Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Design and Development of a RAM-based PUF

Project Report

Candidates:
Zissis Tabouras (s284685)
Elena Roncolino (s304719)
Stefano Palmieri (s281677)

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

Contents

List of Figures

List of Tables

Abstract

This is the space reserved for the abstract of your report. The abstract is a summary of the report, so it is a good idea to write after all other chapters. The abstract for a thesis at PoliTO must be shorter than 3500 chars, try to be compliant with this rule (no problem for an abstract that is a lot shorter than 3500 chars, since this is not a thesis). Use short sentences, do not use over-complicated words. Try to be as clear as possible, do not make logical leaps in the text. Read your abstract several times and check if there is a logical connection from the beginning to the end. The abstract is supposed to draw the attention of the reader, your goal is to write an abstract that makes the reader wanting to read the entire report. Do not go too far into details; if you want to provide data, do it, but express it in a simple way (e.g., a single percentage in a sentence): do not bore the reader with data that he or she cannot understand yet. Organize the abstract into paragraphs: the paragraphs are always 3 to 5 lines long. In LaTeXsource file, go new line twice to start a new paragraph in the PDF. Do not use to go new line, just press Enter. In the PDF, there will be no gap line, but the text will go new line and a Tab will be inserted. This is the correct way to indent a paragraph, please do not change it. Do not put words in **bold** here: for emphasis, use *italic*. Do not use citations here: they are not allowed in the abstract. Footnotes and links are not allowed as well. DO NOT EVER USE ENGLISH SHORT FORMS (i.e., isn't, aren't, don't, etc.). Take a look at the following links about how to write an Abstract:

- `https://writing.wisc.edu/handbook/assignments/writing-an-abstract-for-your-research-paper/`
- `https://www.anu.edu.au/students/academic-skills/research-writing/journal-article-writing/writing-an-abstract`

Search on Google if you need more info.

CHAPTER 1

Introduction

In the last years, the number of small electronic devices that can be connected with big computational units grew exponentially. Embedded systems play a crucial role in fueling the growth of the Internet-of-Things (Iot) in the most diverse domains, such as health care, home automation and transportation. By the end of 2022 the number of IoT devices connected to the Internet is expected to reach the astonishing number of 14.4 billions [1]. The ubiquitousness of such devices coupled with their ability to access potentially sensitive/confidential information has given rise to security and privacy concerns. An additional challenge is the growing number of counterfeit components in these devices, resulting in serious reliability and financial implications.

Physical unclonable functions (PUFs) are a promising security primitive to help address these concerns. PUFs extract secrets from physical characteristics of integrated circuits (ICs) [2] and therefore require minimal or no additional hardware for their operation and are therefore cheaper than other solutions. The instance-specific nature of the secret provide a mean to to uniquely identify and authenticate each device based on a challenge-response mechanism [3].

The aim of this project is to design and develop a RAM based PUF for the SEcube™, a single-chip easily integratable device capable of hiding significant complexity behind a set of simple high-level APIs [4].

The remainder of the document is organized as follows:

In Chapter 2, a brief background and state of the art of this topic is provided;
In Chapter 3, an implementation overview is presented;
In Chapter 4, implementation details are described;
In Chapter 5, results are listed;
In Chapter 6, conclusions and final observations are presented.
Appendix A describes how a demo of the implementation can be run.
Appendix B describes the APIs created for this project.

CHAPTER 2

Background

2.1 State of the art of embedded systems security approach

The current best practice for providing a secure memory or authentication source in mobile systems is to place a secret key in nonvolatile electrically erasable programmable read-only memory (EEPROM) or battery-backed static random-access memory (SRAM) and use hardware cryptographic operations such as digital signature or encryption. Nonetheless, this approach is expensive both in terms of design and of power consumption. In addition, invasive attack mechanisms make such nonvolatile memory vulnerable. Protection agains such attacks is therefore needed and it requires the use of active tamper detection/prevention circuitry which must be continually powered [2].

2.2 Physical unclonable functions

Physical unclonable functions (PUFs) are innovative primitives that derive secrets from complex physical characterstics of the ICs rather that storing the secret in digital memory. Because the PUF taps into the random variation during an IC fabrication process, the secret is extremely difficult to predict or extract. PUFs generate volatile secrets that only exist in a digital form when a chip is powered on and running. This requires the adversary to mount the attack while the IC is running and using the secret, which is significantly harder that discovering non-volatile keys. An invasine attack must measure the PUF delays without altering them or triggering sensing wires that clear out the registers [5].

