant faces the hardware simulator giving inputs to the target and reading its outputs. The flag is usually obtained through a sequence of inputs over time such that the circuit is brought into a state of particular vulnerability, whereby it reveals the normally-inaccessible secret.

5.2 General Description

When designing the infrastructure, the first reasoning has been on choosing a hardware simulator that supports terminal-like commands to manage signals and advance the simulation. From the very beginning, the choice has been on *ModelSim*[1] for its ease of use and its diffusion rate within the the hardware community.

The main objective has been to create a software environment capable to accept many user connections from the outside and launch a ModelSim instance of the target for every user that connects to the service. As well, the environment has to clean temporary files and folders created by the simulation instance, and to close it in order to free memory and CPU occupation.

Sharing a single ModelSim instance between the challengers has been a discarded option, as in this case, there should be a software scheduler handling connections, which would slow down the waiting time of a single participant. One of the EDA-tool-based environment advantages is exactly the fact that the EDA tool is not a shared resource between the participants, as it is possible to create an instance of the software for every participant.

Another characteristic of the environment is that every user must try to solve the challenge in a single connection: this is made necessary since saving the hardware simulation state is hard to support. Such a state is kept into a `transcript` file (see Subsection 5.2.1) of the last simulation before starting to accept new commands. Leveraging this file would cause the solution execution time to be slowed down too much for the participant, and in addition, the CPU occupation would increase in

[1] https://www.intel.it/content/www/it/it/software/programmable/quartus-prime/model-sim.html

a non-negligible manner (as every time the new connection is established, the last transcript file has to be executed).

In the following, the current version of the EDA-tool-based environment is exposed, after a brief description of the main tool supporting it: ModelSim.

5.2.1 ModelSim

ModelSim is a multi-language environment for the simulation of hardware models described in VHDL, Verilog, SystemVerilog or SystemC. The simulation can be performed using the software GUI or using its commands through the command line modality. ModelSim uses temporary files and folders in order to distinguish different projects and simulations. In particular, a file called `transcript` collects the entire set of commands used in a hardware design model simulation, plus many warnings, errors and information strings printed automatically by the simulation results. Such a file can be seen as the history of the simulation that ModelSim is currently performing.

As any HDL simulator, ModelSim needs a *top-level component* for any simulation, to communicate the primary inputs and read the primary outputs. In order to simulate a digital circuit, a special entity called `testbench` is usually created to submit the input stimuli to the circuit. In our case, a proper testbench is not mandatory, as the primary inputs of the top-level entity are assigned using the `force` command (to force a particular value to a signal).

5.3 Environment Structure

The environment structure is based on software entities that allow to an external user (i.e., the challenge participant) to connect to the service and send commands to the ModelSim instance "executing" the challenge target. Between the user and the ModelSim instance, there is a *filter* that verifies whether such commands are allowed. If the rules are respected, these are applied to ModelSim. Set or run commands advance the simulation; read commands produce a response to be sent back to the user. Every of these steps are carried out in a secure execution environment.

The basic software elements present in the environment are:

- the *Docker*[2] *Engine*: as previously announced, in the EDA-tool-based environment a hard-to-break software system is needed. The participant has to solve the challenge acting directly on the simulated hardware model, forcing values to its inputs and reading the result elaboration after some simulated

[2] https://www.docker.com/

time. A tool to manage the security items in order to isolate the challenge environment from the rest of elements in the server is needed.

Docker is a development tool used to avoid repetitive tasks in the development life-cycle of an application, bringing with itself the security concept. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging [11]. Docker has the clear advantage to make the challenge development modular, with the possibility of exposing similar challenge services in different TCP ports of a server. Plus, it offers a good degree of security, as the container creates an isolated game environment with respect to the remainder of the system.

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, a YAML file is used to configure the application services. Then, with a single command, it is possible to create and start all the services from a configuration[3]. Compose strongly facilitates exposing services: in our reference scenario, different challenges are configured with the `.yml` file indicating the useful information as the connection port, the challenge name, and the IP addresses accepted to connect to the services.

Resuming, Docker enables developers to easily group, ship, and run any application as a lightweight, portable, self-sufficient container, which can run virtually anywhere ensuring the entire system security.

- `socat`: In order to expose and use a service, there must be a tool able to handle the bidirectional data transfer from user to the service and vice-versa. The Linux `socat` utility is a relay for bidirectional data transfers between two independent data channels. This tool is regarded as the advanced version of `netcat`. With respect to the latter, `socat` has additional functionalities, such as permitting multiple clients to listen on a port, or reusing connections [26]. The tool also allows to manage users TCP connections inserting also timeouts. It is also possible to accept multiple connections to the same service allowing many people to establish a connection without waiting in a queue their turn.

- *the software filter*: In order to analyze and process the user inputs that represent the ModelSim commands, a software filter is needed, that has been called *restricted shell*. It is implemented as a simple Pyhton script that represents the service to which the participants must connect.

[3] https://docs.docker.com/compose/

It is in fact used to launch the instance of ModelSim linked to the participant, and handles the commands sent from the user side. It must analyze the user inputs and discard the non-permitted ones, resorting to some configuration files written by the challenge organizers. Particular attention must be given to the most dangerous ModelSim commands, as the `force` command that can be used to force the value of internal signals trying to escape from some circuit logic, and find the flag in an easier way. The configuration files role is very important, and they must be written carefully. The configuration files are 3:

- `whitelist`: this file contains the list of allowed commands. These< are the only ones accepted by the system, while the other ones are rejected through notification to user;
- `blacklist`: the blacklist file contains the set of words that must not appear in the input string sent by the challenge participant. For example, the name of the VHDL `signal` storing the flag must not appear in the command string;
- `blacklist-force`: once the command is accepted, a special blacklist for the `force` command is created. Here the set of signals and variables that the participant must not be able to directly force is inserted. Typically, the entire set of internal VHDL signal names and variables is inserted in the file. In this way, the participant uses the hardware model as a *black-box* module, being able to force only its primary inputs.

With the filter option of the script, challenge organizers may also decide to change the *restriction level* of the challenge: in fact,

- they can decide which commands to insert in the `whitelist` file (if all the command names are inserted the whole set of ModelSim commands is accepted);
- they can decide which string to add in the `blacklist` file. For example, they can add the whole set of internal signals, in such a way that the hardware module is used in a complete black-box modality, and the internal signals can not be read or written;
- they can decide the set of signal names to add into the `blacklist-force` file increasing the restriction in the force phase of the signal values. For example, if no signal appears in the `blacklist` file and all the internal signals are in the `blacklist-force` file, the hardware module is used as a partial black-box component, and the user can read the values of whichever signal, but cannot force the internal values. A good practice (almost mandatory) is to write into blacklist file the component name containing the signal that assumes the flag value and the flag value itself.

The connection between the software filter and the ModelSim instance is made through the `pwntools`[4] software library. Among its functionalities, `pwntools` has been chosen for its facilities concerning the inter-process communication, thanks to which the restricted shell communicates to ModelSim the user commands and listens to the responses.

5.4 Behavior

```
WELCOME TO THE k3y_m4n4g3r SERVICE!

WRITE command_list TO SEE THE AVAILABLE COMMANDS


vsim> command_list
THE LIST OF AVAILABLE COMMAND IS:

run
examine
stop
force
show
reset
exit
```

Figure 5.1. The user perspective of the *restricted shell* in a challenge.

As already introduced, the environment supports multiple user connections to the service creating multiple independent simulations with ModelSim. In the following, the overall behavior is analyzed starting from the user input command until the simulation steps of ModelSim. Figure 5.2 resumes the overall structure with an example of 2 users connections.

The user-environment interaction starts after having exposed the service (using *Docker*). A different *Docker-container image* is created for every challenge present in the YAML file, in such a way to isolate every challenge environment. In fact, the user must connect to the correct TCP port depending on the challenge that he or she wants to face. Such a list of ports is specified by the challenge administrator in the YAML file and communicated to the challengers. The TCP connection can be carried out opening a software socket, or using tools as `netcat`.

[4] https://docs.pwntools.com/en/stable/

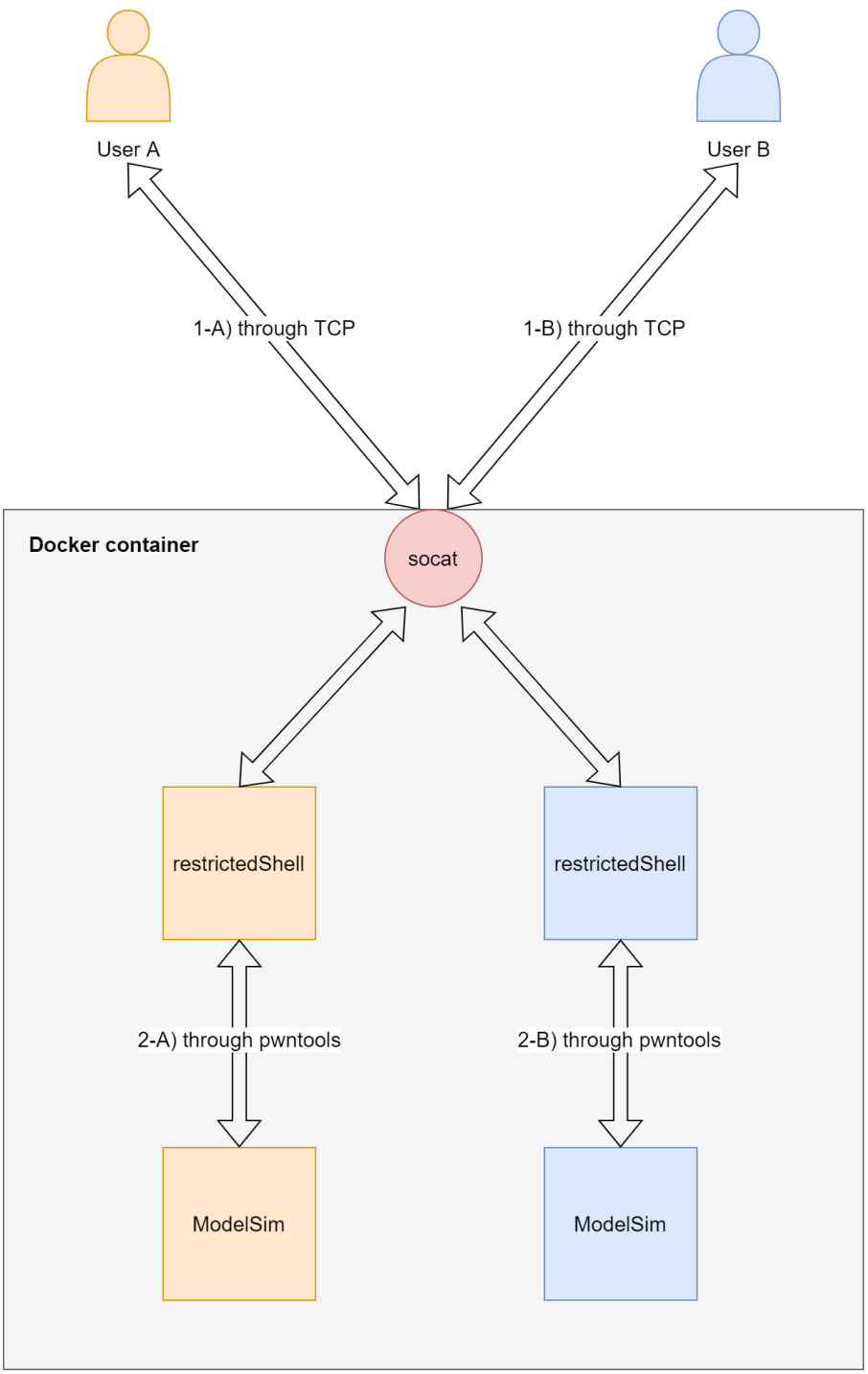

Figure 5.2. An example of connection to EDA-tool-based environment service.

For every challenge, resorting to the description inside `docker-compose` file, the Docker engine takes from `CTF-Design` folder the HDL files required for the challenge, and place them inside the container in order to initialize with the target vulnerable hardware description (Figure 5.3).

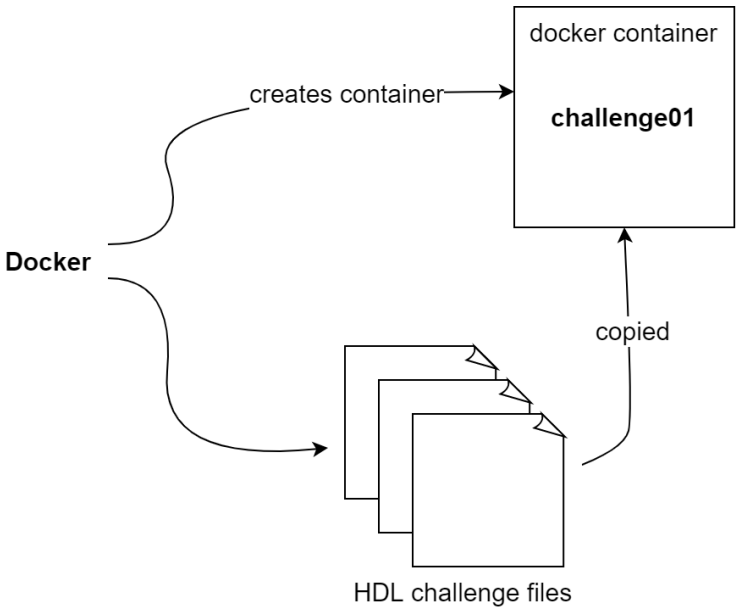

Figure 5.3. Docker automation for CTF HDL files.

The connection starts and is completely handled by `socat`, that opens a new instance of the *restricted shell* for every user that tries to connect to the service. Thanks to `socat`, it is possible to add timeouts in order to reduce the resource occupations of the server that hosts the challenges. This time interval represents the entire connection time, so the challenge resolution script written by the participant has to retrieve the flag within this interval of time. Another timeout value is often inserted: this second parameter defines the time used to exchange a single command, so the user has to submit a new one within this time period.

The service core is the *restricted shell* software that performs the mediator role between the user and the hardware simulator. When the service starts, the *restricted shell* creates a temporary folder where ModelSim has to setup its work environment, and all the CTF HDL files in the container are copied inside this folder ready to be used and simulated by ModelSim. This folder and its content is deleted once the user service session is closed. Figure 5.4 outlines this process: in this way, an isolated environment for every user connection is created inside a temporary folder.

