Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Hardware-based CTF
Project Report

Candidates:
Eugenio Ressa (s281642)
Fulvio Castello (s301102)
Umberto Toppino (s277916)

Referents:
Prof. Paolo Prinetto
Dr. Matteo Fornero
Dr. Vahid Eftekhari

Contents

List of Figures

List of Tables

Abstract

Capture-The-Flag challenges are one of the most recognized ways to prove anyone's skills and expertise in a specific cybersecirity-related topic. This type of event is usually executed as a contest between different individuals or teams, such that a given system has to be analyzed and/or interacted with in order to uncover its well-hidden secrets and master its range of capabilities.

The type of CTF competition which will be hereby discussed pertains the hardware sphere, and it consists in finding a concealed vulnerability inside a Block Cipher written in VHDL. Users can actually interact with a real board and test its functionality by connecting to the challenge itself inside any standard browser, with the final objective of recognizing the intended asset.

CHAPTER 1

Introduction

This project consisted in the creation of a brand new Capture-The-Flag Challenge, based on a Block Cipher design which can be made accessible to the participants by means of a web interface. Each contestant needs to test such device in order to gather information about its behavior, and has the possibility of analyzing the source code used to implement its architecture. Of course, the player has to possess an adequate level of experience in terms of digital design, whilst also displaying a sufficient understanding of the key concepts of hardware security. This event is thus conceived as an individual challenge, in which a specific flag (or equally, a brief textual description of the related vulnerability) has to be retrieved directly from the examination of platform itself, with no interaction among the individual contestants.

The main issue to be dealt with was represented by the way in which the whole application could actually be made available to the end user. To that regard, the low-level logic description of the circuit was implemented on a physical board in order to be accessed remotely, so that each player can easily interact with it by means of a simple and effective GUI on any common browser. Once the correct combination of inputs is found, the competitor has to be able to detect the exceptional event by observing the output signals and report it to the organizers, so that victory in the game can actually be announced.

The remainder of the document is organized as follows. In Chapter 2, the reader is introduced to the world of CTF challenges; in Chapter 3, a generic description of the application as a whole is reported; in Chapter 4, the previously mentioned concepts are very much expanded on, in such a way that the reader can appreciate all the technicalities and the motifs behind the adopted approach; in Chapter 5, the current state of the project is put into perspective with what was initially presented, while some further possible milestones are proposed as well; in Chapter 6, a very quick recap of the whole work is provided; in Appendix A, the reader is instructed on how to recreate a demo of this application; in Appendix B, there is a short explanation on how the internal API can be used to communicate with the challenge from the outside.

CHAPTER 2

Background

The main focus of this chapter is explaining what a Capture-The-Flag (CTF) challenge is, how it works and in which ways it can be carried out.

Capture-the-Flag events are cybersecurity competitions where each participant has to exploit one or more vulnerabilities purposefully included inside the target platform by means of personal experience, skill and knowledge about a specific field. The final objective is to gain access to a predefined asset by means of the so-called "flag". The latter can be defined as a unique string that is formatted in a competition-specific manner that, if captured, certifies the success in the challenge.

There can be three main types of such an event:

- **Attack/Defense**, in which two teams interact with a selected system in order to (respectively) breach into the same or protect its own integrity
- **Jeopardy**, where all the contestants are playing agains the organizers themselves, and never communicate with each other in any way: they just have to test the vulnerable application by themselves in order to understand it and individually report its flaws
- **King of the Hill**: a variation of the first category in which the competing teams rally with each other with the goal of holding control of the system for the greatest amount of time possible.

Many different topics can be covered by these contests, and the individuals always have to possess a more or less advanced understanding of the intricacies they comprise. Some typical examples are: binary, crypto, forensics, hardware, miscellaneous, networking and web. The CTF in question is jeopardy and hardware-related, being based on a vulnerable 256-bit Block Cipher that was implemented onto a remote board, accessible via its public IP (along with a dedicated port) thanks to an online webpage.

CHAPTER 3

Implementation Overview

3.1 Goal of the project

The scope of this project is to design a vulnerable hardware device, in which the fault is to be discovered by each participant alone. Such weakness gets triggered only when a specific combination of inputs is provided, making the whole design partially functional. The player can thus report either the actual flag itself (i.e. a numerical value) or a coincise description of his discoverings, in order to be declared as the winner.

3.2 Architecture overview

The top-level Block Cipher includes a LFSR, an ALFSR, a 1-bit adder with support for the carry bit and some additional logic aimed at increasing the complexity of the whole process. The plaintext is scanned serially by the LFSR, while the seed to the ALFSR is given by the user in a parallel fashion. Each bit coming out of the former register is added to the output of the other one, resulting in the final chipertext. The ackowledge signal is used in conjunction in order to report its validity (after the initialization of the device).

The main takeaway in this context is that the LFSR is indeed a cipher just by itself, as it does perform pseudo-random number generation based on its inputs, but its computations are perfectly linear, as they just involve XOR operations. Thanks to the contribution of the carry, it's immediately possible to insert a crucial non-linearity that ends up making the entire function irreversible, hence not trivial anymore.

3.3 Hardware platform

The design was implemented and tested on a PYNQ embedded board, equipped with a programmable logic equivalent to the Artix-7 FPGA and also a Cortex-A9 processor. The latter can run Python code that interacts with the Block Cipher, exposing the related functionalities as convenient APIs by means of dedicated libraries.

As the PYNQ also includes an ethernet port, it was possible to create a web server that could offer the challenge online by means of a predefined IP address, which returns the end-product: an interactive webpage.

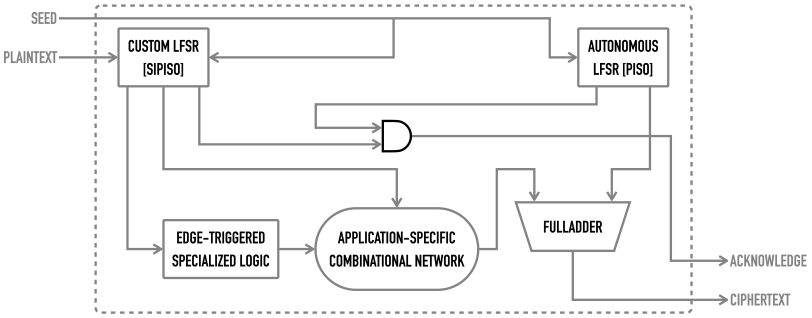

Figure 3.1: High-level schematic of the design.