The concept of PUFs is based on the idea that even though the mask and manufacturing process is the same during the creation of the same type of IC, each IC is actually slightly different due to normal manufacturing variability. PUFs leverage this variability to derive the silicon "biometric", a "secret" information that is unique to the chip. This implies that no two identical chips can be manufactured. Although the use of PUFs is a relatively new technology, it should be noted that the concepts of unclonability and uniqueness of ojects have been extensively used in the past [2].

2.2.1 Types of PUFs

Most of the currently used PUFs fall into two categories:

- strong PUFs, mainly used for authentication, and
- weak PUFs, primarily used for key storage.

A PUF can be modeled as a black-box challenge-response system: an input challenge c is passed to a PUF which returns a response $r = f(c)$, where $f(\cdot)$ describes the input/output relations of the PUF. The black-box model is appropriate to describe PUFs since input parameters of $f(\cdot)$ are hidden from the user since they represent the interfan manufacturing variability that PUFs use to generate unique challenge-response sets.

The fundamental difference between weak and strong PUFs is the domain of $f(\cdot)$, i.e., the number of unique challenges c that the PUF can process. Weak PUFs can only support a small number of challenges (in some cases just a single challenge). On the contrary, a strong PUF can support a large enough number of challenges such that trying to determine/measure all challenge/response pairs (CRPs) within a limited timeframe is unfeasible.

Both weak and strong PUFs rely on analog physical properties of the fabricated circuit to derive secret information and therefore have noise and variability associated with them. For this reason, modern PUFs designs employ multiple error-correction techniques to mitigate the noise and improve realiability.

Examples of strong PUFs include optical and arbiter PUFs, while ring-oscillator and SRAM PUFs are example of weak ones. [2]

2.2.2 SRAM PUF

SRAM PUFs exploit the positive feedback loop in a SRAM. A SRAM has two stable states (used to store a 1 or a 0), and positive feedback to force the cell into one of these two state and to prevent an accidental state transition.

Figure 2.1 shows a common six-transistor configuration of an SRAM consisting of cross-coupled CMOS inverters (M_1-M_4) and access transistors (M_5-M_6).

Theoretically, when a device with a SRAM is powered on and no write operation is performed, the SRAM cell exists in a metastable state where the feedback pushing the cell toward the "1" state equals the feedback pushing the cell toward the "0" state, thereby keeping the cell indefinitely in this metastable state. However, in actual implementations one feedback loop is always slightly stronger than the other due to small transistor threshold mismatches resulting from process variation. This means that the cell at start up relaxes into either the "1" or "0" state. The final state of the cell depends on the difference between two feeback loops and it is therefore not strongly impacted by temperature or power supply fluctuations. Nonetheless, if the two feedback loops are sufficiently close then random noise or other small environmental fluctuations can result in an output bit flip. Therefore, error correction of this output will be necessary. Error correction can be performed by using repeated measurement: since the relative strengths between the two feedback is relatively static, by measuring the outputs of the cell repeatedly one can assess the stability of a SRAM PUF bit and selectively use the most stable bits as the PUF output. [2]

2.3 SEcube^{TM}

The SEcube^{TM} (Secure Environment cube) Open Security Platform is an open source security-oriented hardware and software platform. It provides hardware and software holistic security focusing on common operational security concepts like groups and policies instead of classical security concepts such as cryptographic algorithms and keys [6]. The SEcube^{TM} is the smallest reconfigurable silicon that combines three main cores in a single-chip design. It embeds a low-power ARM Cortex-M4 processor, a flexible and fast Field-Programmable-Gate-Array (FPGA), and an EAL5+ certified Security Controller (SmartCard), as shown in Figure 2.2. This make the SEcube a secure environment since it is based on a modular software architecture where all functions are isolated [7].