After this setup phase, the ModelSim process is launched in the command-line

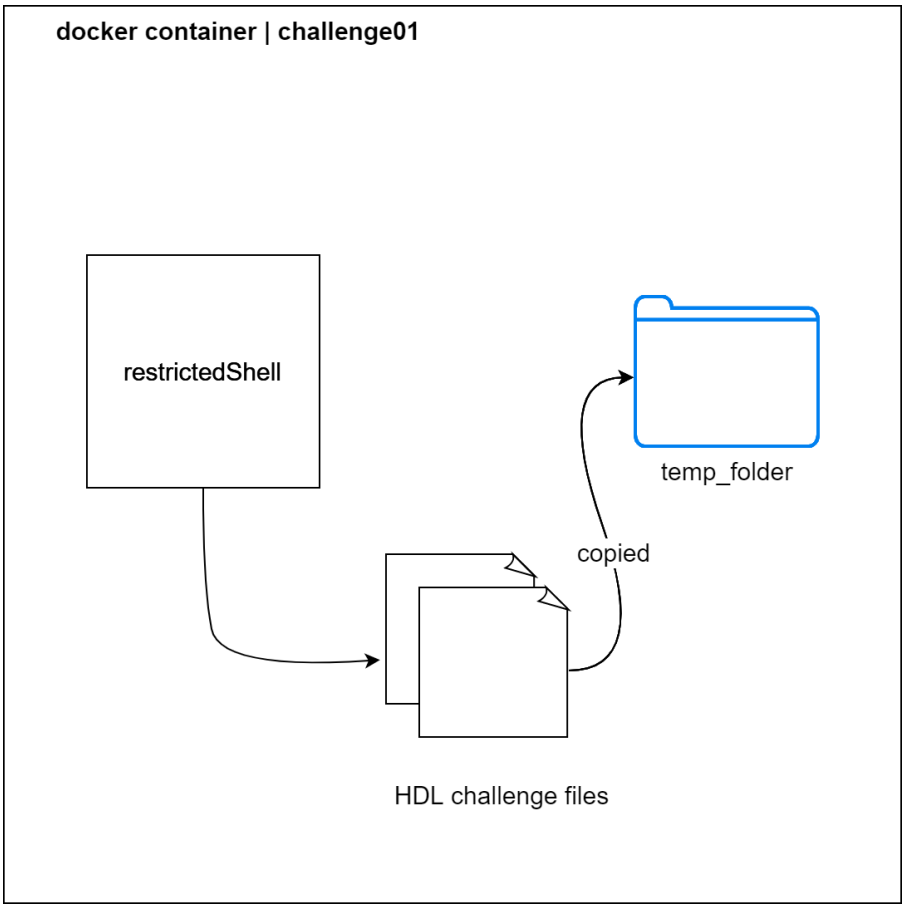

Figure 5.4. Restricted shell user simulation environment.

mode. The hardware model is compiled, the simulation starts, and the service is ready to receive the ModelSim commands from the user. Once ModelSim command is received by the *restricted shell*, filtering phase can start using the configuration files `whitelist`, `blacklist` and `blacklist-force`.

In this phase, if the command inserted by the user is included in the specified set of commands inside the `whitelist` file, the second level of filtering can start, where the check of specific words against the `blacklist` file is done. If the response is negative (i.e., there are no words in the sent command that are present in the blacklist), this second verification step is passed as well, and it means that there is no dangerous word forbidden by the environment inside the input string. Last, if the command is `force`, the presence of a forbidden name signal or an unwanted word may create an unexpected behavior. Therefore, the `blacklist-force` file is inspected in order to see if the input string contains one of the forbidden words.

The challenge designer must carefully decide which ModelSim commands are allowed to solve the challenge. A minimum set of commands is represented by (i) the `run` command, used to allow the simulation advancement, (ii) the `force` command already seen, and (iii) the `examine` command, used to read the signals value. Any other inclusion in the command list must consider the possibility of introducing information leak: the ModelSim command reference manual [18] has to be consulted deeply, looking to whether possible command arguments can create problems and how to edit the `blacklist` file to solve them. The opposite problem appears in editing the `blacklist` and `blacklist-force` files: if not properly configured, they can lead to an impossibility of using the ModelSim software properly (as a matter of example, if the `examine` command was added to the `blacklist`, every challenge would be impossible to be solved).

One very last filtering level is performed to check if dangerous characters are present inside the input string. The presence of dangerous characters can lead to not allowed actions (e.g., force a value to an internal signal blocked by the `blacklist-force` file). A well structured test phase has been needed in order to adjust the particulars that represented the vulnerability points of the software environment.

If all the filtering levels are passed, the command is forwarded to ModelSim that produces an output on the terminal. The program flowchart can be seen in Figure 5.5.

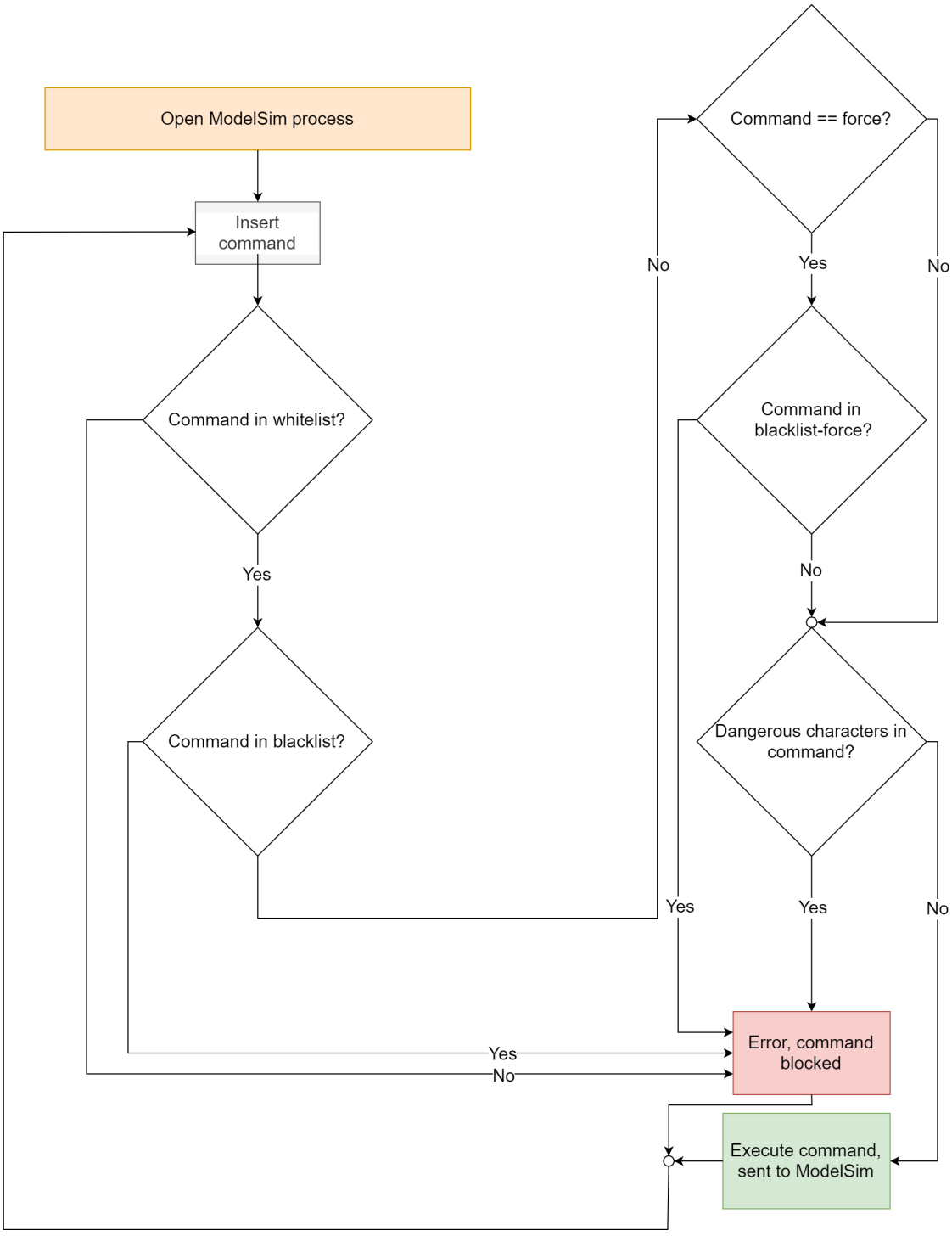

Figure 5.5. Flowchart of the *restricted shell* program.

5.5 Resources Occupation

The EDA-tool-based environment uses ModelSim as hardware simulator and Docker as framework for the development of a secure and isolated environment in order to simplify the deployment of the service. The presence of all these software can impact to the server resources in a non-negligible manner. At the moment, the environment has been prototyped only resorting to the *Intel® ModelSim - FPGA Starter Edition Software*, that in comparison with *Intel® ModelSim FPGA Edition Software* is 33% slower in terms of simulation performances[5]. Plus, ModelSim has not been created for executing hardware Capture-The-Flag challenges originally, but with the aim of performing step-by-step simulations with one instance opened and running on a personal workstation, PC, or server. The inter-communication process performed in this environment by the *restricted shell* is a specific way to perform hardware CTF. To conclude, this hardware simulator is single-threaded, which means that for every instance of ModelSim, a different physical core is used.

In the following, resource occupation measurements are reported. The environment has been tested in the *Digital Hardware Emulation Range* of the PAIDEUSIS cyber range (see Section 3.2), hosting different challenges.

Data tables have been created using the Docker command `docker stats`[6], which gives info on the following parameters:

- **CPU %**: the percentage of the host CPU and memory that the container is using;
- **MEM USAGE/LIMIT**: the total memory the container is using, and the total amount of memory allowed to be used;
- **NET I/O**: the amount of data the container has sent and received over its network interface;
- **PIDS**: the number of processes or threads the container has created.

In order to organize the collected data, some graphs have been plotted to show how the resource occupation parameters change during the resolution time. The `docker stats` command returns a live data stream for running containers. Collected data have been produced from launching a *challenge solver script*, that connects to the services hosted by the docker containers and uses the minimum number of ModelSim commands to observe the flag. In particular, the obtained graphs are related to the CPU percentage, the memory occupation that the container is using,

[5] https://www.intel.it/content/www/it/it/software/programmable/quartus-prime/model-sim.html

[6] https://docs.docker.com/engine/reference/commandline/stats/

and the NET I/O parameter that refers to the overall amount of data exchanged through the container network interface from the beginning of the solver script. This last parameter has a different kind of shape with respect to the others, as it gives a cumulative information of the network interface traffic from the beginning of the simulation, while the others give an instantaneous information.

The first results are those related to the easiest challenge developed, named `Cr4ck_the_CVM` (see Appendix C). It does not require many commands in order to be solved: the solver script is able to find the flag on average in just 3.85 s. Figure 5.6 shows how the CPU utilization percentage changes during the solution time (execution time of the Python solver script). It can be seen that the parameter value overcomes the 100% of CPU utilization: the fact that a Starter edition of the hardware simulator has been used surely has a non-negligible impact on this data. Although, on the other side, this is the prove of the fact that ModelSim is not a lightweight program: in fact, even to solve the easiest challenge, the CPU utilization goes over 100% (i.e., more than one CPU core is used).

As already said `docker stats` gives information about another important parameter: the amount of memory used by the container. Figure 5.7 shows how the memory occupation changes over time for `Cr4ck_the_CVM`. It can be seen that, contrary to the CPU% parameter, the memory occupation is not an issue. In fact, the maximum utilization does not even reach 60 MiB of occupation (namely 56.73 MiB), provided that the maximum available memory is of 125.6 GiB for the development server used to test the environment.

The third data is of the network traffic in the container interface. Figure 5.8 shows the total amount of data exchanged through the I/O interface over time.

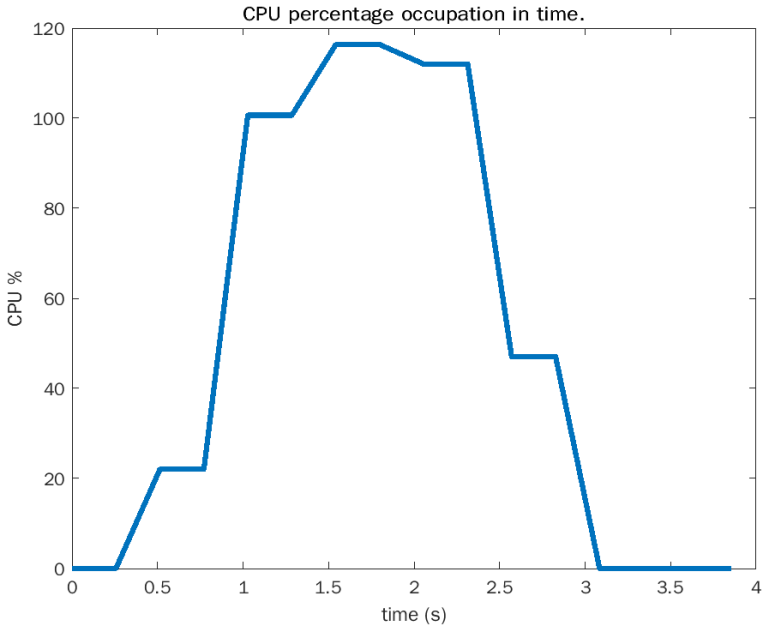

Figure 5.6. CPU percentage occupation during `Cr4ck_the_CVM` solving in the EDA-tool-based environment.

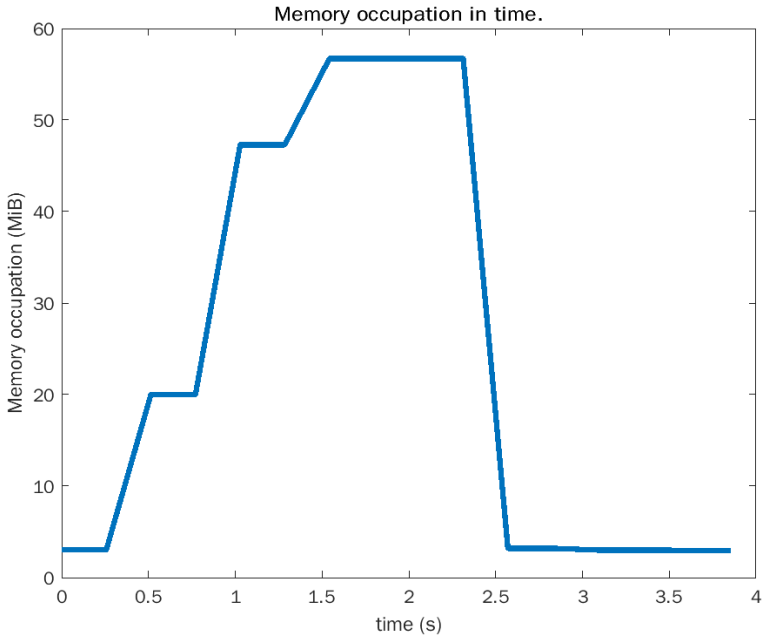

Figure 5.7. Memory occupation during `Cr4ck_the_CVM` solving in the EDA-tool-based environment.