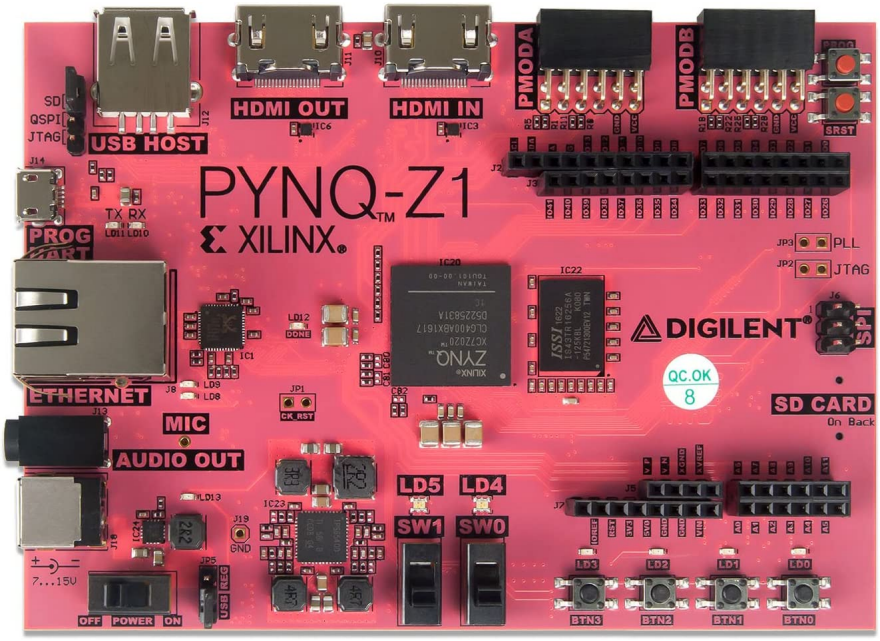

Figure 3.2: A PYNQ-Z1 Python Productivity board.

CHAPTER 4

Implementation Details

In this chapter, a bottom-up approach will be followed in order to present all the parts that make up the challenge itself:

- **Logic design**, the Block Cipher at Register Transfer Level, including all its internal components and subparts
- **Bistream**, the files coming from an HDL wrapper needed to configure the FPGA
- **Middleware**, the library used to link the design to the MCU on the embedded platform (i.e. *Overlay*)
- **Web server**, the framework used to build and host the web server directly on the board (i.e. *CherryPy*)
- **Front-end**, the web page displaying the GUI to the end-user.

4.1 Logic design

The lowest layer of the whole project consists of a vulnerable Block Cipher, implemented with a parallelism of 256 bits. This architecture is capable of taking a unique seed (*exactly* 256-bits long) together with a plaintext of variable size (*at least* 256-bits long) in order to generate the corresponding ciphertext, whose length will be determined by the latter input. Such process is based on two main PRNGs, and the operation should theoreticall be non-invertible: the vulnerability consists in eliminating the non-linearity that defines this very same property.

Note that all the source VHDL files describing the following components were commented extensively, and can be found in the "VHDL/VHDL.srcs/sources_1/new/" directory.

4.1.1 LFSR

A Linear Feedback Shift Register is represented by a set of D Flip-Flops linked in succession, where the output of each represents the input of the following one: the primary intake of this structure is represented by the D port of the first FF, whereas the Q pin of the last one serves as the main product. Such arrangement acts as the dividend of the operation that is being implemented: in fact, this kind of block performs nothing more than a division between polynomials.

The divisor is instead made up by a set of intermediate bits (that will be called "taps") which all go through an XOR port in order to generate a feedback signal that ends up in the first element of the chain. Their position can vary, but there exists only one combination of them that makes the related

polynomial primitive. This definition refers to the fact that, during its activity, a LFSR is capable of transitioning amongst a limited amount of states. When its divisor is a primitive polynomial, then it will be referred to as a **maximal-length** LFSR, meaning that it's able to cycle over all the 2^N-1 possible states (where N is the number of bits, i.e. 256 in this context).

Its computation is pseudo-random, as it always has to start from an initial value (called "seed") different from zero, and the sequence of phases it will trantition into will always be the same (depending on the hardwired taps it includes). In this case, the LFSR starts exactly in this *absorbing* state, being stuck with all its bits at 0, until the user begins to load the message that will need to be encoded.

As the vulnerability had to be well-hidden inside the design in question, this block needed to be heavily modified in order to include an additional internal entity, referred to as *scrambler* inside the code. The purpose of the latter is to provide the same sequential functionality of the ALFSR (see below), coupled with a convoluting process aimed at making its real purpose more obscure to the contestant.

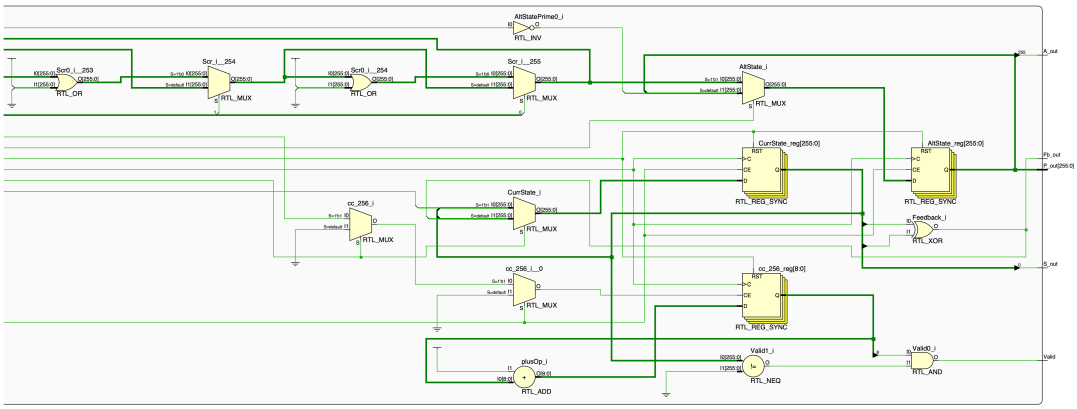

Figure 4.1: Noteworthy portion of the LFSR.

4.1.2 ALFSR

This device is essentially the same as the one described above, with the only difference that its seed is loaded parallelly only once, and then the register is left running virtually forever with no other user input. This is the actual reason for which this structure is given the attribute **autonomous**, as in

Autonomous Linear Feedback Shift Register (ALFSR). It represents the second main contribution to the encoded product of the Block Cipher, and its output signal is still serial.