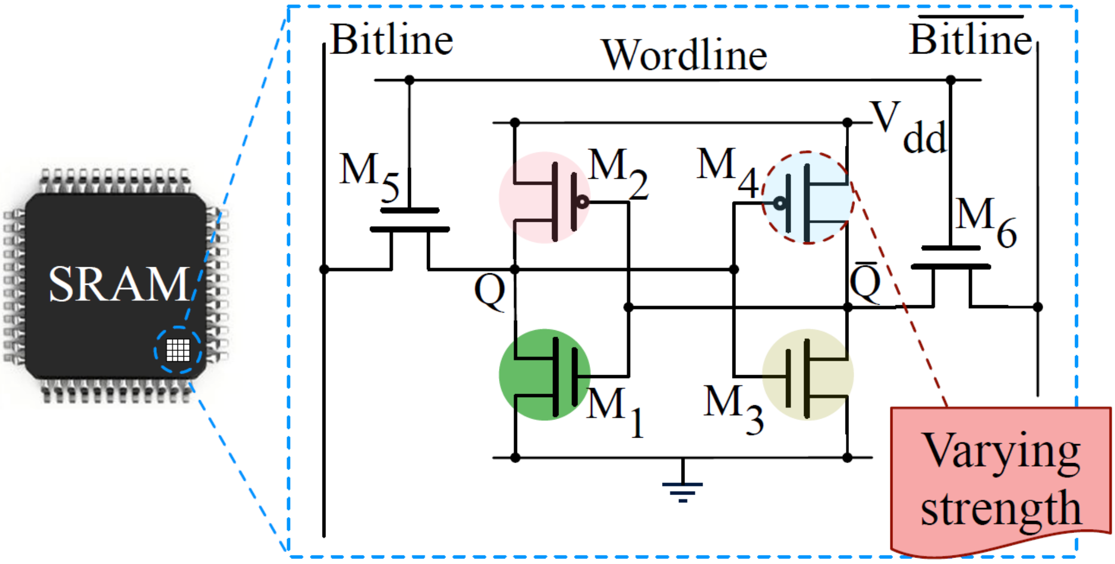

Figure 2.1: SRAM bit cells [3].

Figure 2.2: The three components of the SEcube^{TM}: the ARM Cortex-M4 processor, the FPGA and the EAL5+ SmartCard[6].

CHAPTER 3

Implementation Overview

3.1 Idea behind the project

As already stated, the goal of this project is to provide a secure PUF to check the authenticity of IoT devices in order to avoid impersonation attacks. The proposed solution shows how an SRAM PUF can be implemented. The used SRAM is part of a SECube^{TM} device.

The main idea behind this implementation of an SRAM PUF is that whenever the host tries to connect with the SECube^{TM}, a challenge-response mechanism will take place in order to establish that the SECube^{TM} the host wants to connect to, is the original one and that it has not been replaced. The first time the host connects with the SECube^{TM}, it asks the device to send back a list of strings (which will be called *responses*). These strings are the initial values that the SRAM assumes before being overwritten. At power on, each SRAM cell always tends to have the same stable state (a *0* or a *1*), i.e., each cell always assumes the same state every time it is powered on. The values of this cells cannot be predicted or simulated since they depend on the physical implementation of the SRAM (see 2.2 for more details). The first time the host receives these responses from the device it wants to connect to, it stores them in a file that will be used as a database to be used for future authentication checks.

In later connections with the device, the host sends a challenge to the device it wants to connect to and waits for a response. This challenge is the address location whose content will be checked by the host against the dabase of values it has previously stored during the first connection with the device. When the device receives the challenge, it reads the content of the address and sends a response back to the host. The host then checks if the response it has received matches with the one stored in the database and it can therefore asssess if the device it is trying to connect to is the real one.

3.2 Implementation overview

The implementation of this project can be divided into two flows:

1. The first flow consists in the host retrieving all the responses from the device;
2. The second flow consists in the challenge-response authentication mechanism between the host and the device.

3.2.1 Flow 1

At power on, the first thing the device does is read the values present in the SRAM and store them in a secure place where they can be read later: This procedure has to be carried out every time the device is powered on, since the retrieval of SRAM values is the basis upon which the whole PUF mechanism is based. When the host tries for the first time to connect to the device, it requests the device to send back the list of values it has read from the SRAM. The host will then store these values to implement the challenge-response authentication mechanism the next time it wants to connect to that device. (see Fig. 3.1)

3.2.2 Flow 2

Similarly to *Flow 1*, at power on the device stores the SRAM values in a secure place. When the host wants to connect to the device (and *Flow 1* has already taken place once), it sends the device a challenge, i.e., the index of a response it wants to retrieve. When the host receives the response, it checks that it maches the value at the index indicated in the challenge and that was previously stored. If the two values match, the host can establish the connection with the device, otherwise the connection is interrupted. (see Fig. 3.2);