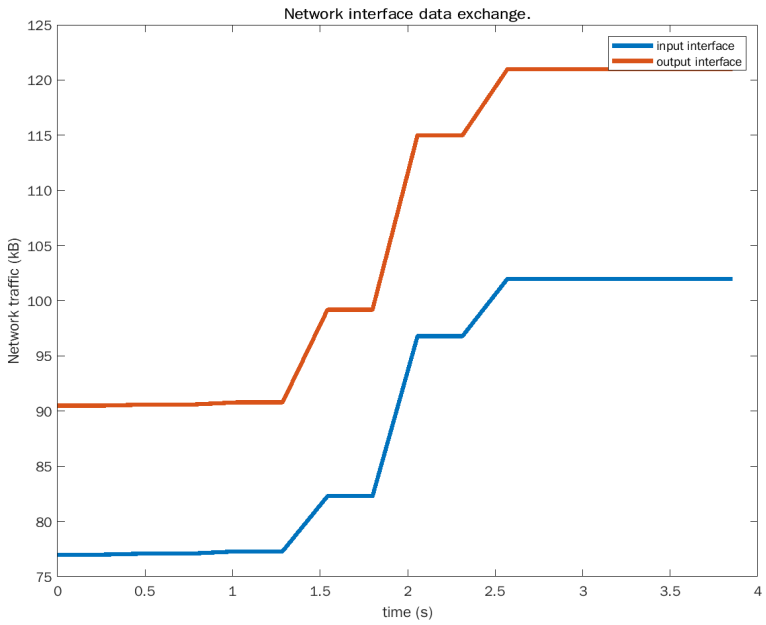

Figure 5.8. Total network data exchanged during `Cr4ck_the_CVM` solving in the EDA-tool-based environment.

Similar values can be seen for the challenge `Its_Too_Hot` (see Appendix C), that may be considered more difficult than the previous one, but not particularly heavier in terms of HDL files and of line codes number. The solver script finishes its tasks in a even smaller interval of time (namely 2.5 s). Figures 5.9, 5.10 and 5.11 show the CPU utilization, memory occupation and total network traffic. Similar considerations as for `Cr4ck_the_CVM` can be done here.

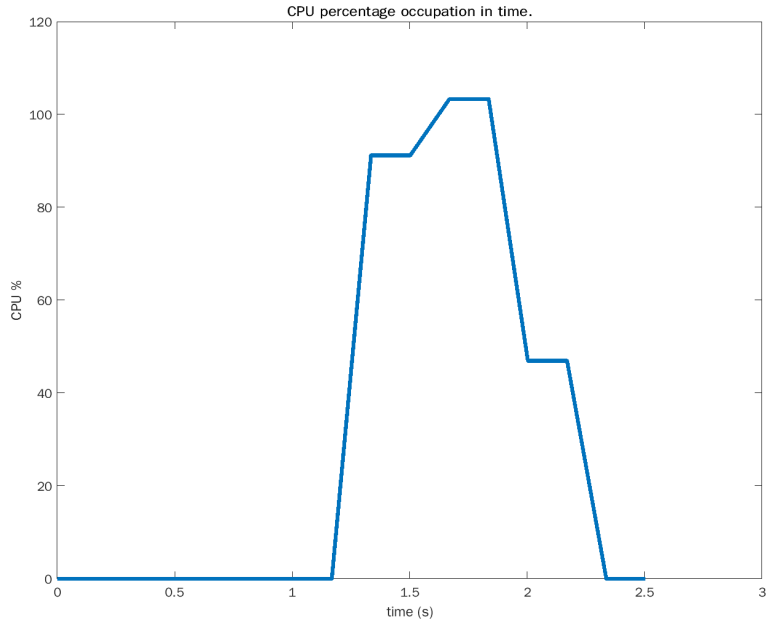

Figure 5.9. CPU percentage occupation during `Its_Too_Hot` solving in the EDA-tool-based environment.

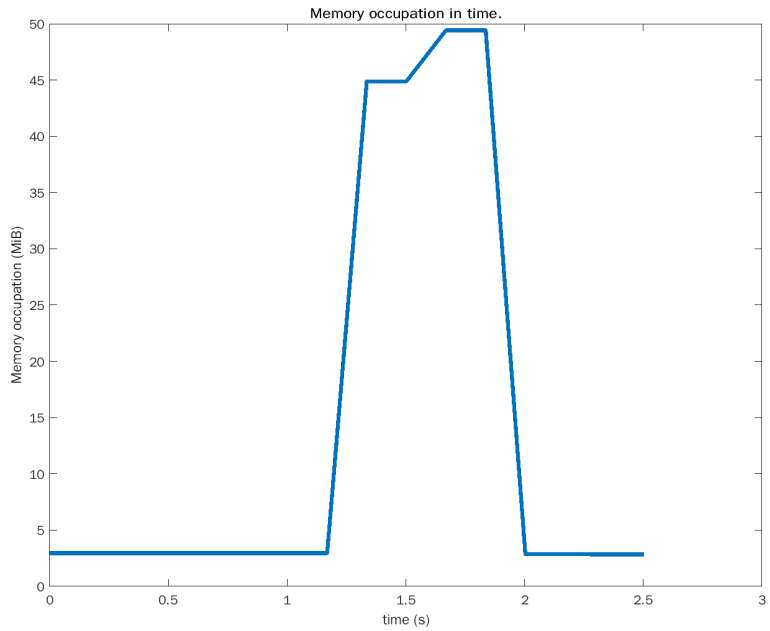

Figure 5.10. Memory occupation during `Its_Too_Hot` solving in the EDA-tool-based environment.

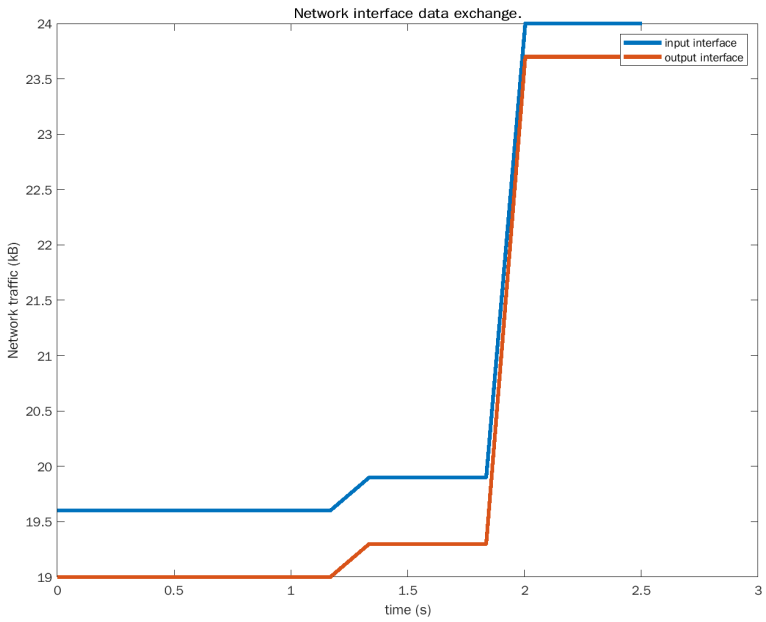

Figure 5.11. Total network data exchanged during `Its_Too_Hot` solving in the EDA-tool-based environment.

For the third experiment, the `k3y_m4n4g3r` challenge is tested. This challenge is considered a difficult one, with many HDL files to analyze. For what concerns the memory occupation, it can be seen that it reaches values that are slightly higher with respect to the other challenges because it is heavier, but still reasonable values are kept looking to the memory limit imposed by the server memory size (Figure 5.13). The critical parameter is the CPU usage as well, that overcomes the 100% (Figure 5.12). The considered time span is much longer for this challenge, as the challenge requires a higher number of steps to be solved, and thus, a longer time is interested (namely, 59.7 s).

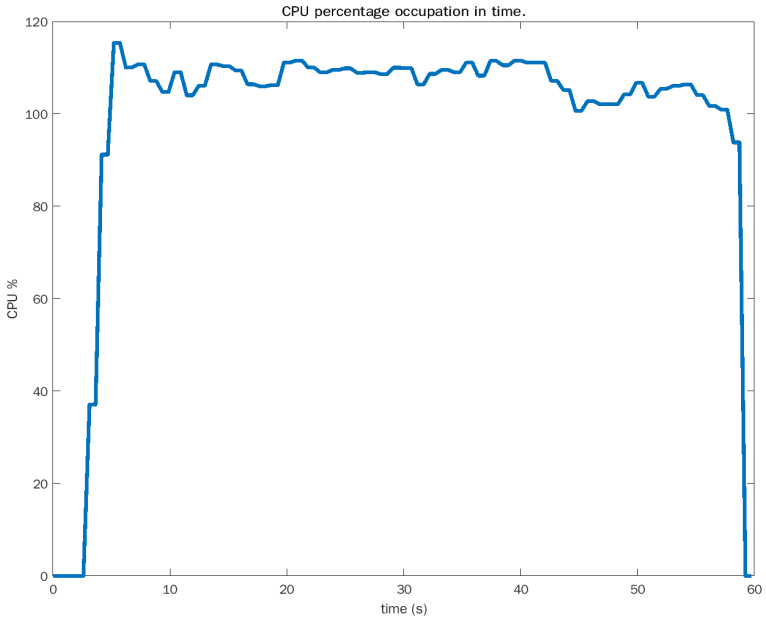

Figure 5.12. CPU percentage occupation during `k3y_m4n4g3r` solving in the EDA-tool-based environment.

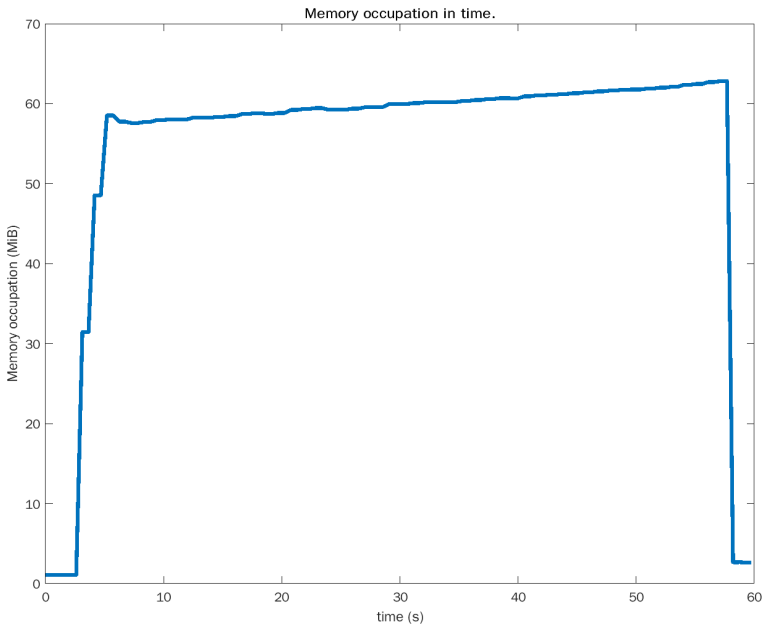

Figure 5.13. Memory occupation during `k3y_m4n4g3r` solving in the EDA-tool-based environment.

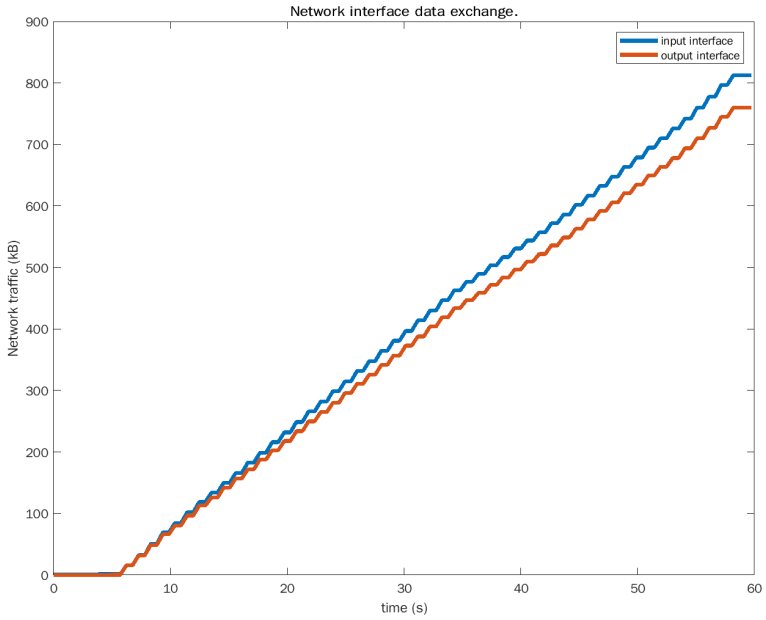

Figure 5.14. Total network data exchanged during *k3y_m4n4g3r* solving in the EDA-tool-based environment.

The fact that ModelSim is a single-threaded simulator can be noticed only looking to the results of a double connection to the service hosting the `k3y_m4n4g3r` challenge. From Figure 5.15, 5.16 and 5.17, one can notice that all the parameters are almost doubled with respect to the example with one connection, even the CPU occupation. In fact, more than 2 CPU cores are occupied with more than 2 instances of ModelSim opened and running (i.e., CPU% over 200%).

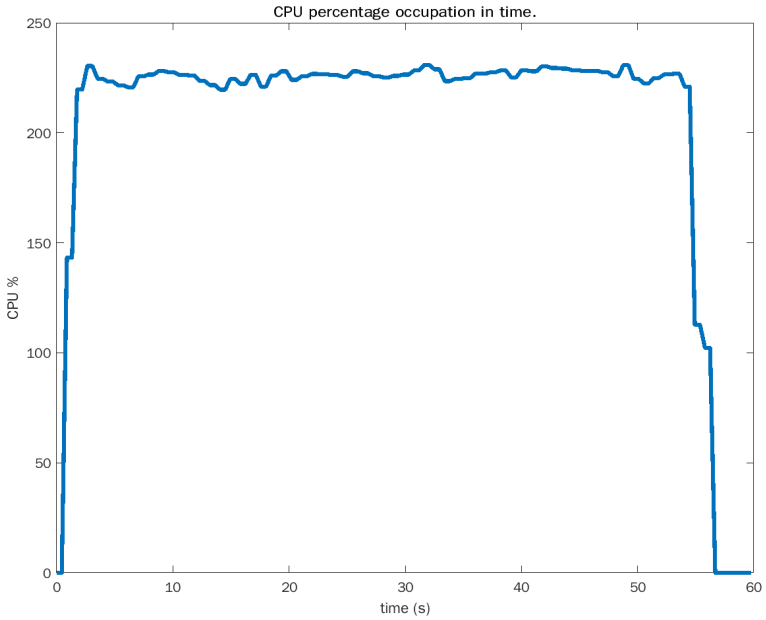

Figure 5.15. CPU occupation percentage of a double connection to `k3y_m4n4g3r`.