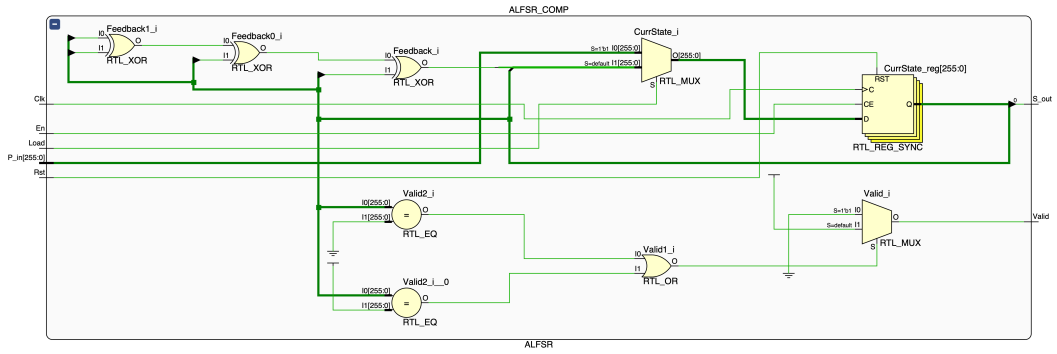

Figure 4.2: Schematic of the ALFSR.

4.1.3 FA

This is a simple fulladder with carry in and carry out support (implemented by means of a simple D Flip-Flop). It is characterized by the following equations:

$$S = A \oplus B \oplus C_{in} \tag{4.1}$$

$$C_{out} = (A \cdot B) + (A \cdot C_{in}) + (B \cdot C_{in}) \tag{4.2}$$

where A represents the output of the combinational network (detailed later), or in principle the contribution of the LFSR, while B is the output of the ALFSR.

4.1.4 EDGE

This circuit represents an enabler for the vulnerability: as soon as its main output is asserted, the whole Block Cipher gets trivially reduced to the LFSR only. The two key points are: how the purposefully-injected fault is triggered, and in which way the extreme simplification of the top level design is achieved.

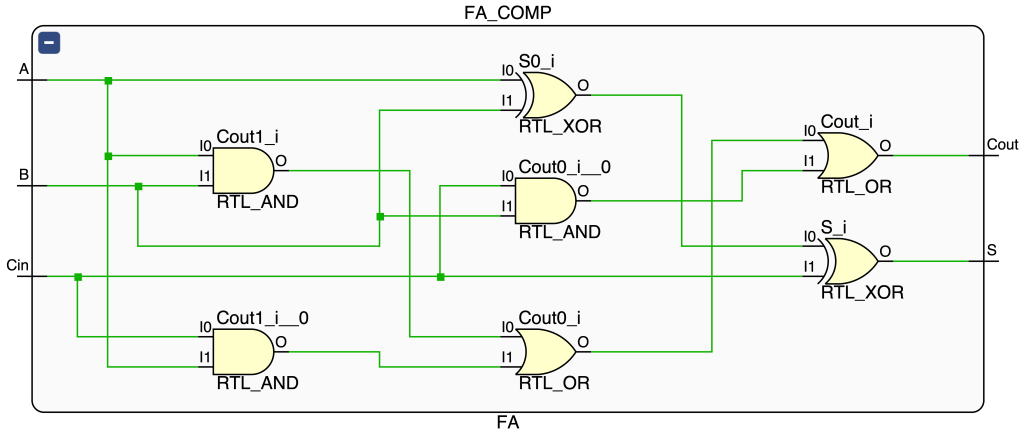

Figure 4.3: Schematic of the fulladder.

Starting from the first matter, this architecture was implemented by means of a peculiar Mealy State Machine which closely monitors the feedback signal of the LFSR during its initialization. If such bit goes to 1 exactly once every four clock cycles (strictly: 0001, 0001, 0001, ...), then, whenever the LFSR acknowledges the validity of its output, the special internal enable is activated. Note that the final value for its principal output signal gets sampled on a rising edge of the valid bit: this was the main reason for which this structure was called "EDGE". Instead, as soon as even a single feedback does not respect the above sequence, the product of this block gets irreversibly stuck at 0 (or until the next reset, at least).

On the other hand, the vulnerability itself is achieved in conjunction with the aforementioned *scrambler* and the combinational network, in such a way that the contribution of the ALFSR simply gets omitted from the final operation. More details will follow.

4.1.5 NET

This combinational network represents the last element that makes up the vulnerability of the design in question. It has three main functionalities:

1. completing the "secondary ALFSR" located inside the real LFSR by creating the feedback network for the so-called *scrambler*

2. computing the sum between the outputs of the LFSR and of the ALFSR′, with support for the

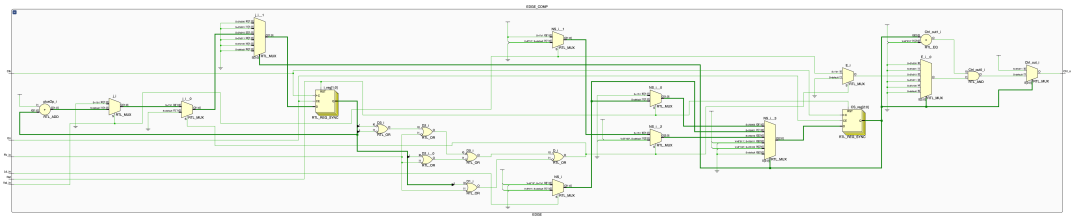

Figure 4.4: Schematic of the enabler.

carry bit as well

3. estimating the carry out of the previous arithmetic operation.

This organization of logic gates is aimed at recreating the same output of the main ALFSR in order to put it in XOR with the product of the LFSR, effectively anticipating the contribution of the subsequent fulladder. The hardcoded equation for the bit that will represent the first operand of the main sum in the top level structure is:

$$NET_{out} = LFSR_{out} \oplus ((ALFSR'_{out} \oplus C'_{in}) \cdot EDGE_{out}) \tag{4.3}$$

where the output of ALFSR′ and the carry in are identical to the "real" ones, but are computed in a much more cryptic and concealed way (the taps of the scrambler are the same as the ones found in the primary ALFSR, which in turn features a different divisor from the one hardcoded in the LFSR—however, all these registers are maximal-length).

Note that this whole functionality is gated by the output bit of the enabler: if the latter remains at 0, then the output of this network will be equal to the contribution of the LFSR only, and the user won't notice any defect in the response.