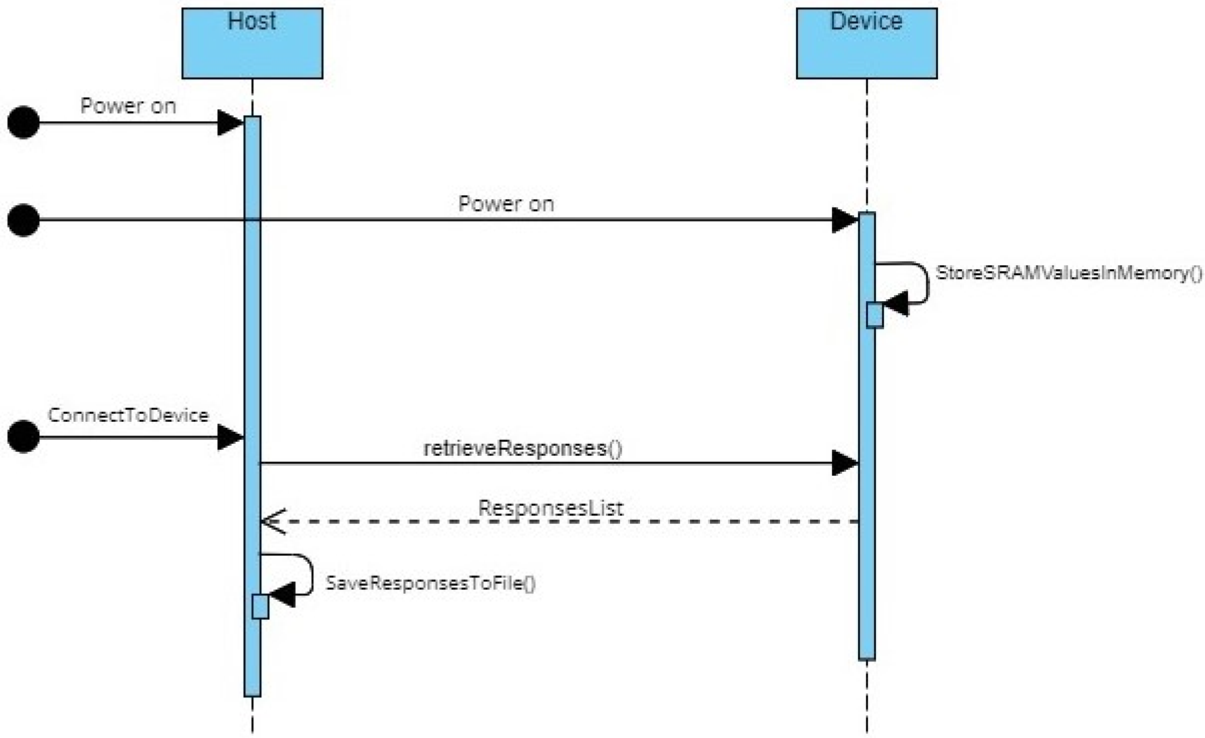

Figure 3.1: Flow 1

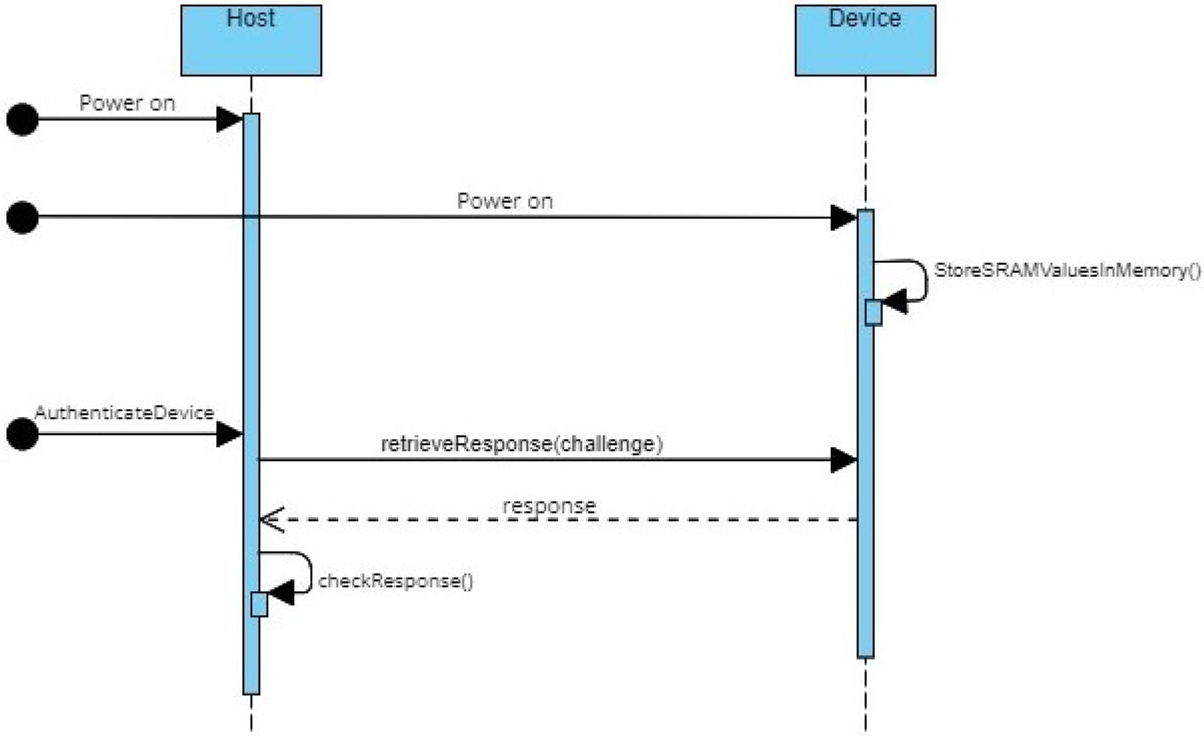

Figure 3.2: Flow 2

CHAPTER 4

Implementation Details

In this chapter a more in depth explanation of the implementation of the project will be provided. The operations for retrieving the responses (which will be also refered to as *PUFs* from here on) and implementing the challenge-response mechanism are done partly on the host side and partly on the board (synonymous of *device*) side.

The board is responsible for the following operations:

1. Read the PUFs from the RAM and store them in the SRAM;
2. Return the list of all PUFs to the host;
3. Return a specific PUF corresponding to the challenge sent by the host.

On the contrary, the host is in charge of the following:

1. Asking the board to retrieve the list of all PUFs and storing them in a file that will be used as database for future reference;
2. Challenge the board to retrieve the PUF and a specific address and then compare it with the corresponding one stored in the database.

4.1 PUF retrieval and DB initialization

At the beginning it is necessary to access the RAM as soon as possible to avoid compromising the content and copy the values found into the SRAM so that they can be read later on. To perform this operation there are some things to be done on both host and board side.

4.1.1 Host side

To get the correct data from the RAM that represent the PUFs it is necessary to access the memory immediately after the board's startup and before writing any data to the RAM. Also to an erase of the memory must be done, so before using copying the PUFs into the SRAM a board initialization is performed.

Once this is done the Host works in three stages. One to prepare the parameters to be send to the board, one to get the response from the board and one to access the DB to store the PUFs. All those stages are implemented in the *examples > puf_db_init.cpp* file.