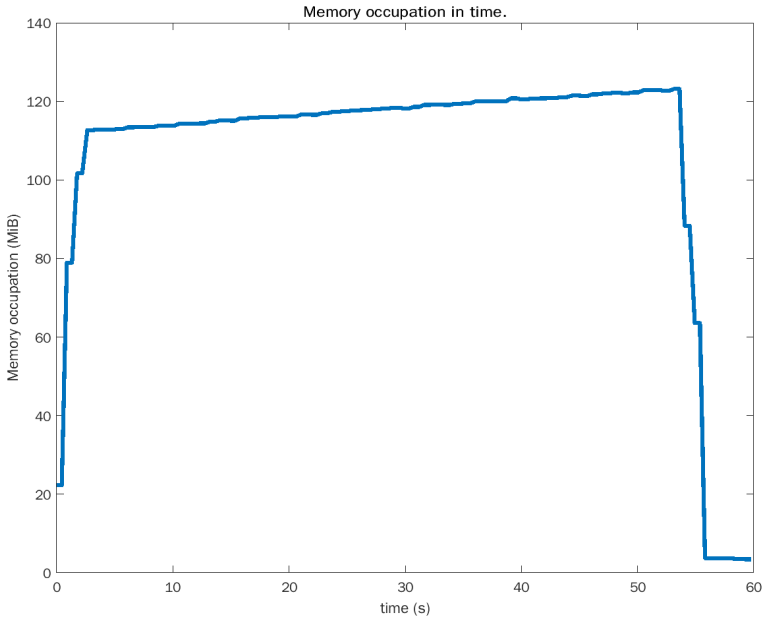

Figure 5.16. Memory occupation of a double connection to `k3y_m4n4g3r`.

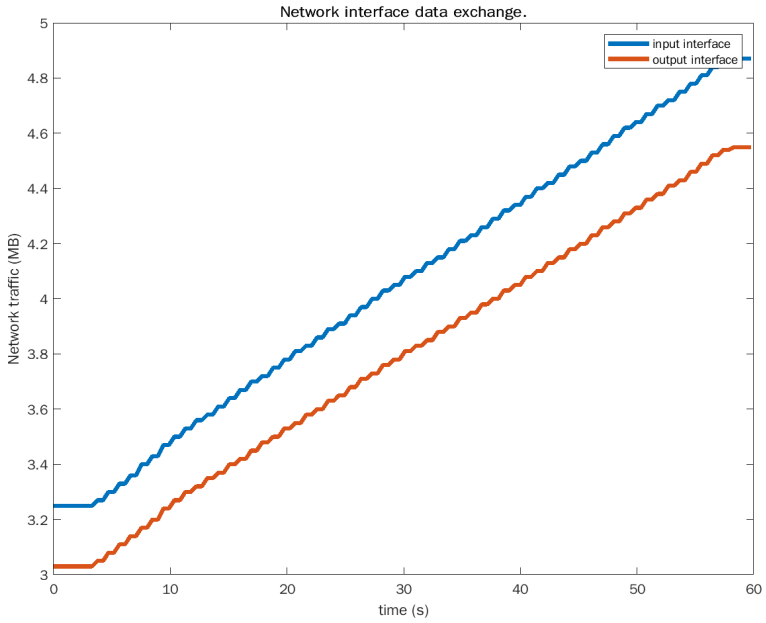

Figure 5.17. Total network data exchanged of a double connection to `k3y_m4n4g3r`.

The last example is performed with 3 connections to the 3 different challenges. This experiment is performed in order to show how Docker separates the containers creating 3 different and isolated environments. From Figure 5.18 and 5.19, it can be noticed that the memory occupation is not correlated from the 3 challenges, and, as expected, also the CPU usage is completely independent form container to container, as for every connection different CPUs are used (hosting different instances of ModelSim). The resulting graphs are just the superposition of the previous ones, because of the container property of isolation.

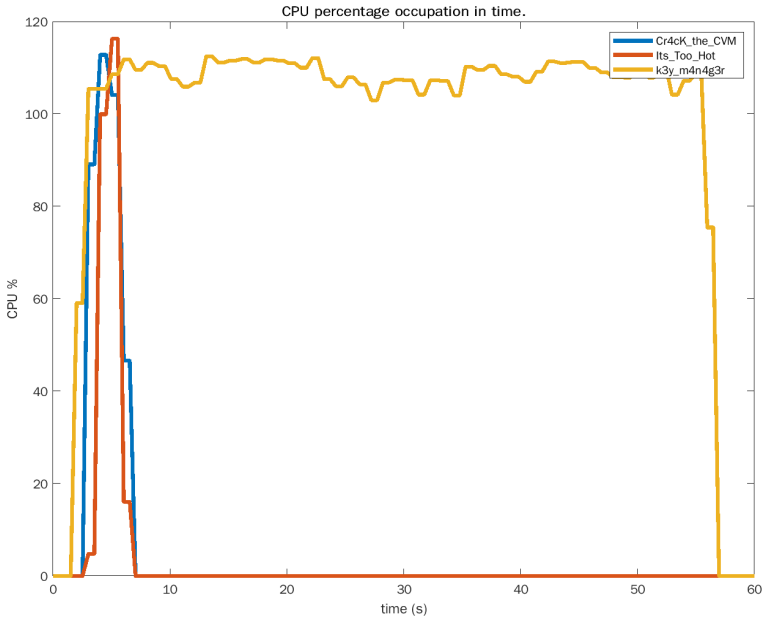

Figure 5.18. CPU percentage occupation of a simultaneous resolution of `Cr4ck_the_CVM`, `Its_Too_Hot` and `k3y_m4n4g3r`.

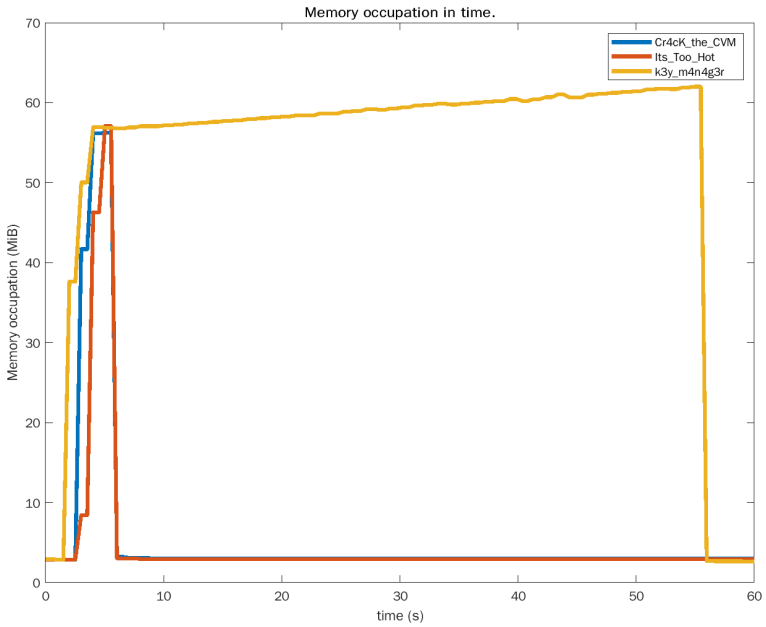

Figure 5.19. Memory occupation of a simultaneous resolution of `Cr4ck_the_CVM`, `Its_Too_Hot` and `k3y_m4n4g3r`.

In conclusion, the critical parameter for the resource occupation in the EDA-tool-based environment is the *CPU usage*, motivated by the fact that ModelSim works as a single-threaded program and occupies a core for every opened instance of it. This is also critical thinking about the fact that, in a CTF competition, many teams connect to the service in parallel, and for each connection, an instance of ModelSim is opened. This problem is caused by the fundamental reason depending on which *ModelSim is not thought for CTF competitions* and, in general, multiple connections. Anyway, a professional version of the tool can help the environment in lowring the resource occupations parameters of the server, as it is reasonably faster and optimized.

For what concerns the mass memory occupation, the evaluation that can be done is about the ModelSim software. In fact, the installation is performed by the Docker container and it only needs the `.run` file that occupies 1.38 GB of mass memory space. In theory, every Docker container is completely isolated from the others, so every challenge constitutes a different environment with different properties, but Docker is able to optimize the resource occupation using the last build cache. Thanks to this, the space needed by ModelSim is constant, namely about 1.38 GB.

5.6 Learning Outcomes

In a simulated environment like the one above presented, it is clear that the user must have naturalness in many fields that concern the software and hardware domains. The scope of the *EDA-tool-based environment* is to drive the challenge participant in retrieving the flag using and using only his hardware knowledge. In order to make this concept happen, the environment must be solid and unbreakable.

For solving the EDA-tool-based challenges, the participant must have familiarity with ModelSim, knowing its fundamental behavior and commands, and being able to expand his skills consulting the online reference guide. The knowledge of the system environment might help participants to use the provided tools, but this is not the only requirement to solve the challenges. Other requirements are constituted by basics on hardware digital design, reverse-engineering skills, and HDL understanding (VHDL, Verilog, or others depending on the used description language). Depending on the challenge, other knowledge requirements might be helpful or even necessary, such as boolean network behavior, hardware test infrastructures (i.e., scan chains, test compressors, test decompressors, JTAG, BIST, etc.), hardware-trojan detection, basics on hardware vulnerabilities, FSM design, hardware based security modules design and behavior (such as memory controllers), pipelined designs, and many others.

Sometimes, a knowledge on the usage of others support tools might be very useful. A common example is given by *Automatic Test Pattern Generation* (ATPG) tools or fault simulators, as they can help participants on in the solving process of

the challenges.

The requirements are obviously focused on the hardware domain, but other skills are always good to be possessed, such as problem solving, Python scripting (almost essential) and computer network knowledge. In order to demonstrate the claim that *an hardware CTF can be solved only having some knowledge in the hardware domain*, the two challenges `Cr4ck_the_CVM` and `Its_Too_Hot` have been submitted in the final individual CTF competition organized by *CyberChallenge.IT*, where Jeopardy challenges from many categories were submitted to about 400 people. The participants were not trained in the hardware domain, and the only expertise was most likely deriving from their personal knowledge. From the total of 400 participants, 2 people were able to solve only 1 of the 2 hardware challenges: `Cr4ck_the_CVM`, that is the easiest one, written in a behavioral-style language. This anticipates the conclusions of this work, with respect to what still needs to be done to adapt the training programs to this type of challenges.

Chapter 6

Hardware-device-based Environment

6.1 Motivations

Using physical devices to set up hardware CTFs is the most natural thought that a CTF competitions creator can have. The employ of real vulnerable hardware would allow participants to have a more direct and constructive experience from a practical point of view with respect to other types of challenges or competitions. With real hardware, it is possible to make the environment much more realistic for the players, by creating scenarios where a wide variety of tools (e.g., for directly sensing electric signals) can be used.

The present Section is intended to give details about a platform developed to make these challenges possible. The very core is based on *reconfigurable hardware devices*, that can be programmed to host generic components.

In particular, to make the developer's experience more fluid and modular, there is the need to provide a complete infrastructural environment, caring all aspects from the interface offered to the user up to the insides of the programmable hardware device. In fact, this latter needs to host in the design some *standard components* that go beyond the challenge target, accounted to offer a constant interface with the game environment.

On the other side, to communicate with vulnerable hardware, the participant needs a well-structured environment, with a communication part for which the challenger is unaware in order to keep him/her focused on the vulnerable device to be attacked.

6.2 General Description

The environment has been developed following a *bottom-up* approach. Using the SE*cube*™ Development Kit board, having both a FPGA and a microcontroller connected with a parallel interface, an entire environment has been developed studying and modeling the firmware, the software and the hardware components placed inside FPGA already developed for this board.

The FPGA internal configuration architecture starts from the current software system available for the SE*cube*™ platform[1], and especially from the idea of a component able to host generic multiple IP cores inside the FPGA (i.e., `IP Manager`[2]), editing it to host just one component (i.e., the CTF target) and adapting the whole firmware and software infrastructure to wrap the CTF target.

6.2.1 SE*cube*™ Development Kit

The SE*cube*™ chip is a 3D multi-module SoC (System-on-Chip), integrated in a 9mm x 9mm BGA package. The single chip embeds three hardware components: a powerful processor, a flexible FPGA, and an EAL5+ cerfified smart card (Figure 6.1)).

The main hardware component of the SE*cube*™ used in this environment are the CPU and the FPGA. The processor used inside the SE*cube*™ is the STM32F429 by STMicroelectronics™[3] with a single ARM Cortex M4 RISC processor.

The FPGA component is a Lattice MAchXO2-7000 device[4]. The configuration within SE*cube*™ threats the FPGA as an external memory giving to the processor the control of FPGA pins.

6.3 Environment Structure

The environment is developed starting from the original SE*cube*™ SDK. The SE*cube*™ Open Source Software Architecture is structured in several *Abstraction Layers* and Application. At each Abstraction Layer, several sets of APIs are provided. These layers are used and customized for CTF competitions. The firmware is modified to communicate with the two game modalities (Jeopardy and Attack/Defense), that requires two different Host-side applications. A basic idea for the communication between player and the CTF target is represented in Figure 6.2.

---

[1] https://github.com/SEcube-Project/SEcube-SDK

[2] https://github.com/SEcube-Project/IP-core-Manager-for-FPGA-based-design

[3] https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html?querycriteria=productId=SS1577

[4] http://www.latticesemi.com/view_document?document_id=38834

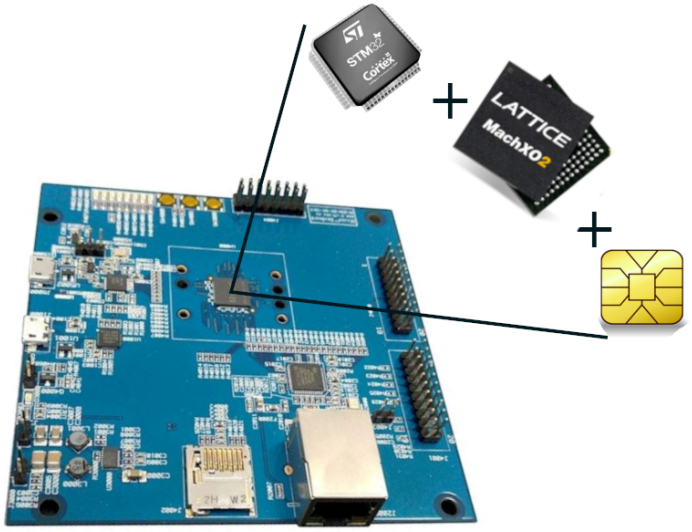

Figure 6.1. The SE*cube*™ Development Kit and its components.

A component called `IP-Core Manager` is used as a starting point for the *FMC for CTF* project, that contains the main components for the FPGA and allows to put in communication microcontroller and the component inside FPGA

6.3.1 Basic FPGA Component Description

In this Section, it is explained how the FPGA communicates with the STM32F429 microcontroller and how the `IP-Core Manager` is remodeled for the *FMC For CTF* project.