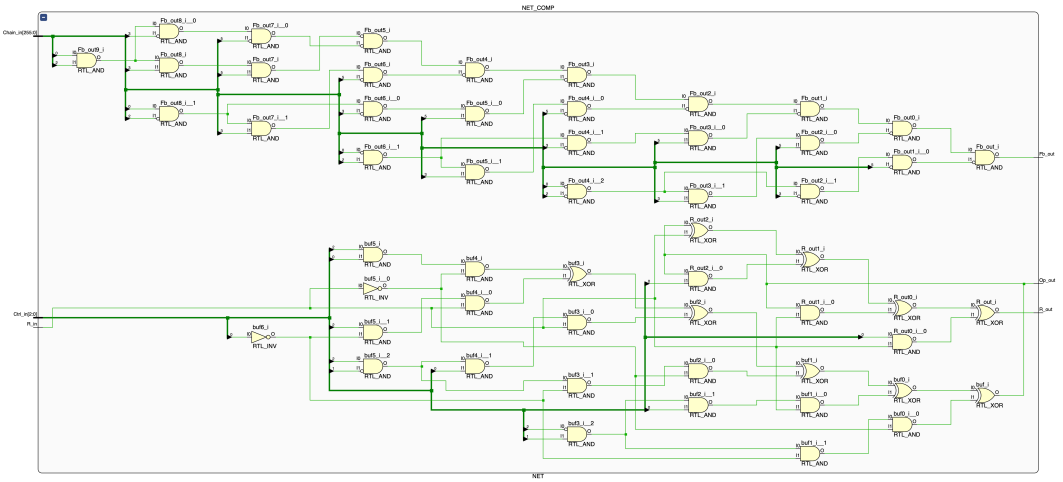

Figure 4.5: Schematic of the combinational network.

4.1.6 BlockCipher

The top level entity is thus made of the two main contributions of the LFSR and the ALFSR, the outputs of which are added one by one after going through the application-specific additional logic. The ciphertext is marked as "valid" only after both registers are loaded correctly: this information is conveyed by means of a dedicated output bit as well.

Looking at it from this level, the vulnerability can be attributed to an additional ALFSR, here called ALFSR′, which is disassembled and spread across the whole design: the register part (*scrambler*) is inside the LFSR, while the XOR for feedback creation is inside the combinational network. On the other hand, also the fulladder is regenerated again within the "NET", in such a way that the actual output of the primary fulladder will be represented by:

$$FA_{out} = LFSR_{out} + ALFSR_{out} - (ALFSR'_{out} \cdot EDGE_{out}), \quad ALFSR_{out} = ALFSR'_{out} \tag{4.4}$$

which represents the final outcome of the encryption process.

Note that all the debug signals required by the challenge and both the principal products (ciphertext and valid bit) are rendered sequential by means of output D Flip-Flops, in order to make the interface with the subsequent layers possible.

Lastly, this implementation was tested by means of an appropriate testbench, located inside the "VHDL/VHDL.srcs/sim_1/new/" directory together with many useful input memory files. Any simulation of "tb_BlockCipher.vhd" would lead to the creation of several output files in as well

(placed in "VHDL/VHDL.sim/sim_1/behav/xsim/"): they include the collection of values that each signal shown on the final webpage takes on during the whole runtime, in order to handily compare them and check their correctness.

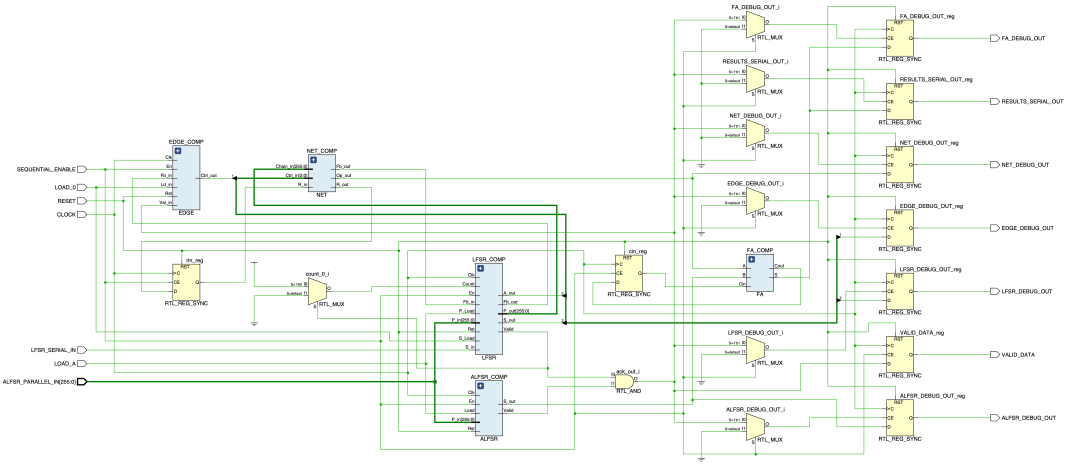

Figure 4.6: Entire schematic of the Block Cipher, showing the internal connections between all its fundamental components.

4.2 Bitstream

The design described above had to be converted into a custom IP, which was in turn included within an HDL wrapper in order to go through the necessary steps of implementing its logic functionalities onto the embedded board. A detailed documentation aimed at this goal was kindly provided by **Coralie Allioux**, who created the *Tutorial Vivado 2020.2.pdf* file expressely for this reason.

The Vivado project used alongside that paper is located in the "BlockCipher/project_1/" directory, while "BlockCipher/ip_repo/" was automatically created and contains the required file structure for the aforementioned IP. For the sake of convenience, the input netlists of the bitstream generation process are duplicated inside "PYNQ/Board_Files/src/", whereas "PYNQ/Board_Files/gen/" contains the resulting files needed to configure the FPGA on the remote platform (**.bit**, **.hwh** and **.tcl**). Please note that these three MUST have the same name in order to properly work with *Overlay*, so a refactoring will likely be needed before sending them to the PYNQ.

Name	Offset
ALFSR_PARALLEL_IN7	0x00
ALFSR_PARALLEL_IN6	0x04
ALFSR_PARALLEL_IN5	0x08
ALFSR_PARALLEL_IN4	0x0C
ALFSR_PARALLEL_IN3	0x10
ALFSR_PARALLEL_IN2	0x14
ALFSR_PARALLEL_IN1	0x18
ALFSR_PARALLEL_IN0	0x1C
LFSR_SERIAL_IN	0x20
SEQUENTIAL_ENABLE	0x24
CLOCK	0x28
RESET	0x2C
LOAD_0	0x30
LOAD_A	0x34
LFSR_DEBUG_OUT	0x38
ALFSR_DEBUG_OUT	0x3C
FA_DEBUG_OUT	0x40
EDGE_DEBUG_OUT	0x44
NET_DEBUG_OUT	0x48
RESULTS_SERIAL_OUT	0x4C
VALID_DATA	0x50

Table 4.1: AXI4 peripheral registers addressing.

All the signals that make up the I/O list of the Block Cipher had to be instantiated as 32-bit registers inside the AXI4 peripheral because of a software limitation of Vivado: this way, the parallel in going into the ALFSR (originally 256-bits long) had to be split into 8 different registers that would later get concatenated internally. A table containing the current addressing offsets of such registers is here provided: those are the memory locations that higher-level layers will need to access when trying to interact with the logic design.