For the request of the PUFs we don't need to send any parameters. In fact we expect only the list that will be provided from the board. So all we need is a variable where we will store the board response. The actual communication with the board is done through the use of functions implemented in the L1_security API. There we have provided the *L1GetPUFS* function, see figure 4.1, that is responsible of transmitting the parameters to the board and receiving the results from the board.

```
void L1::L1GetPUFS(uint32_t* puf){
    L1GetPufException getPufExc;
    // size of data to be sent. we do not send any data.
    uint16_t dataLen = 0;
    uint16_t respLen = 0;
    try {
        // actual transmission of the data buffer and of the command ID to
        // identify the function to be executed on board side
        TXRXData(L1Commands::Codes::GETPUFS, dataLen, 0, &respLen);
    }
    catch(L1Exception& e) {
        throw getPufExc;
    }

    // check on the size of the data received
    // expected a 1000 32 bit values (4*1000 bytes)
    if(respLen != 4*1000){
        printf("[L1 error] not all pufs have been received\n");
    } else {
        // Read the received buffer.
        // Store the response in the "response" variable indicating how many bytes we expect (1 word = 4 bytes)
        this->base.ReadSessionBuffer((uint8_t*)puf, L1Response::Offset::DATA, 4*1000);
    }
}
```

Figure 4.1: Function responsible for the communicating with the board and receiving the list of PUFs.

In that piece of code we can see that we are expecting 1000 PUFs. In that operation we must provide the size of the data we are transmitting/receiving and also the ID of the command to be executed on the board side. The response then will be given through a buffer from which we can indicate where to store the result and the amount of bytes the result is composed of. Then the host is responsible of writing that data to the DB.

4.1.2 Board side

In this stage the board is responsible of retrieving the initial data, PUFs, from the RAM and store them to the SRAM so that they can be accessed safely later.