To communicate with the design flashed inside the FPGA, a *mediator* entity is needed: since the programmed hardware is a generic component, it is not possible to know in advance which and how many inputs and outputs it has, while a standard interface is needed. This mediator has the aim to read the data that microcontroller wants to send to a vulnerable design inside the FPGA, and forward its responses to the microcontroller.

Therefore, the SE*cube*™ microcontroller has the only task of viating data between the Host-side application and the FPGA (and vice versa). In order to accomplish this,it has to execute a specific firmware that only performs this functionm and it is able to interface with the broker component mentioned above.

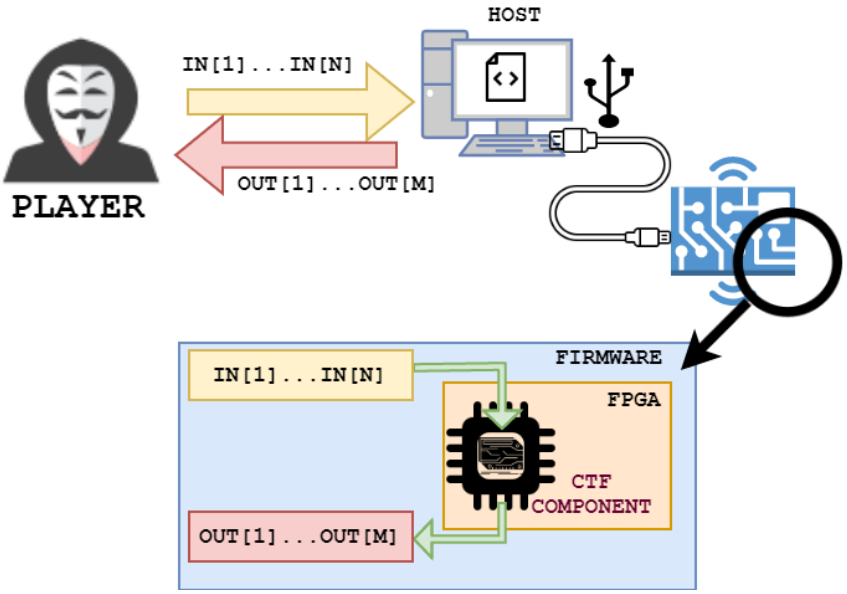

Figure 6.2. Communication between player and CTF target component.

The Starting Point: IP-Core Manager

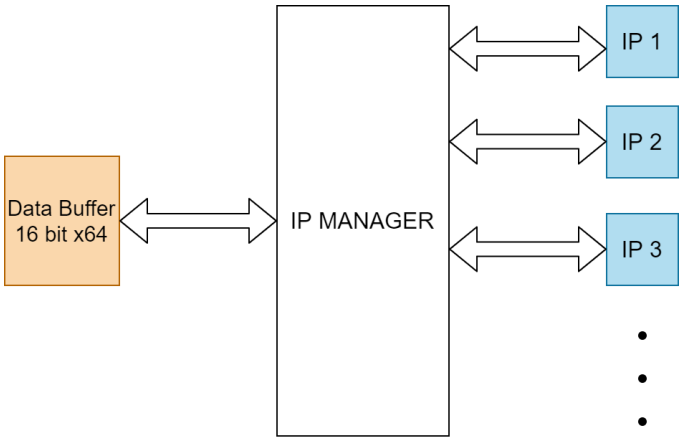

Figure 6.3. The `IP-Core Manager` component.

The broker structure is composed basically from two components and a certain number of IP cores (Figure: 6.3):

- The `Data Buffer` represents the interfacing memory module with the CPU, where commands and inputs for the FPGA are stored and where outputs are placed at the end of the operations. The block has 2 interfaces: one communicates through FMC with the CPU, and the other one is used for communications with the internal cores through the `IP-Core Manager`;
- The `IP cores` are the target modules of the CPU-FPGA communications;
- The `IP Manager` itself, with the communication task as described above.

The FMC for CTF Project

The basic idea for this project is to allow CTF designers, following limited instructions, to create generic hardware components that the environment is capable to host inside it. *FMC for CTF* is a project created starting from the `IP-Core Manager` idea, for which the full setup for the IP cores is useless.

The FMC (*Flexible Memory Controller*) is a component inside the CPU of the SE*cube*™ that takes the role as mediator between CPU and FPGA. The `IO-Connector` component described below is developed following the rules that allows the CTF component to communicate with the microcontroller.

The `IP-Core Manager` has been remodeled with the aim to host only one IP core, so that the `IP Manager` component can be deleted, directly bypassing the communication from `Data Buffer` to the vulnerable component designed specifically for the challenge, from now referred to as `CTF-Component` (Figure: 6.4).

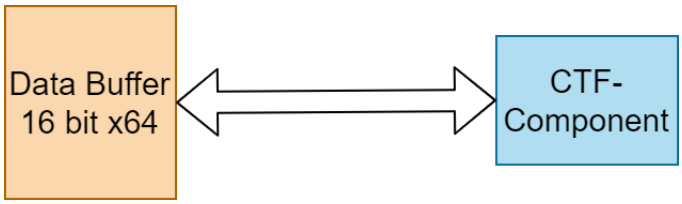

Figure 6.4. The `IP-Core Manager` remodeled for one IP core (`CTF-Component`).

In this way, the `Data Buffer` takes the role of wrapping the `CTF-Component`, and it is transformed in the `IO-Connector` for FPGA component. The 3 main components are described in Figure 6.5:

- The `IO-Connector`, that is a communication module between `CTF-Component` and microcontroller through the `TOP-FPGA` wrapper. The `IO-Connector` is the `Data Buffer` component remodeled and resized with the role to have only registers that manage the `CTF-Component` inputs and outputs. To keep a generic structure, an `IO-Connector` for `CTF-Components` with 10 inputs and 5

outputs (with maximum 16 bit each input or output) is designed. This means that the vulnerable hardware design is forced to have maximum 10 inputs and 5 outputs of maximum 16 bits.

The addressing for `IO-Connector` is like a simple register file: the registers from 1 to 10 are dedicated for the inputs (without including clock and reset signals), while registers from 11 to 15 are for outputs. The register 0 is a special register called `Data-Stable`: when it is set with a decimal 1, asserts a simultaneous assignment of the signals from `IO-Connector` to the `CTF-Component`, from which output are stored inside the `IO-Connector` output registers.

- The `CTF-Component`, which is the vulnerable hardware design to which the participant has to face. The communication with the SE*cube*™ CPU is made possible thank to the `IO-Connector`. The entire environment is also created in order to simplify the design phase of the challenge creator, that only needs to create the `CTF-Component` VHDL code, without caring about the communication part.

  To create a `CTF-Component`, the designers should take into account:

  - the maximum number of input/output for the component;
  - the creation of a component called `TOP_CTF` that is the wrapper of the real `CTF-Component`. This component should respect the following declarations in the VHDL code, with this order:
    - `clk` (clock) signal;
    - `rst` (reset) signal;
    - a maximum of 10 input signals with the limit of 16 bit in size;
    - a maximum of 5 output signals with the limit of 16 bit in size.
  - the creation of a configuration file called `IO_TOP.conf` that reports names and sizes of the input/outputs of the CTF target.

- The `TOP-FPGA` component, that is a simple wrapper used to make compatible the `IO-Connector` with FMC module.

Synthesis Automation

An automation script has been created with the aim of speeding up everything and allowing hardware-designers to focus only on the hardware components and launch only the script created for the synthesis (Figure: 6.6).

The script follows a particular roadmap with the following steps:

1. The `CTF-Component` and the file `IO_TOP.conf` are analyzed;

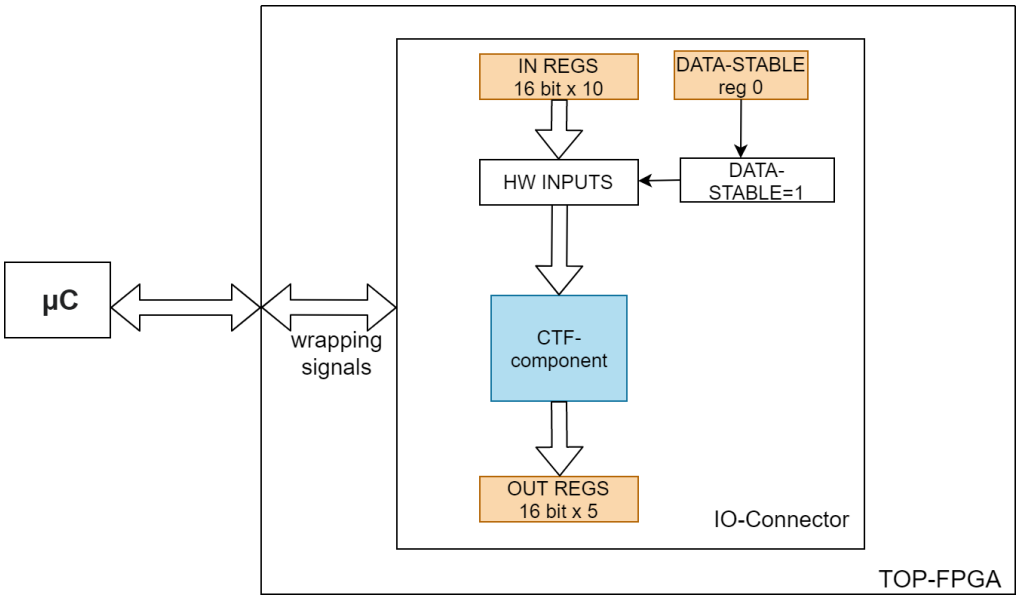

Figure 6.5. The FMC for CTF Project with all components.

2. If these files are correct, the file `IO_CONNECTOR.VHDL` with inside the instantiation of the `CTF-Component` with signal connection is created;
3. The synthesis process is started;
4. If there are no timing or synthesis errors, the corresponding bitstream is generated and it is placed in the firmware files that should be flashed inside the FPGA.

With this script, the CTF organizer has only to think about the creation of the vulnerable `CTF-Component` and the configuration file described, as the `IO-Connector` and the other modules are automatically created by the synthesis script. For this script, the *Lattice Diamond* software tool[5] is required.

[5] https://www.latticesemi.com/latticediamond

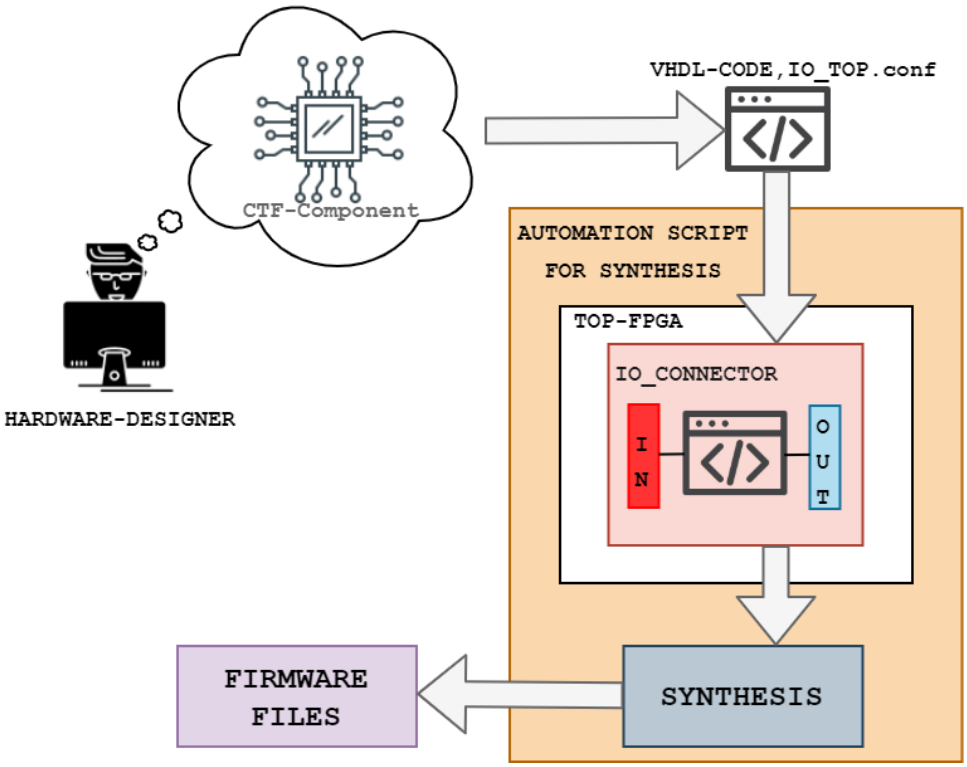

Figure 6.6. Synthesis automation.

6.4 Behavior

6.4.1 Software Description

Jeopardy Setup

The basic idea for the Jeopardy setup is to have a complete black-box component, making the player focused only on hardware with the aim to find the flag exclusively within the FPGA. This Host-side application made for Jeopardy competitions (called `Host_CTF`) is only required to manage and parse the input that will be sent to the board. For this kind of challenges, registers that contains the flag are added inside the FPGA, and it is difficult (if not impossible) to retrieve the flag by reversing the FPGA bitstream.

The `CTF-Component` has a hardwired flag inside, and must propagate it through the outputs signals. The *main* function inside `Host_CTF` should be modified by CTF-designers with the aim of parsing input and sending them correctly to the board, or printing custom outputs in the participant terminal.

The board always accepts 10 integer inputs, plus the reset that is managed separately. In response, the board gives back the 5 integer outputs and, only for Jeopardy, the output from flag registers like additional outputs, as the flag is embedded inside the FPGA. Obviously, the number of inputs or outputs could be increased at the expense of losing space inside the FPGA due to the I/O registers.

The modes to play in this Jeopardy device-based challenge are mainly two:

1. giving a board with the `Host_CTF` already compiled to each team, letting them to connect the SE*cube*™ board, execute the application and start the challenge;
2. connecting the board to a server and starting the `Host_CTF` service within a Docker container using `socat` (see 5.3).

For the Jeopardy modality, it is possible to assign the LEDs control to an output by adding inside `IO_TOP.conf` a third column and writing `ON` for only one output that should be attached to LEDs.

Attack/Defense Setup

The real innovation is actually the transition from Jeopardy hardware challenges to Attack/Defense ones. This modality has probably never been explored for hardware challenges, and comes from the idea of putting online a service managed by a hardware component where the flag is found only by violating the hardware and not the system that manages it. A big difference between the Jeopardy mode, described above, is that the flag is stored *outside* the board, inside a file placed in the virtual machine assigned to each team. Each team have different flags that are updated by the arbiter using a `group_id` password unique for each team. If the opponents exploit the vulnerabilities, the board output can trigger the output of the flag stored inside the file.