4.3 Middleware

PYNQ is an open-source project made by Xilinx® to better take advantage of the functionalities included in the embedded microcontroller. Using Python libraries, designers can exploit the benefits of programmable logic and microprocessors to build more capable electronic systems.

The related SoC is based on a dual-core ARM® Cortex®-A9 processor (referred to as the Processing System or PS), integrated with FPGA fabric (the Programmable Logic or PL). The PS subsystem includes a number of dedicated peripherals (memory controllers, USB, UART, I^2C, SPI, etc.) and can be extended with additional hardware IP in a PL architecture.

Overlays, or hardware libraries, are to be intended as an interface between the Processing System and the FPGA. This way, they can be used to accelerate a software application or, in general, to integrate it with a hardware layer. In this context, *Overlay* is used to transform the Block Cipher into a Python script.

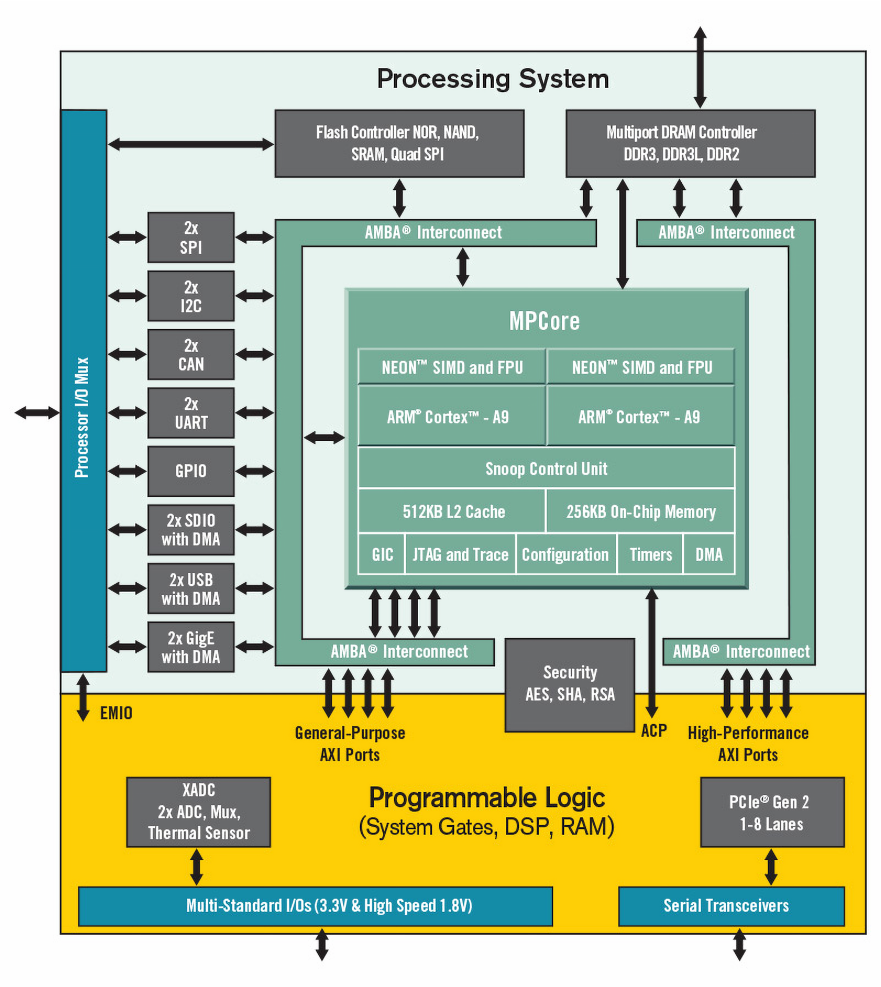

Figure 4.7: Schematic of a Zynq SoC.

4.3.1 Board interfaces

The Zynq has 9 AXI registers used to interface between the PS and the PL. On the PL side, there are 4 AXI Master HP (High Performance) ports, 2 AXI GP (General Purpose) ports, 2 AXI Slave GP ports and 1 AXI Master ACP port. Python code running on PYNQ can access the IP connected to an AXI Slave, which in turn is linked to a GP port (MMIO is used to achieve this association).

Any IP connected to the AXI Slave GP port will be mapped into the system memory map: MMIO can be used access all mapped locations, in such a way that any read/write command is a single transaction to transfer 32 bits of data from/to a memory section. The **.hwh** file is responsible, among other functionalities, of storing the memory map of the selected architecture: inside the <MODULES> section, the <ADDRESSBLOCKS> tag specifies the offset associated to each port of the design.

Note that all registers have a fixed size (e.g. 32-bits), hence, whenever a single bit has to be managed by means of the *std_logic* VHDL type, it will be sufficient to bind the corresponding signal to just one bit of it (the LSB in this case).

4.3.2 Driver

Starting from the outcomes of the bitstream generation process, the **.bit** file is loaded inside the PL, while the **.hwh** is used for the interface with the PS. Once an overlay has been instantiated, it is linked to the driver that actually calls the read/write instruction, making the commun ication with the IP possible.

The **CipherDriver** (cfr. "PYNQ/UserInterface/CypherDriver.py") is a class used to expose a method that performs the communication to the PL. Starting from the input values of the Block Cipher sent by the GUI (more on that later), it returns a JSON containing the relevant outputs of the same design.

4.4 Web server

The previous services were then made accessible from remote by means of a web server running on the PS side of the board. Such back-end is based on the *CherryPy* Python web framework, which links local resources to a public URL.

Inside the *web_server.py* file (placed in "PYNQ/UserInterface/") the application is launched and is capable of managing HTTP requests coming from an external host. This function is then linked to the IP address of the network interface of the board itself, with a specified port (where 8080 is the default). Then, whenever a POST is requested from the client, the server responds by means of a GET that delivers the *webpage.html* file (located in the same folder), containing the dynamic data that the custom IP itself will have returned.

4.5 Front-end

The Graphical User Interface was here implemented by means of a webpage accessible through any popular browser. This solution is thought to be the best choice considering the ease of use, availability of functionalities and general accessibility.

The GUI is made up of four different parts:

- static HTML webpage
- CSS-based interactive elements
- several JS scripts
- CTF challenge source code and previews.

4.5.1 Static HTML

Right after connecting to the provided URL, the contestant is presented with a straightforward webpage which contains four main sections:

- the challenge background, which gives the player some basic information regarding its scope, while setting the mood of the game thanks to a short backstory
- a table containing the list of components that make up the implemented logic, together with the links to download and/or preview the source code
- the input form containing two fields to be filled by the participant—above this segment, a brief paragraph gives the user some additional information regarding the accepted formats of the input data

- many output labels that will populate with the response coming from the PYNQ, including both the main output signals of the Block Cipher and a few other relevant signals that are deemed very important for debugging purposes.