Since the content of the RAM will be overwritten, the PUFs must be copied as soon as possible. For this reason this operation has to be done before any memory initialization. So this operation is done in the startup assembly code in which the reset handler can be found. This way we guarantee that we read the RAM before any memory initialization that could result in compromising the PUFs.

Since the flash needs control registers to be set in order to be accessed in a safe way we relied on the HAL functions provided by STMicroelectronics. More precisely the functions to unlock and program/write in memory. The code written to perform this operation can be seen in figure 4.2

Using the reference manual for the MCU (STM32f429) we know the memory mapping of both the SRAM(0x020000000 - 0x02002FFFF) and flash memory(0x08000000 - 0x081FFFFF). Starting from that we scanned the SRAM and loaded the content to a part of the flash that will not overwrite sensible data.

At the end of the execution of the assembly code we expect 1000 PUFs to be stored in the flash memory which will be accessible once we enter the main and eventually the execution loop. At that point we can call the implemented functions and access the content of the SRAM.

Once we have the PUFs in memory we need access to them. This is done in the *puf_retreive*() function, found in the *se3_dispatcher_core.c* file. This is the function associated to the command code transmitted from the Host side to the board.

To make the command call possible it is necessary to define these commands in the *se3_dispatcher_core.h* which must reflect the ID associated to the same command also on the host side.

```
store_puf:
        push {lr}
        Bl HAL_FLASH_Unlock
        ldr r7,=start_flash
        ldr r5,=start_ram
        ldr r6,=end_ram
        eor r2,r2,r2

loop1:  ldr r2,[r5]
        add r5, #4
        mov r0, #2
        add r1, r7, #0
        BL HAL_FLASH_Program
        add r7,#4
        cmp r5,r6
        bls loop1
        pop {lr}
        bx lr
```

Figure 4.2: Assembly code to store PUFs into the flash memory.

```
uint32_t puf_retreive(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)
{
    uint32_t puf_num = 1000;
    uint32_t flashAddress = 0x080E0000;

    // Store flash content into the variable that will be returned to the Host
    for(uint32_t i=0; i<4*puf_num; i++)
    {
        *((uint8_t *)resp + i) = *(uint8_t *)flashAddress;
        flashAddress++;
        // Indication of how many BYTES have been transfered. It will be passed to the host
        *resp_size+=1;
    }

    return SE3_OK;
}
```

Figure 4.3: puf_retreive function.

In the *puf_retreive*() function we simply read from the SRAM the PUFS that we stored in the assembly code. This data is the data that will be sent to the host side.

4.2 Application of a challenge and verification of the device

Once we have the DB filled with the PUFs we can use that to check the authenticity of the board by comparing the PUFs in the DB to the ones regenerated by the board. Again to realize the functionality we will have to do some things on the host and some on the board side.

4.2.1 Host side

The approach is similar to the one used for reading the PUFs but now we have different parameters to transmit. In fact we need to provide some data, the challenge, for the board to work on. So the Host is responsible of getting the challenge, use it to access the DB and retrieve the expected response, PUF. Then the challenge is sent to the board expecting in return the actual response to that challenge.

Then the next stage is to perform the actual transmit/receive which is done using in the L1ChallengePUF function of the L1 API, shown in figure 4.4.

In this function we have some data to transmit and the TXRX works with bytes we have to provide the size of the data to be sent/received in bytes which is 4 since we transmit just 32bit of the challenge. Then as response we expect just one PUF so another 4 bytes.

```
void L1::L1ChallengePUF(uint32_t challenge, uint32_t* response){
    L1ChallengePufException challengePufExc;
    // size of data to be sent. size of the challenge + expected PUF
    uint16_t dataLen = 4;
    uint16_t respLen = 0;

    // filling the buffer with the data to be sent specifying also the offset and the data length
    this->base.FillSessionBuffer((uint8_t*)&challenge, L1Response::Offset::DATA, dataLen);
    try {
        // actual transmission of the data buffer and of the command ID to
        // identify the function to be executed on board side
        TXRXData(L1Commands::Codes::CHALLENGEPUF, dataLen, 0, &respLen);
    }
    catch(L1Exception& e) {
        throw challengePufExc;
    }

    // check on the size of the data received
    // expected a 32 bit value (4 bytes)
    if(respLen != dataLen){
        printf("[error] no result received\n");
    } else {
        // Read the received buffer
        // Store the response in the "response" variable indicating how many bytes we expect (1 word = 4 bytes)
        this->base.ReadSessionBuffer((uint8_t*)response, L1Response::Offset::DATA, dataLen);
    }

}
```