Attacking an online hardware service is like violating a web service, with the difference that players should know the hardware component that is providing the service, like the remote Jeopardy hardware challenge described above. In Attack-/Defense challenges, the teams should guarantee the continuous availability of their services, trying to patch the hardware vulnerabilities and at the same time violate the other team services. In hardware terms, patching means writing again the VHDL code of the vulnerable components, synthesize the new design, and flash it inside the FPGA.

The game is usually divided into rounds called *ticks*, at the end of which the flag is changed and a runtime control is started to check if the systems of teams are working correctly.

In such a kind of games, there are three main roles:

- The **defender**, encharged to patch the hardware by analyzing and modifying the VHDL files of the component provided to each team;

- The **attacker**, encharged to exploit the vulnerabilities discovered to attack the other teams and retrieving the flags;
- The **arbiter**, encharged to change the flag every tick and to check that every team is guaranteeing the correct availability of the service.

If the arbiter finds inconsistencies between the flag and the service offered, penalty points will be added to the team that is not guaranteeing the correct service. In fact, each team knows its own flag, and has the entire control of its system and it is responsible for the functioning of the service itself.

From the hardware CTF creator point of view, a dashboard called `CTF_MANAGER` is developed with the aim to automate the environment setup or execute only simple steps shown in Figure 6.7.

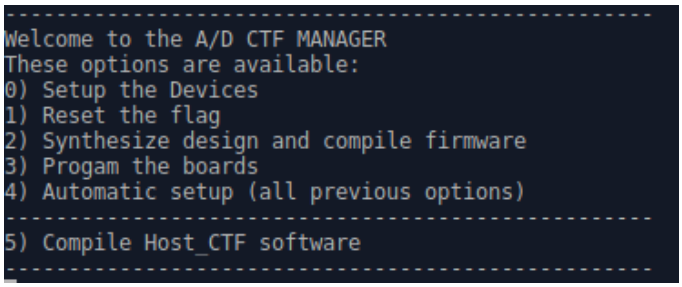

Figure 6.7. The Dashboard for challenges organizers

Developing an entire automation workflow allowed the creation of a player environment with a dashboard (Figure: 6.8) that automates the synthesis, programming, starts and stops of the service. Commands offered by such a dashboard are:

- The **Status of the service** command, telling the user if the service is online and which board is busy;
- The **Synthesize and compile firmware** command, allowing the user to synthesize the modified component in VHDL and prepare the firmware file to be patched;
- The **Program a board** command, that flashes the firmware inside the selected board;
- The **Start a service using a board** command, making the service available online;
- The **Stop the service** command, that shuts down the service.

Figure 6.8. The dashboard for teams that participate to Attack/Defense challenges.

This bench of services has to deal with the inevitable fact that programming the FPGA takes a few minutes, and the service would spend too much time offline. In Attack/Defense challenges, a fundamental rule is to take the service online as much as possible, and penalties are applied to teams with respect to their SLA (*Service Level Agreement*) points.

For this reason, it has been decided to *buffer the system*, which consists into the usage of two boards: one to be reached by the online service, and the other one to be patched and reprogrammed. Once a board has been patched, it is ready to be put online, replacing the other one. In this way, the boards can be switched into the service in a very short time, without incurring SLA penalty points.

From the application point of view, the difference between Jeopardy and Attack/Defense is the possibility, for the arbiter, to change the flag as a special user with a specific password for each group. The service is put online using `socat` as for the Jeopardy remote challenges.

To manage the board duplication, Docker engine is used with the role to isolate a single board when it has to be put online. Docker is also useful to isolate the machine where is plugged the vulnerable virtual host from external attacks. The environment scheme is represented in Figure 6.10, and the concept for boards setup that should be given to each team is depicted in Figure 6.9.

Two dockers containers are created for this environment:

- The `service_fpga_env` container, that, with *Host_CTF*, allows to start an isolated service with a selected board;
- The `check_fpga_env` container (usually used after the programming phase), which thank to a register inside the FPGA (checked by a custom host), it is capable to report to a user if the board FPGA is programmed and ready to be used.

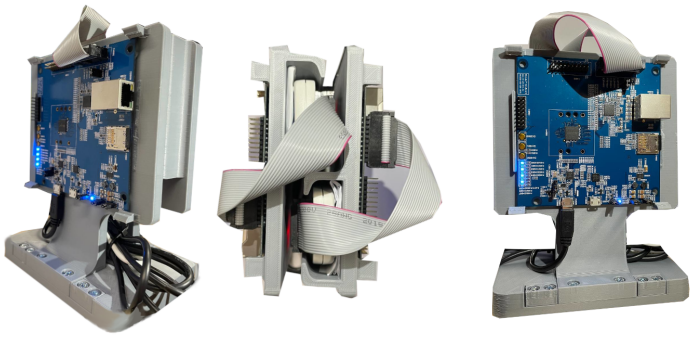

Figure 6.9. The Boards Setup for Attack/Defense Challenges.

6.5 Resources Occupation

The benchmarking experiments run on the environment demonstrated a modest impact on memory and CPU resources. From the point of view of mass memory, the weight of Lattice Diamond and the tool used for programming SE*cube*™ must be taken into consideration (i.e., around 8 GB). These tools are used only to make the synthesis and program the board in the Attack/Defense challenges.

To keep the service online, Docker is used with a container that uses the Alpine image (light distro), and inside has `Host_CTF`, which is an executable of a few MB. In the Table 6.1, it is noticeable that, for a single team, few MB are requested to host the whole system. This can be considered as a remarkable strong point of this environment.

Table 6.1. Table for mass memory occupation by Docker images for Attack/Defense environment.

REPOSITORY %	SIZE
service_fpga_env	224MB
check_fpga_env	224MB

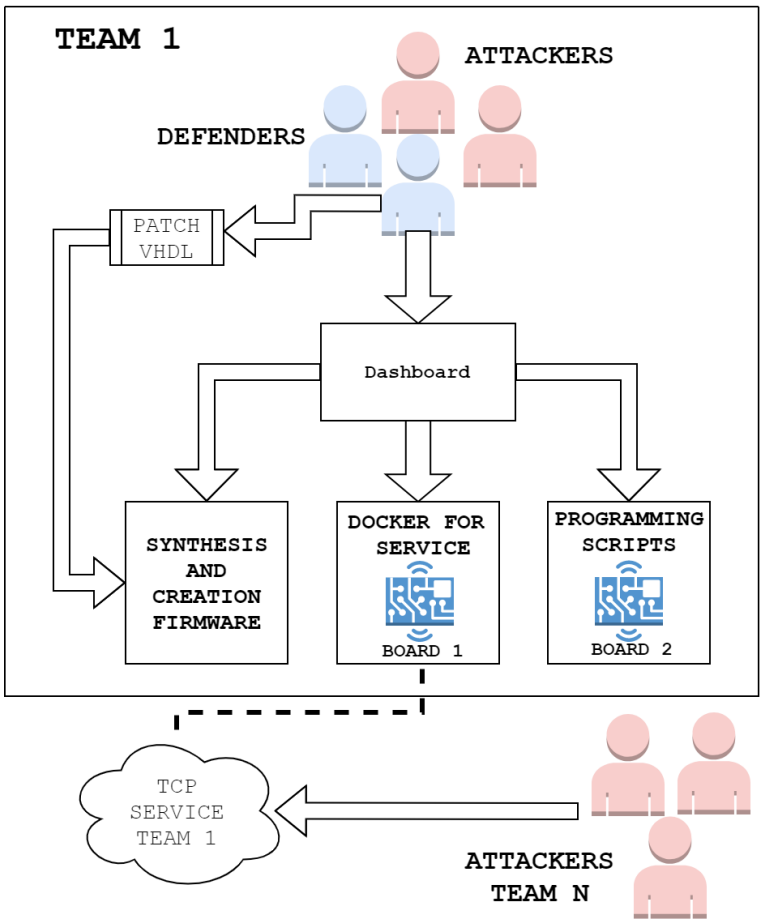

Figure 6.10. Structure of the Attack/Defense Environment.

In the graph in Figure 6.11, it is explained how `check_fpga_env` container requires resources from CPU. The utilization trend has this shape because the software that checks if the board is connected is continuously launched in loop. It should be taken into consideration the fact that this container runs every time a team wants to change the firmware inside a board. The resources are therefore occupied for a small period during the overall game duration.

Figure 6.12 shows the `check_fpga_env` memory occupation over time. It reaches a maximum of 3.2 MiB and takes up very few resources. The information about network statistics for the `check_fpga_env` container are useless, as it does not have external connections.

As for service container, it must be taken into account that it is running continuously. The memory and the CPU resources requested by the container are considerably low even under attack. Figures below represent a single connection

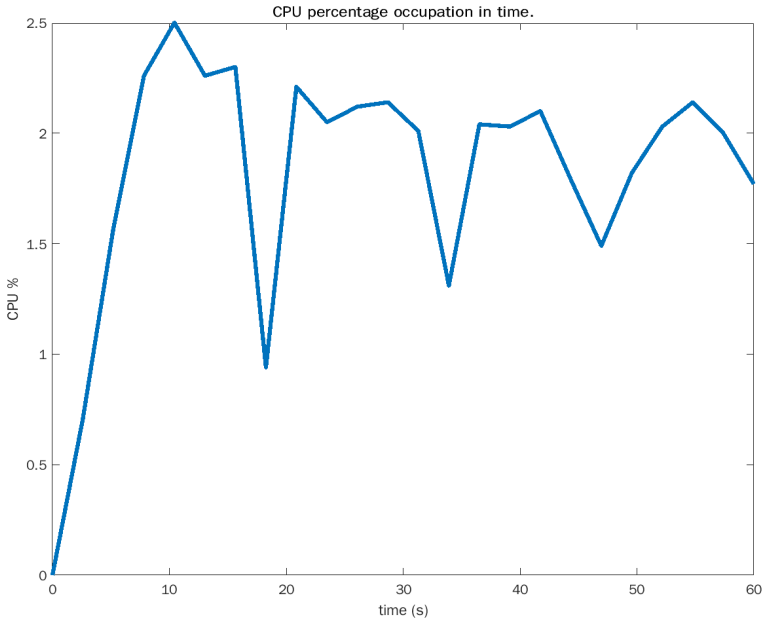

Figure 6.11. The CPU percentage occupation by the `check_fpga_env` container.

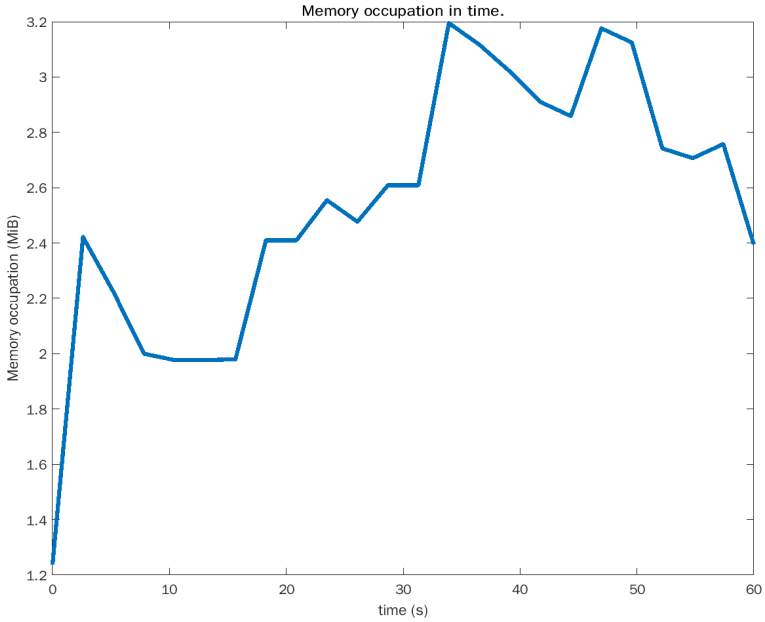

Figure 6.12. The memory occupation by the `check_fpga_env` container.

and, after 30 second, 1000 attacks executing the solving script simultaneously.

As it can be seen in Figure 6.13, it is clear that under 1000 local attacks, the CPU requires at most 10% of usage and the memory utilization (Figure 6.14) stays under 4MiB of usage, which is a very good result.

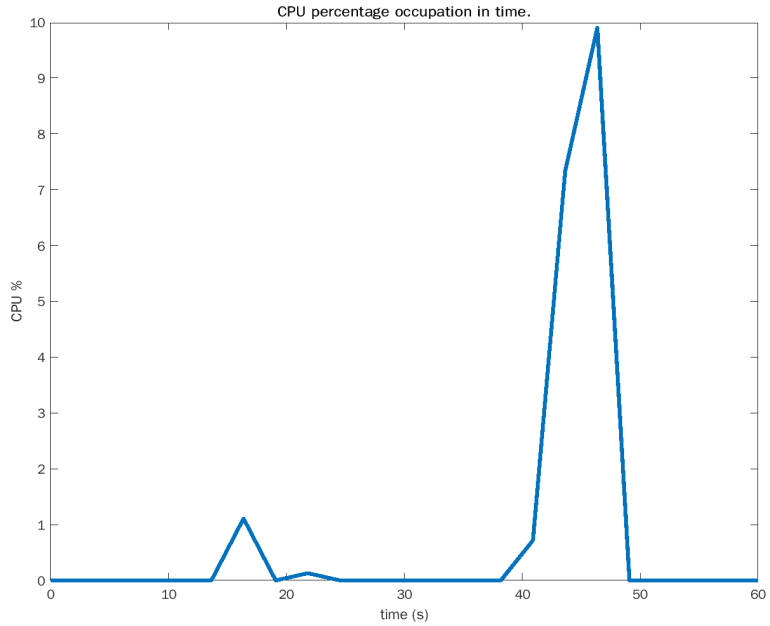

Figure 6.13. The CPU percentage occupation by the `service_fpga_env` container.

Obviously, these attacks are executed simultaneously, but they are scheduled server-side by *socat*: in fact, in device-based CTF context, it is possible to make one attack at a time due to the fact that the hardware device is a shared resource between the participants. Thus, there is only one executable per time in running phase.

Figure 6.15 shows the cumulative amount of traffic exchanged by the container. The same trend can be observed for the input and the output channel under attack: this derives from the fact that, for every input sent to FPGA, a corresponding output is to be sent to the player. The amount of data input is bigger than the output because the input channel as defined for the environment `CTF-Components` is bigger than the output one.