The HTML heavily relies on *Bootstrap*, a popular front-end library widely used in web development that greatly simplifies the formatting and adaptability of the webpage.

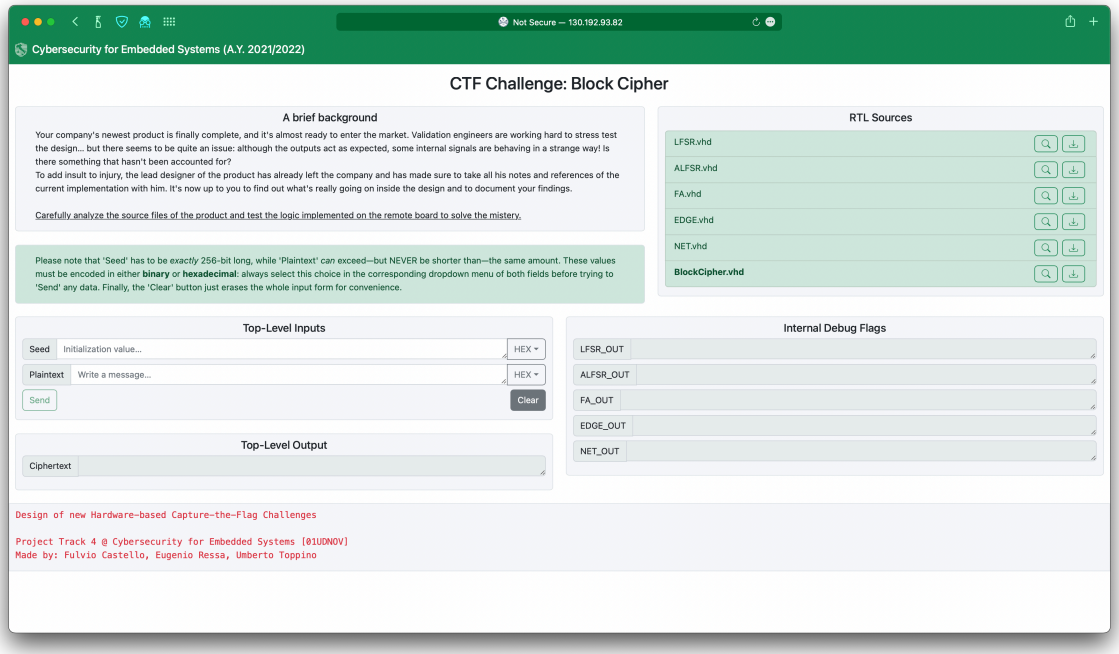

Figure 4.8: Homescreen of the challenge, zoomed out in order to display all the described elements.

4.5.2 JavaScript snippets

The JavaScript code implements many functionalities that would otherwise not be possible by HTML alone:

- all input and output fields are resized automatically depending on their content—this greatly improves readability and makes the flow of the challenge much smoother
- the preview system of the VHDL netlists is implemented through JS
- the 'Clear' button resets the whole input form for convenience
- the actual communication between the front-end GUI and the back-end hosted on the PYNQ is carried out thanks to Ajax POST requests; more specifically:
  1. the used fills the input data and presses the 'Send' button
  2. the input data is used to generate a POST request that is delivered to the back-end server
  3. the back-end, after elaborating the data and collecting the output from the FPGA, responds to the Ajax request with a string to be parsed into a dictionary

4. a JS script collects the incoming object and uses its content to populate the output fields of the HTML, without the need to reload the webpage.

- both input text fields are checked locally in real time (rather than on the *CherryPy* server) to make sure the data being written is well formatted and doesn't contain invalid messages—the user is not able to send the information unless all parsing constraints are satisfied.

Regarding the last point, it's important to notice that there exist a couple of edge cases that need to be taken into consideration: if any of the three PRNGs (LFSR, ALFSR and ALFSR′) never leaves the "absorbing" state, it remains forever stuck and the acknowledge signal will not display the required rising transition. As the middleware of this application waits for that particular edge of the valid flag in order to return the correct values, it is in fact crucial to avoid this type of situation in order not to end up inside an infinite loop. Therefore, both the seed and the plaintext must not be made of all zeros; additionally, the former cannot be formed by all ones either, due the related *scrambler* getting loaded with the complemented version of such value (i.e. all zeros, again).

The "PYNQ/UserInterface/webpage.html" file contains, among many other pieces of code, the snippets dedicated to this kind of checking: if any of the input fields passes the initial length test, but would lead to the previously mentioned infinite loop, the request is not even forwarded to the board at all, while every output label gets abruptly overwritten with the message **PROCESS TIMEOUT** to alert the user.

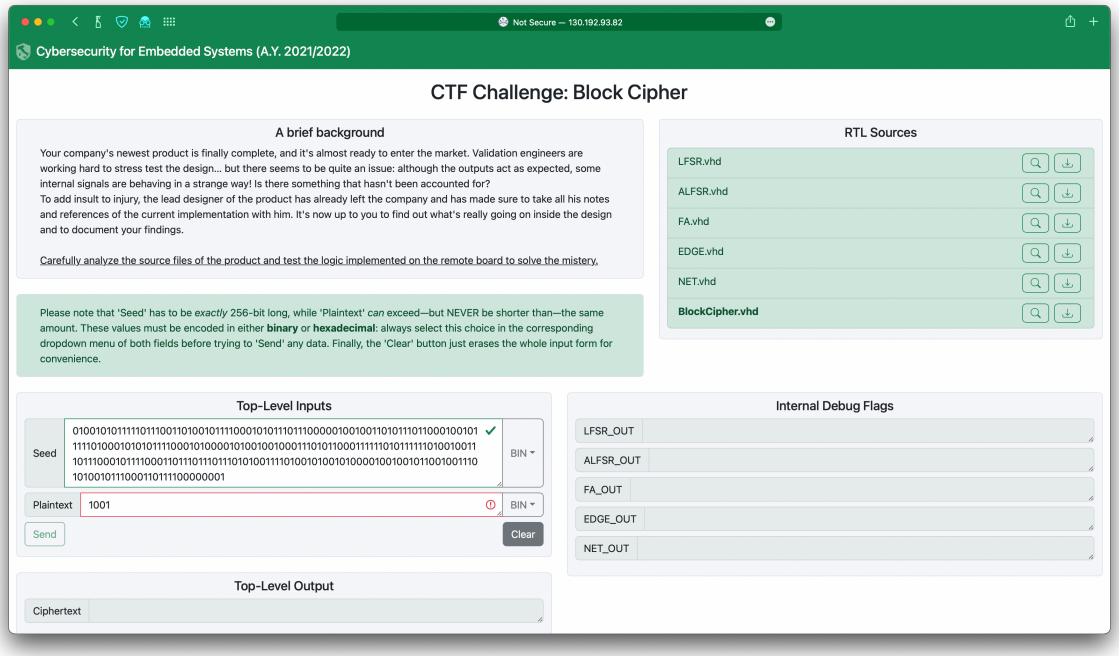

Figure 4.9: Example of failed parsing rules in the input form: the 'Send' button is disabled.