Figure 4.4: L1ChallengePUF function for transmitting challenge and response PUF

Once the Host has both expected and actual response it compares them using an acceptable Hamming distance. The one chosen is a Hamming distance of 4 based on statistical results. These operations are done in the *examples > puf_challenge.cpp* file shown in figure 4.5

```
printf("\n\n==========\n");
printf("Apply PUF challenge ...\n\n");

uint32_t board_puf;
uint32_t host_puf = 0;
// the challenge which can be provided in other ways. In this case it is hardcoded
uint32_t challenge = 0x080E0004;
uint32_t DB_addr;

// using the challenge as an address to access the DB(file) entry
DB_addr = (challenge - MEMBASE);
if(DB_addr%4 !=0 ){
    printf("[Host ERROR]: challenge address 0x%X is not word aligned", challenge);
    return 1;
}
host_puf = readPUF(DB_addr/4);

// print of the parameters to be passed to the board, just for debugging
printf("[Host] Challenge applied: 0x%X \n", challenge);

// calling function of L1 API responsible for the communication between the host and the board
l1->L1ChallengePUF(challenge, &board_puf);

// print of the puf received from the DB and the one received from the board
printf("[Host] Board response received successfully\n");
printf("[Host] DB_puf:      x%X\n", host_puf);
printf("[Host] Board_Puf:   x%X\n", board_puf);

printf("\n----------\n");
// check on the pufs using hamming distance of 4
if(hammingDistance(host_puf, board_puf)<5)
    printf("[Host] pufs DO match!!!\n");
else
    printf("\n[Host] pufs DON'T match!!!\n");
printf("----------\n");

return 0;
```

Figure 4.5: puf_challenge.cpp

4.2.2 Board side

The Board at this point has been provided with the challenge. This data has been sent using Little endian encoding so before being able to use the parameter passed a reconstruction is needed.

So we extract the compacted data received and obtain the challenge. From there we can access the SRAM using the challenge as an address. This will be the actual PUF returned to the Host.

```
uint32_t puf_challenge(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)
{
    // variable used to store the reconstructed data coming from the host
    uint32_t challenge;

    // Little endian: req=04000E08
    // Reconstruction of received data considering endianness
    challenge = (uint32_t)req[0] | ((uint32_t)req[1] << 8) | ((uint32_t)req[2] << 16) | ((uint32_t)req[3] << 24);
    // Read puf from flash
    for(uint8_t i=0; i<4; i++)
    {
        *((uint8_t *)resp + i) = *(uint8_t *)challenge;
        challenge++;
        *resp_size+=1;
    }

    return SE3_OK;
}
```

Figure 4.6: puf_challenge_board function on the board side

CHAPTER 5

Results

In this chapter we expect you to list and explain all the results that you have achieved. Pictures can be useful to explain the results. Think about this chapter as something similar to the demo of the oral presentation. You can also include pictures about use-cases (you can also decide to add use cases to the high level overview chapter).

5.1 Known Issues

One many issue of this kind of approach could be that there is not the possibility to avoid a Man-in-the-Middle, in order to avoid that a man can steal information from this kind of information it is necessary to encrypt the communication. It is important to say that the type of encryption and the necessity to encrypt or not depend from the type of device and by the level of sensibility of the datas.

5.2 Future Work

Many are the implementations that can be done on this project. The main one is to evaluate and store in a secure place the hash value of the file containing the challenge-response. This kind of implementation can be used in order to ensure the integrity of the challenge-response of a particular device. The idea consists in evaluating the hash value of the file before taking information from it and comparing it with the digest that we store in another place. If the value is the same it means that the file is not corrupted.

Once a challenge-response is used, it has to be eliminated and not used anymore in the future; in this way it is possible to avoid replicant attacks. —- TO BE EXPLAINED BETTER ————————-

CHAPTER 6

Conclusions

To conclude, with this project it has been shown how a sram PUF works and how to implement it. In particular at the beginning It has been presented the problem present in these years, and why this kind of solution is important.

Subsequently has been shown some different kinds of PUF and how to approach the implementation of one of them....

It has been shown how to increase the security of the implementation with some simple precautions as a hash function that can help to increase integrity... .....

In the end it was shown some results of this approach....

To continue————————————-

Bibliography

[1] IoT.Business.News, *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally*, 2022, `https://iotbusinessnews.com/2022/05/19/70343-state-of-iot-2022-number-of-connected-iot-devices-growing-18-to-14-4-billion-globally/` [Online; Accessed 2022, 29 July].