6.6 Learning Outcomes

The presented environment for device-based challenges still has to be tested in some major competitions to observe the effective results from the players. Considering

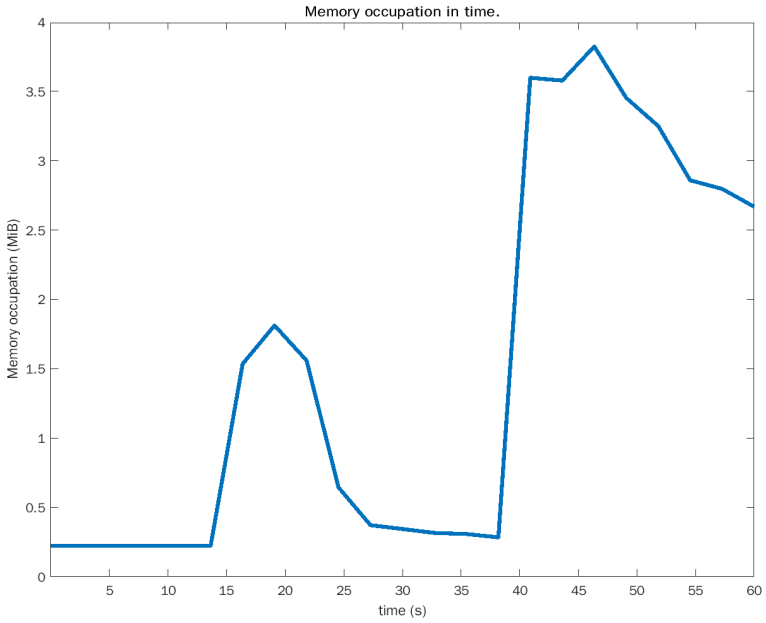

Figure 6.14. The memory occupation by the `service_fpga_env` container.

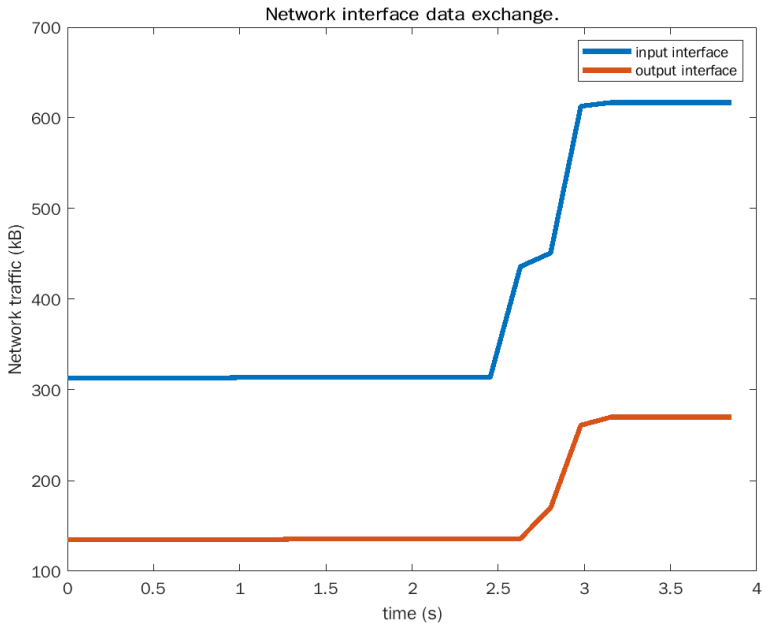

Figure 6.15. The total network data exchange from the `service_fpga_env` container.

that the challenges based on hardware Attack/Defense cover the same topics of the Jeopardy type and offer other notions to learn, they are an essential part of the hardware-based CTF, as they explore both the attack and the defense fields in the hardware security domain.

What is expected, in particular from the Jeopardy-type hardware challenges, is understanding and reverse-engineering of hardware components designed in HDL language and hardware digital design concepts. In this regard, the learning outcomes are similar to the EDA-tool-based challenges (without the knowledge on EDA-tools commands). In addition, for Attack/Defense challenges, it is also expected that the participants learn how synthesis tools are used for hardware components, and the criticalities behind a synthesis (e.g., the respect for timing constraints).

In conclusion, Jeopardy challenges with physical design are essential in order to have a direct experience with the vulnerable hardware. Physical items can be used in order to interact, like LEDs or buttons, but the Attack/Defence typology also requires different knowledge, from the synthesys tool experience to the VHDL design of synthesizable hardware components, or to FPGA programming.

Chapter 7

Conclusions

The work done in this thesis is focused on the development of two environments for hosting CTF competitions treating hardware security issues: one *EDA-tool-based* and the other *Device-based*. These two environments represent scenarios with the highest development potential from the typologies of hardware CTF competitions and with the highest number of learning outcomes reachable thank to the hostable challenges. Moreover, such environments are difficult to be found in the current panorama of CTF events.

Independently of the fact that hardware challenges know a limited employ, the platform can be considered something innovative, as these environments allow a CTF event maintainer to host hardware targets in a non-specific way, and then develop generic and reshaped scenarios, which makes these environments long-lasting. The environment genericity attribute is the real point of strength: given the environment, it is very easy to create a new challenge with a vulnerable hardware design of whichever behavior for both EDA-tool-based and Device-based environment.

7.1 Environment Differences

There are several main differences between the two environments described. First, the Device-based environment takes advantage of the FPGA circuit. This makes the hardware based environment non-paralellizable, as the board can be used once per session. The physical hardware circuit that hosts the vulnerable design becomes a shared resource because that can be attacked to one participant/team at a time.

One solution would be increasing the number of boards, but this would require a non-negligible increase in costs if the used board is expensive and the participants are many. A scheduling algorithm (as FIFO) is required, and the same board has to provide the same service to many people/teams in a specified and variable interval of time (decided by the organizers of the CTF). On the contrary, for the EDA-tool-based environment, every connection performed by the participants is assigned to

a different instance of the hardware simulator. In this way, the same vulnerable hardware design can be attacked by different teams at the same time. Clearly, for the EDA-tool-based environment, the parallel simulations impact in an evident manner the performances of the system that hosts the hardware CTF competition (as highlighted in Chapter 5).

From the point of view of system scalability, it is convenient to use EDA-tool-based environment for a larger user base, as the software simulator is much more scalable.

7.2 Game Experience

The Device-based environment makes the scenario much more realistic, as we are dealing with real boards. Therefore, the usage of instruments and peripherals such as oscilloscopes, keyboards, LEDs and screens can be used to solve the challenges, unlike EDA-tool-based environment. The usage of physical devices is clearly more fascinating, but it is a completely different experience from the EDA-tool-based for many points of view.

Device-based challenges can be seen as black boxes: the participants set the inputs to be given to the vulnerable circuit, and after some time, they read the output values. They have no control of the internal part of the vulnerable component, even for the examination of the signals. For EDA-tool-based challenges, instead, inspecting internal signals is allowed. This gives a different gaming experience, in which you can test a hardware component step by step, deciding which value has to assume a given signal in a given instant of time, something that the Device-based platform does not support for obvious reasons. As for the material delivered to participants for challenges hosted in both environments, VHDL source files (with the masked flag o even without the presence of the flag for Attack/Defense challenges) represent the starting point, and the challengers have to carefully study them to retrieve the flag.

7.3 Resources

The EDA-tool-based environment implemented for this thesis uses ModelSim as simulation engine for the hardware, and this unfortunately occupies a lot of resources from the point of view of CPU, as already discussed. The fact that every user can perform multiple connections, and for everyone an instance of ModelSim is opened, further increases the problem. An upgrade of the software to a professional version could improve performances. Still, that ModelSim is not created to perform this type of tasks remains a fact.

The Device-based environment, which uses physical SE*cube*™ boards, requires much less CPU resources, as the computing part is made up by physical hardware

itself, which represents an active and not simulated computational unit. Therefore, the complexity of the simulation engine is moved from the host CPU to the actual hardware of the challenge. In this way, the role of providing the service to the user is split between the server and the SE*cube*™ board, that hosts the vulnerable design inside the FPGA circuit.

Furthermore, since in this environment a simulator such as ModelSim is not required, it is even lighter from memory usage point of view. Still, a major problem is anyway represented by the fact that electronics SE*cube*™ boards have a non-negligible cost.

7.4 Personal Considerations

Deciding which environment is better than the other is a choice that has no pure response. As usual, it may depend on many factors, ranging from the expected learning outcomes, the physical devices availability, the estimated cost of the system building to the hardware setup of the server that hosts the CTF competition. Both the environments are able to provide the service even remotely, so the location of the CTF competition is not a relevant factor.

With the environment based on the EDA tool, the main and real problem encountered has been the fact that ModelSim had to be adapted in such a way to accept a limited set of commands and put this service online. The first problems explored were focused on how to expose the service and to manage the interaction with the hardware simulator. Once solved, the focus has been moved on how to create a well-working environment for the filter feature of the `restricted shell`. This part also required a lot of testing activities, with exhaustive tries on corner-case commands that an user could insert.

A lighter software able to simulate generic hardware components would be more suitable than ModelSim, an example could be *GHDL*[1]. With GHDL an environment able to offer complete and partial black-box components and a user experience very similar to the environment based on ModelSim is possible to be developed, but clearly the learning outcomes are different because of the difference on the hardware simulator used (at least those related to the EDA-tool commands knowledge, even if the environment can be adapted accepting the ModelSim commands and emulating the same behavior).

Based on what we have seen so far, this type of challenges is better suited for large-scale Jeopardy-type challenges, thanks to the system scalability, still, the server hosting the competition must be able to substain the performance requirements of the challenge. An EDA-tool has been chosen to simulate the hardware because there is a direct parallelism between simulated hardware and the model

[1] https://github.com/ghdl/ghdl

that must be analyzed by the players. This is dictated by a *what-you-see-is-what-you-get* approach.

We wanted to keep the same approach for the environment based on physical devices, where the problems encountered were mainly due to the Attack/Defense part, and in particular when two SE*cube*™ boards were introduced for every team to reduce the programming latency that this phase intrinsically has.

The double board usage led to the development of an infrastructure capable of isolating hardware programmers and boards, in such a way to be able to select the wanted board for the programming and for the service phase. As explained above, this choice has been made in order not to make SLA penalty points invalidate the challenge. This type of challenge is best suited for small-scale challenges, where you want to get your hands on real hardware and ensure a more realistic experience.

7.5 Future Work

In the future, for the EDA-tool-based environment, it would be good to improve the system creating a multi-language environment capable to host designs written in other languages such as SystemC or Verilog. In this way, other type of hardware component models, or even systems and peripheral, models can be introduced (such as analog-to-digital converters). In this direction, it will be possible to create an environment where the vulnerable hardware communicates with a vulnerable software (for example, describing the whole system in SystemC). Another upgrade can be improving the performances with the current EDA-tool-based environment, trying to create a lighter system (i.e., replacing ModelSim with another simulator, finding restricted versions of it, or acquiring licenses that improve the performances of the simulator).

For the Device-based environment, different boards with larger FPGAs would be of help, in such a way to accommodate more challenges in a single FPGA or even an entire processors. It is also possible to think of a remote hybrid system where both EDA-tool-based and Device-based resources would be exploited, in order to obtain further versatility of the platform. For example, if the CTF organizer want to use the hardware in black-box mode, the requests are passed to the hardware-based modality, while if there are requests to force internal signals or read internal component states, it could be passed directly to the EDA-tool-based environment. In this way, the two environments can coexist, host the same challenge at the same time, and communicate between them.

Appendix A

EDA-tool-based Environment - Guideline

This Appendix is intended to give the reader instructions on how to set up the entire *EDA-tool-based environment* through a step by step guideline. The aim is to be able to read the guide, repeat the points, and create the service to which the CTF challenges participants have to connect.

A.1 Set-up the environment

In order to host the environment, there are some software requirements that the server has to take into account: `Docker` and `Docker-compose` have to be installed and the ModelSim `.run` file is required to be placed inside the `src/` folder described in the following.

The directory tree of the complete environment appears as follows:

```
/
├── docker-compose.yml
└── src/
    ├── Dockerfile
    └── CTF_env/
        ├── Environment/
        │   ├── restrictedShell.py
        │   └── Hardware/
        │       ├── model_runner.tcl
        │       └── ctf_hw/
        └── CTF_design/
            ├── CTF1/
            ├── CTF2/
            └── CTF3/
```

- `docker-compose.yml`: this is the YAML file used by docker-compose to take the information to start the services, as the port number of every challenge, the IP-addresses accepted for the connections, or the name of some environmental variables used by the environment to run the hardware challenge.
- `src/`: the `Dockerfile` used to automate the sequence of steps for the correct behavior of the environment is inside this folder with the remaining core part of the environment, moreover the `.run` ModelSim file has to be placed here.
- `restrictedShell.py`: is the Python script in charge of opening ModelSim, filter the user commands and communicate them retrieving the responses of the hardware simulator. It is the real core of the entire environment, with it the user is able to communicate with ModelSim, in fact it represents the service exposed by the environment.
- `Hardware/`: is the folder where the selected challenge VHDL files are copied inside (for the precision inside `ctf_hw` folder), and a file in charge of performing the compilation and starting the simulation of ModelSim (`model_runner.tcl`) is present.
- `CTF_design/`: it is the folder where the set of hardware VHDL challenges files have to be placed, the restrictedShell takes the files of the selected challenge from this folder copying them inside `ctf_hw/`. The sets of VHDL files have to be placed in different sub-folders for every challenge.

In order to add a new challenge the organizer has only to put the relative information in the YAML file inserting the port number of the service and the environmental variables as the CTF challenge name and the top level component name. Then he has to place the VHDL files of the challenge inside a sub-folder in `CTF_design` folder. In order to expose the services present in the YAML file its only needed to write the following command: `sudo docker-compose up -d -build`, then the challenges images are built and the services are exposed.

A.2 The participant point of view

The user has to connect to the wanted challenge service knowing the server IP-address that hosts the challenge and the port number of the service. Then he can send the commands to the restricted shell and try to find the flag. Figure 5.1 shows the user perspective.

Appendix B

Hardware-device-based Environment - Guideline

An explanation of how to run a challenge step by step is given for both Jeopardy and Attack-Defense challenges.

The required software is:

- `Docker` to host the containers
- `docker-compose` to automatize the containers creation
- `STMCubeProgrammer` used to program the board with a firmware
- `Lattice Diamond` to do the synthesis and create the CTF-Component bitstream for FPGA to be placed inside the firmware.
- `gcc-arm-none-eabi` libraries that allows the firmware compilation.

For the first four of these software it is required to have the executable folder path placed inside the `PATH` variable, with the aim to run the scripts correctly.

To program the boards, the ST-Link debugger[1] is required.

B.1 Set-up the Jeopardy environment

Jeopardy is a branch of the `FPGABasedCTF` repository dedicated to the challenges on the FPGA in a jeopardy game. The environment is very simple: only a challenge custom executable file is needed in order to establish a communication with the FPGA.

The Jeopardy environment could be executed remotely or even in local.