4.5.3 Preview system

A key part of the challenge is the ability for the user to access the source code to try and reverse engineer it, possibly finding the correct solution. As explained in the previous section, the webpage contains a list of links to download the source files and to preview them directly online.

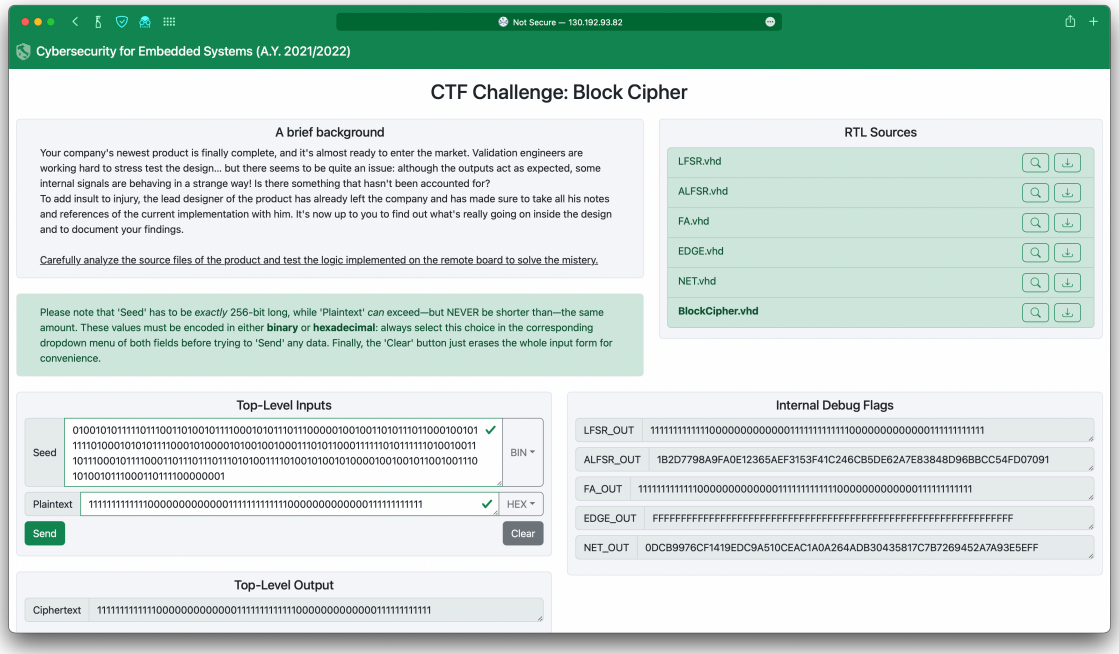

Figure 4.10: Successful request + response (including contest victory).

Each of the preview buttons launches a pop-up (more specifically a *modal window*) which captures the source code of the chosen source file. The latter is already well formatted and color-coded inside the corresponding external HTML file, having been generated by means of the contents of "PYNQ/Previews/" in the following way:

1. cut-paste the desired VHDL netlist inside *test.html*
2. open the latter inside any browser—the *run_prettify.js* script will be launched automatically (all credits go to **Google Inc.**)
3. save the newly visualized webpage as a **.html** file with an arbitrary name, and move it to "PYNQ/UserInterface/previews/".

This way the VHDL source code is visible on the fly, without the need to download it on a local machine to be opened with a suitable text editor/IDE.

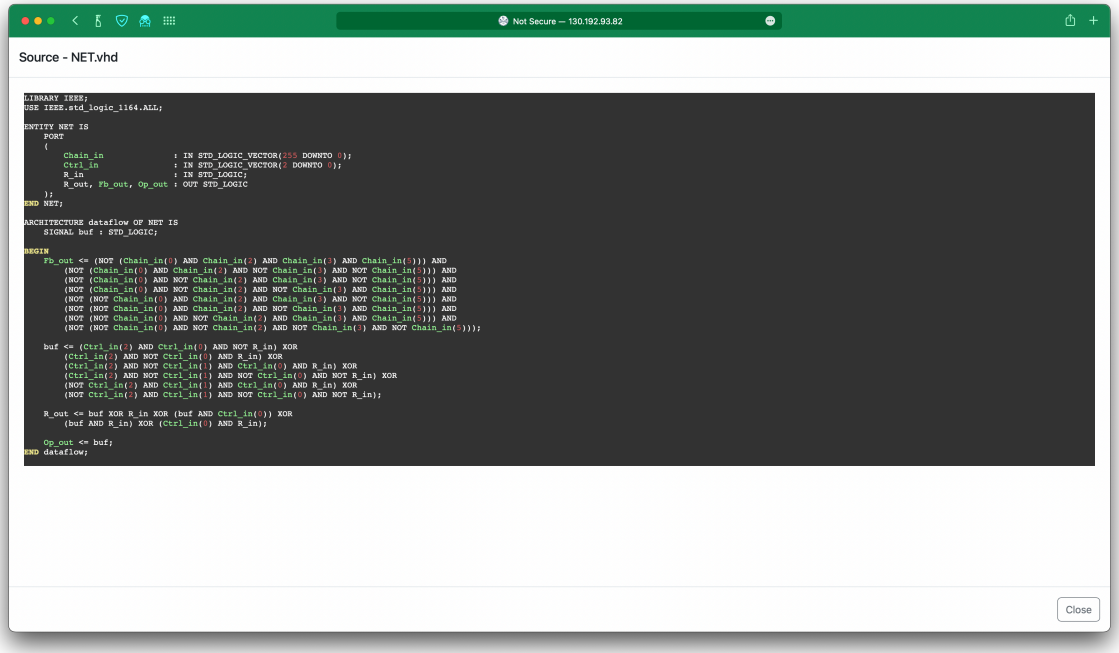

Figure 4.11: Preview page of the combinational network.

CHAPTER 5

Results

The CTF challenge was successfully implemented and deployed as described above: it's already available and 100% functional, provided the web server is launched beforehands and the remote board is accessible. Testing can be done by connecting to `http://130.192.93.82:8080/`, and the minimal form flag was found by means of the *flag_main.cpp* C++ script (located in the "VHDL/" directory): $\mathbf{0001000100010001000100010001000100010001000100010001000000000000000000000000000000000000000000000000000000000001000100010001000100010001000100010001000100010001000100010000000000000000000000000000000000000000000000000000000000000100010001000100010001000100010001000100010001}_{\mathbf{2}}$ or equally $\mathbf{1111111111111000000000000011111111111110000000000000111111111111}_{\mathbf{16}}$.