[2] C. Herder, M. Yu, F. Koushanfar and S. Devadas, *Physical Unclonable Functions and Applications: A Tutorial*, in *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126-1141, Aug. 2014, doi: 10.1109/JPROC.2014.2320516.

[3] S. Sutar, A. Raha, and V. Raghunathan, *Memory-based Combination PUFs for Device Authentication in Embedded Systems*, for the School of Electrical and Computer Engineering, Purdue University, 5 December 2017, arXiv:1712.01611v1

[4] A. Varriale, E. I. Vatajelu, G. Di Natale, P. Prinetto, P. Trotta and T. Margaria, *SEcube™: An open-source security platform in a single SoC*, 2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2016, pp. 1-6, doi: 10.1109/DTIS.2016.7483810.

[5] G. E. Suh and S. Devadas, *Physical Unclonable Functions for Device Authentication and Secret Key Generation*, 2007 44th ACM/IEEE Design Automation Conference, 2007, pp. 9-14.

[6] M. Fornero, N. Maunero, P. Printetto, G. Roascio, A. Varriale, *SEcube^{TM} Open Security Platform, Introduction.* Released October 2021. `https://github.com/SEcube-Project/SEcube-SDK`.

[7] *What is SEcube^{TM}*, `https://www.secube.blu5group.com/`. [Online; Accessed 2022, 05 August].

[8] Donald E. Knuth (1986) *The TEX Book*, Addison-Wesley Professional.

[9] Leslie Lamport (1994) *LATEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.

APPENDIX A

User Manual

A.1 SEcube™ Software Development Kit (version 1.5.2)

A.2 Licence

All SEcube releases published on this website are Open Source - GPL 3.0 and are developed by the Academia Community.

A.3 Terms of use

By downloading the software from this page, you agree to the specified terms.

The software is provided to you "as is" and we make no express or implied warranties whatsoever with respect to its functionality, operability, or use, including, without limitation, any implied warranties of merchantability, fitness for a particular purpose, or infringement. We expressly disclaim any liability whatsoever for any direct, indirect, consequential, incidental or special damages, including, without limitation, loss revenues, lost profits, losses resulting from business interruption or loss of data, regardless of the form of action or legal thereunder which the liability may be asserted, even if advised of the possibility likelihood of such damages.

A.4 PUF

This is a project made for the course of Cybersecurity for embedded systems of the Masters on embedded system in Politecnico di Torino.

The purpose of this project is to extract the RAM PUFs from the memory of the SEcube storing them on a DB, in this case a simple txt file, and perform also a challenge on the board.

A.5 Instructions to run the project

A.5.1 Import the project

Instructions on how to import the project are the same as the ones for the original project provided in the wiki

A.5.2 Run the project

the steps to run the project are the following

- Erase the flash memory of the SEcube

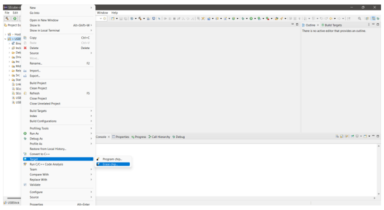

Figure A.1: Erase flash

- Flash the SEcube

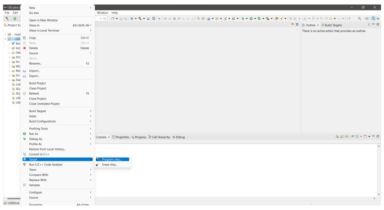

Figure A.2: Chip flash

- Run on the host the "puf_db_init.cpp"

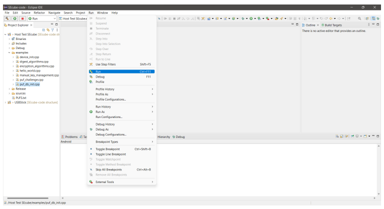

Figure A.3: Puf db init

- Run on the host the "puf_challenge.cpp"

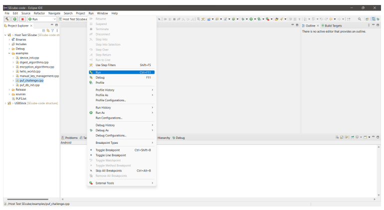

Figure A.4: Puf challenge

APPENDIX B

API

If you developed some source code that is supposed to be used by other software in order to perform some action, it is very likely that you have implemented an API. Use this appendix to describe each function of the API (prototype, parameters, returned values, purpose of the function, etc).