[1] https://www.st.com/en/development-tools/st-link-v2.html

The directory tree of the complete environment appears as follows:

```
/
├── Sources/
│   ├── SETUP/
│   │   ├── CTF_DESIGN/
│   │   │   └── CTF1/
│   │   ├── FPGA_SYN/
│   │   ├── SECube-SDK/
│   │   ├── Compile_and_program_firmware.sh
│   │   ├── Compile_host_software.sh
│   │   └── Program_only.sh
│   └── PLAYER_SOURCES/
│       └── CTF1/
├── docs/
│   └── CTF1/
├── Dockerfile
├── Host_CTF
├── docker-compose.yml
└── Launch.sh
```

Inside the `Sources/SETUP/` folder, it is contained all the material that is needed to the challenge creator (i.e., the compilation scripts of the host and of the firmware, the synthesis script, and the programming script). This folder has not to be delivered to the challengers.

Specifically, for the remote version, the organizers should also put online the service and it could be done executing the `Launch.sh` script. This script will mount the board and set the environment for docker to isolate this board inside the container. If the board is just been programmed, wait some minutes before start the `Launch.sh` script with the aim to have the FPGA programmed and the board ready.

In the `Compile_host_software.sh` script, the organizer can choose the folder name inside `SETUP/CTF_DESIGN` where the `CTF-Component` description is contained.

In the `docker-compose.yml` file, the organizer could choose the port where host the service.

B.2 The participant point of view for Jeopardy

For the local version, the participants have only the `Host_CTF` file and they will run it in order to use the service. The hardware design (with the masked flags) is provided to players inside the folder `Sources/PLAYER_SOURCES/`, and here they can analyze the VHDL code.

Inside the `/docs` folder, for each CTF, there are documents with the scoreboard description and hints for challengers. For the remote challenge, the player should

connect through TCP at the IP address with a specific port given by the organizers. For both remote and local, the service is working correctly if the *WELCOME TO THE SERVICE!* string is returned.

B.3 Set-up the Attack-Defense environment

The participants are provided of two pairs of SE*cube*™ boards and ST-Link debugger: one pair used to expose the game service, and the other one used for patching and substituting when required.

The main scripts for starting the service, program the board and stop the service are inside the `Environment/Scripts` folder. Instead, inside the `Sources` folder, there are some useful scripts used to compile the various Host C++ programs and setup the challenge.

For all the phases of the game, different Docker containers are used in order to provide a secure and isolated environment. To setup the challenge and initialize the environment, the `CTF_MANAGER.sh` script should be launched and the automated steps should be performed. It is recommended to use the `Automatic setup` option (see Figure 6.7) to do all the steps required to setup the challenge.

This script will setup all the boards with programmers and places the first version of the firmware inside boards. Then, a service must be started with the aim to give the first access to arbiter, that will set the password for the group and the first flag.

The directory tree of the complete environment appears as follows:

```
/
├── Environment/
│   ├── Check/
│   │   ├── src/
│   │   │   ├── Check.sh
│   │   │   ├── Dockerfile
│   │   │   └── Machine_Check
│   │   └── docker-compose.yml
│   ├── Service/
│   │   ├── src/
│   │   │   ├── FLAG
│   │   │   ├── GROUP
│   │   │   ├── Dockerfile
│   │   │   └── Host_CTF
│   │   └── docker-compose.yml
│   └── Scripts/
│       ├── Check_Board.sh
│       ├── Program_Board.sh
│       ├── Start_Service.sh
│       └── Stop_Service.sh
├── Sources/
│   ├── PLAYER_SOURCES/
│   │   ├── CTF_DESIGN/
│   │   │   └── CTF1/
│   │   ├── FPGA_SYN/
│   │   ├── SEcube-SDK/
│   │   └── Compile_service_firmware.sh/
│   └── SETUP/
│       ├── Scripts/
│       │   ├── Compile_host_software.sh
│       │   ├── Compile_setup_firmware.sh
│       │   └── Setup_USB.sh
│       ├── SEcube-SDK/
│       ├── workspace_setup/
│       └── CTF_MANAGER.sh
├── Dashboard.sh
└── docs/CTF1/
```

The `SETUP` folder should not be given to the participants, and it is used only to setup the environment before that the challenge start. The `FLAG` file is updated by the arbiter interacting with the team as a normal player, but using a specific password that allows him/her to change the flag. The password is inside the `GROUP` file, that is set up before the challenge starts.

B.4 The participant point of view for Attack-Defense

From the user point of view, a dashboard is provided in order to simplify the experience. An example of the sequence of steps to be performed if we want to patch and then launch our new version of the hardware design is explained here, exploiting the `Dashboard.sh` script:

1. Launch the service on the unpatched board (the service should be always provided) simply choosing the right option with the dashboard, selecting the board ID of the service board (for example the ID 0 is used);

2. It it possible to patch the hardware design inside the `Sources/PLAYER_SOURCES/CTF_DESIGN/` directory;

3. The patched component must be synthesized and compiled, the new firmware is produced, selecting the option with the dashboard that will perform all these steps;

4. Now its possible to program the board that is not in service with the new version of the system, using the program option in the dashboard;

5. Once the board is ready, it is possible to switch the service to the patched board, so the Service phase is performed on the board that we have just programmed.

An image of how the dashboard appears is shown in Figure 6.8

Appendix C

EDA-tool-based Challenges Description - Examples

The challenges developed for the EDA-tool-based environment are created in a way to address different learning outcomes. In fact ,they are also designed with a different VHDL architectural style starting from a purely behavioral description style (such as `Cr4ck_the_CVM`) moving to a more structural architecture (such as `k3y_m4n4g3r`). The developed challenges are rated with a difficulty index based on some experience in hardware design. Also, the scoreboard description (i.e., the text that is showed to the participants, containing the challenge description) is reported.

C.1 Cr4ck_the_CVM

C.1.1 Scoreboard Description

```
OUR COFFEE VENDING MACHINES ARE OUT OF MONEY!

A technical paper has been found near a coffee vending machine;
it contains a VHDL description that we believe be part of design of
the vending machine itself.

This documents are sent to you with the aim to find a problem
inside the hardware description of the machine.
We suspect a trojan has been inserted by an untrusted designer
to allow her/him to get all money of the machine...

You can find additional details in the file:
- HW_1.05_Cr4ck_the_CVM.zip
that includes the VHDL files of the machine.
```

C.1.2 Challenge Description

`Cr4ck_the_CVM` was the first challenge designed. In order to solve it, the participant has to find an hardware trojan inside the VHDL files of the components, given to the challengers. Activating it, the participants are able to find the flag that must be submitted to the organizers. To solve the challenge, a little bit of reverse engineering skills and VHDL understanding is required to the participants. However, this is the simplest challenge designed. The difficulty rate has been estimated in 3/5.

C.2 Its_Too_Hot

C.2.1 Scoreboard Description

```
A non-reliable company gave us a peripheral IP core,
thought to work in a SoC environment, for controlling
the working temperature of the system.
The controller has to shut down the SoC when the
threshold temperature is reached.

Today, some of our customers have found some problems
on the behaviour of the circuit, the SoC was not
turned off (on_off signal remains stucked at 1)
even if the threshold temperature was reached
and their circuits were got burned!
Are you able to locate the problem? We are interested
on the state of the circuit when the problem appears.
We are sure that it is inside this peripheral...

You can find additional details in the file:
- HW_1.06_Its_Too_Hot.zip that includes
the VHDL description of the peripheral.
```

C.2.2 Challenge Description

The second designed challenge is `Its_Too_Hot`. As well in this challenge, the aim is to find an hardware trojan that causes a misbehavior of the circuit, but, in this case, the understanding of a structural description useful to activate the trojan is required to solve it. The difficulty rate has been estimated to be 4/5.

C.3 K3y_m4n4g3r

C.3.1 Scoreboard Description

```
WHAT IS INSIDE THIS COMPONENT?

An unknown hardware component has been found, and we want to
discover as much as possible about it. Doing some research,
it came out that it is a key manager. We also got its VHDL
description.

Looking to it, we observed that the vendor has forgot to
unsolder the Normal/Test pin.

This documents are sent to you with the aim of finding a way
to read its content.

You can find additional details in the attached file:

- HW_1.0X_K3y_m4n4g3r.zip

that includes the VHDL files of the key manager.
```

C.3.2 Challenge Description

The last designed challenge is `k3y_m4n4g3r`. It is the hardest designed challenge between those related to EDA-tool-based environment, and it addresses a different vulnerability with respect to hardware trojans. In this case, the topic of test infrastructures is interested. The participant has to exploit the scan chains of the circuit to extrapolate sensible information an consequently find the flag. To do that, he/she has to understand some circuit logic described in a structural description way. The difficulty rate of the challenge has been estimated in 5/5.

Appendix D

Hardware-device-based Challenge Description - Examples

The only challenge developed at the moment for Device-based environment is named `Access_Manager`, which is developed for both Jeopardy and Attack-Defense modality.

D.1 Access_Manager

D.1.1 Scoreboard description

```
The Access_Manager is a circuit that handles the access of
some users inside a specific set of rooms (identified by their
number, from room 0 to room 7). There is a set of users saved
inside the design that are able to enter inside specific rooms:

- user 0X0A00 inside room 0
- user 0X9B01 inside room 1
- user 0X1202 inside room 2
- user 0XC303 inside room 3
- user 0X6004 inside room 4
- user 0X4405 inside room 5
- user 0X9706 inside room 6
- user 0X0107 inside room 7

Moreover, there are other users able to enter inside
```

```
the rooms which code is not saved inside the hardware design,
but their access is allowed by a combinational logic
tree inside the design. The director's room (room 3)
can be accessed in the same way of the other rooms.

Every team has the same hardware design inside the FPGA
of the SeCube hosting the challenge, are you able
to enter inside the director's room of the other teams
without inserting one of the allowed user codes?

--------ONLY FOR ATTACK-DEFENSE-------------------------
Every team can reprogram the FPGA using the Dashboard.sh
script modifying the VHDL code of the design inside CTF_Design.
Remember to keep always the service on switching between
the two SeCube boards provided!
--------------------------------------------------------

-----------ONLY FOR LOCAL JEOPARDY----------------------
In order to communicate with the board you have to launch the
Host_CTF.exe file submitting the inputs of the hardware
design (in the VHDL order not counting the clk signal)
and looking to the response.
--------------------------------------------------------

To communicate with the SeCube FPGA you have to submit
the inputs of the hardware design
(in the VHDL order not counting the clk signal)
and looking to the response. Keep attention that the
inputs must be in a decimal format!
```

D.1.2 Challenge Description

This challenge consists in capturing a flag hidden in a hardware access manager. To extract the flag, the component design has to be understood. The challenge could be Jeopardy or Attack/Defense type, and the hardware design is flashed inside the FPGA of the SE*cube*™ board. In case of local Jeopardy challenge, the player should use the executable given by organizers to interact with the board and exploit vulnerabilities inside of it. The challenge component has a hardware bug inside the design, and players should try to do a forbidden access inside a specific room with a certain user. The learning expected by the players are how to reverse-engineering the RT-level description of a digital circuit in order to capture a flag and how the synthesis process works.

Bibliography

[1] R. Baldoni, R. De Nicola, and P. Prinetto. «Il Futuro della Cybersecurity in Italia: Ambiti Progettuali Strategici». In: Consorzio Interuniversitario Nazionale per l'Informatica - CINI, 2018. ISBN: 9788894137330. Chap. 4, pp. 80–86. ISBN: 9788894137330.

[2] Giulio Berra, Gaspare Ferraro, Matteo Fornero, Nicolo Maunero, Paolo Prinetto, and Gianluca Roascio. «PAIDEUSIS: A Remote Hybrid Cyber Range for Hardware, Network, and IoT Security Training». In: ().

[3] R. S Cheung, J. P Cohen, H. Z Lo, and F. Elia. «Challenge based learning in cybersecurity education». In: *Proceedings of the International Conference on Security and Management (SAM)*. The Steering Committee of The World Congress in Computer Science, Computer ... 2011, p. 1.

[4] *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. https://cwe.mitre.org/data/definitions/79.html. [Online; accessed 21-July-2020]. 2020.

[5] *CWE-89: Neutralization of Special Elements used in an SQL Command ('SQL Injection')*. https://cwe.mitre.org/data/definitions/89.html. [Online; accessed 21-July-2020]. 2020.

[6] *CWE-94: Improper Control of Generation of Code ('Code Injection')*. https://cwe.mitre.org/data/definitions/94.html. [Online; accessed 21-July-2020]. 2020.

[7] *Cyber Ranges - NIST*. https://www.nist.gov/system/files/documents/2018/02/13/cyber_ranges.pdf. Accessed: 2021-10-27.

[8] *CyberChallenge.it - about*. https://cyberchallenge.it/about. Accessed: 2021-10-27.

[9] *Cybersecurity Jobs Report: 3.5 Million Openings In 2025*. https://cybersecurityventures.com/jobs/. [Online; accessed 15-November-2021]. 2021.

[10] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. «From game design elements to gamefulness: defining" gamification"». In: *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*. 2011, pp. 9–15.

[11] *Docker container page*. https://www.docker.com/resources/what-container. Accessed: 2021-10-29.

[12] C. Eagle. «Computer Security Competitions: Expanding Educational Outcomes». In: *IEEE Security Privacy* 11.4 (2013), pp. 69–71.

[13] *Google Capture The Flag*. https://capturetheflag.withgoogle.com/.

[14] *HackDAC*. https://www.dac.com/Conference/HackDAC. Accessed: 2021-10-27.

[15] K. Huotari and J. Hamari. «Defining gamification: a service marketing perspective». In: *Proceeding of the 16th international academic MindTrek conference*. 2012, pp. 17–22.

[16] K. Leune and S. J Petrilli Jr. «Using capture-the-flag to enhance the effectiveness of cybersecurity education». In: *Proceedings of the 18th Annual Conference on Information Technology Education*. 2017, pp. 47–52.

[17] J. Mirkovic and P. AH Peterson. «Class capture-the-flag exercises». In: *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. 2014.

[18] *ModelSim manual*. https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7. Accessed: 2021-11-02.

[19] C. I. Muntean. «Raising engagement in e-learning through gamification». In: *Proc. 6th international conference on virtual learning ICVL*. Vol. 1. 2011, pp. 323–329.

[20] Paolo Prinetto and Gianluca Roascio. «Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy.» In: *ITASEC*. 2020, pp. 177–189.

[21] Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. «Hardware-based Capture-The-Flag Challenges». In: *2020 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE. 2020, pp. 1–8.

[22] *Reply Challenges*. https://challenges.reply.com/tamtamy/home.action. Accessed: 2021-10-27.

[23] *RHme - 2015*. https://github.com/Riscure/RHme-2015. Accessed: 2021-10-27.

[24] *RHme - 2016*. https://github.com/Riscure/RHme-2016. Accessed: 2021-10-27.

[25] *RHme - 2017*. https://github.com/Riscure/RHme-2017. Accessed: 2021-10-27.

[26] *Socat - getting started*. https://www.redhat.com/sysadmin/getting-started-socat. Accessed: 2021-11-12.