Be careful when copy-pasting these values directly into the online webpage, as they include whitespaces and newline characters in order to properly fit into the portrait page layout of this PDF file!

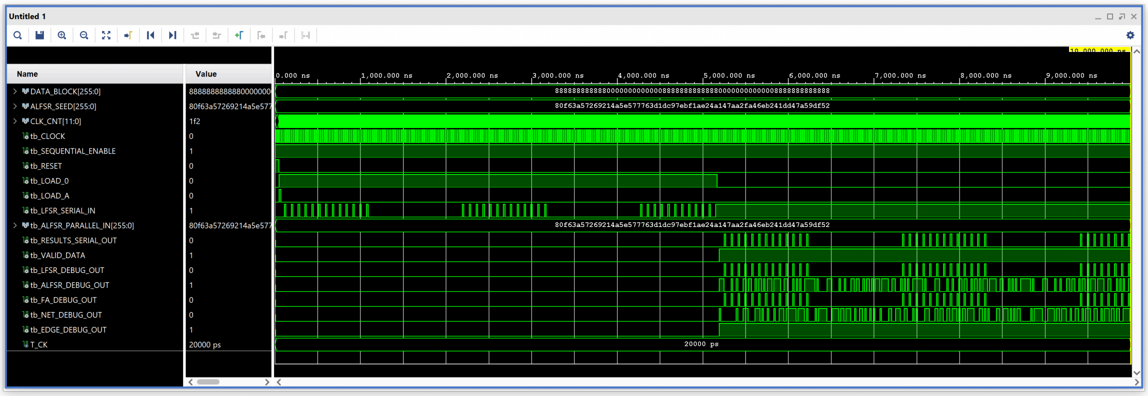

Figure 5.1: Simulation of the vulnerability being triggered.

5.1 Known Issues

There are no known issues for the current state of the project, so this section is devoted to listing the differences between the initial demo and the final version instead:

- The favicon of the website was changed in order to be aligned with the one found on the official website of the *Politecnico di Torino* institute
- The 'Reset' button was removed from the GUI, as its functionality gets implicitly called every time the user presses 'Send' instead. The initial idea was to have the possibility of loading the plaintext with one or more breaks before reaching the acknowledge, as to leave the LFSR running and going through many more intermediate states. This concept was eventually scrapped as it didn't provide any substantial benefit to the complexity of the challenge, while creating many more implementation struggles and negatively impacting the synchronization side of the multi-user aspect
- The last screenshot of the original presentation included a preview of the *EDGE.vhd* source file: the latter had to be totally reworked during the final stages of develompent, after some (tricky to detect) defects were discovered inside the the corresponding design.

5.2 Future Work

The provided web user interface will probably serve as more of a mockup, while the final challenge will have to be included within an official platform by the real organizers. This process will need to adhere to the instructions detailed in Appendix B.

Two possible improvements to the low-level design can also be discussed: the implementation of the 'Reset' button, as described above, and the conception of a much more sophisticated vulnerability (produced by a real team of cybersecurity experts, for instance), referring to the enabler, the combinational network and the *scrambler* components.

CHAPTER 6

Conclusions

As a recap, the most interesting characteristic of this project is for sure its heterogeneity in terms of levels of abstraction in which it operates: starting from the lowest layer, the logic design is implemented onto the FPGA, which in turn communicates with the Python code running inside the MCU of the board. This way, the main methods are exposed online as APIs to be called via HTTP requests, and a web application made of HTML/CSS/JavaScritpt code performs the final processing over the transmitted data in order to make it human-readable.

This CTF challenge requires the use of a real microcontroller system, and involves the knowledge of many programming (and design) languages, in such a way that a whole chain of hardware and software modules need to collaborate in an organic way. The result is ultimately represented by a kind of "pipe" that conveys information from the simplest electronic elements on a remote fixed location, all the way to the screen of the end-user: wherever they are, and whatever device, browser or internet connectivity they might be using.

APPENDIX A

User Manual

The application has to be managed through a command line:

1. connect remotely via ssh:

   ```
   ssh challenge@130.192.93.82
   ```

2. insert the password—login credentials will have to be requested directly from the people managing the physical board
3. after successful authentication, use the newly opened terminal
4. get root privilegies:

   ```
   sudo -i
   ```

5. clone the "PYNQ/" directory inside the board (there are no specific requirements about the destination)
6. install both **pynq** and **cherrypy** libraries (the latter needs Python 3.6 or above)
7. navigate to "PYNQ/UserInterface/"
8. start the backend:

   ```
   python3 web_server.py
   ```

9. the challenge is now up and running at `http://130.192.93.82:8080/`.

In case the logic design needs to be modified, a new bitstream will have to be generated accordingly. The resulting files will all have to be named "design_1" (as detailed in Section 4.2) and to be pushed to the board again:

```
scp path/local/file challenge@130.192.93.82:PYNQ/Board_Files/gen
```

where the destination was previously defined during step 5.

APPENDIX B

API

This CTF challenge can be easily integrated inside external platforms by querying

`130.192.93.82:8080/chipe2`

with a POST HTTP request, sending as body a *single line string* representing a JSON that would contain the following concatenated keys:

- **seed**: the seed of the ALFSR
- **seedFormat**: the format of the seed ("BIN" or "HEX")
- **message**: the plaintext to be encoded
- **messageFormat**: the format of the message ("BIN" or "HEX").

Please adhere to the following randomized example:

```
{"seed":"A5AB23178ACAE61268591959195ACAC4124120910293012 93B12094923587128","seedForm
at":"HEX","message":"A5AB23178ACAE61268591959195ACAC4124120910293012 93B1209492358712
8","messageFormat":"HEX"}
```

of course, changing values and encodings according to the specific use case (and again, be careful when copy-pasting directly from here).

The response will include again one single string formatted with the same principle as before, only this time featuring:

- **output_CP**, enclosing the chipertext
- **output_LFSR**, enclosing the output of the LFSR
- **output_ALFSR**, enclosing the output of the ALFSR
- **output_EDGE**, enclosing the output of the enabler
- **output_NET**, enclosing the output of the combinational network
- **output_FA**, enclosing the output of the fulladder.

As a final note, remember to perform the parsing of the input fields *before* calling this API, as the latter doesn't perform any check on the information that's being handled (i.e. the length of the seed and the plaintext, their format and the possibility of indefinitely falling into the "absorbing" state of both PRNGs).