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
Dr. Nicolò Maunero

Contents

List of Figures

Abstract

Recently, the amount of small electronic devices embedded in larger systems has grown exponentially. These devices play an essential role in almost all domains in everyday life, such as health care, houses and cities automation, wearable devices, autonomous driving, industrial and farming applications. Unfortunately, this enormous growth goes hand in hand with the growth of counterfeit devices, thus giving rise to privacy and security concerns.

To counteract this problem, an interesting solution is the use of PUFs (Physical Unclonable Functions). PUFs use instance-specific features of a device to provide a unique way to identify that device and therefore allowing a secure authentication method.

The goal of this project is to develop a SRAM based PUF for the SEcube^{TM}, a single-chip integratable device, to provide a secure authentication method between a host and the device.

The idea behind this authentication procedure is that whenever the host machine tries to connect to a device, a challenge-response mechanism takes place. The first time this connection is established, the device sends the host a list of values that the host will store for future reference. In later connections, the host asks the device to provide a specific response and, if the device answers correctly, the connection is approved, otherwise it is interrupted.

The SRAM PUF developed in this project provides a way to securely authenticate a device when a connection to it must be established.

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 billion [1]. The ubiquitousness of such devices coupled with their ability to access potentially sensitive and 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 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^{TM}, 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 against 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 characteristics 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 invasive 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 objects 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 innate 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 reliability.

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 feedback 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™

The SEcube™ (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™ 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].

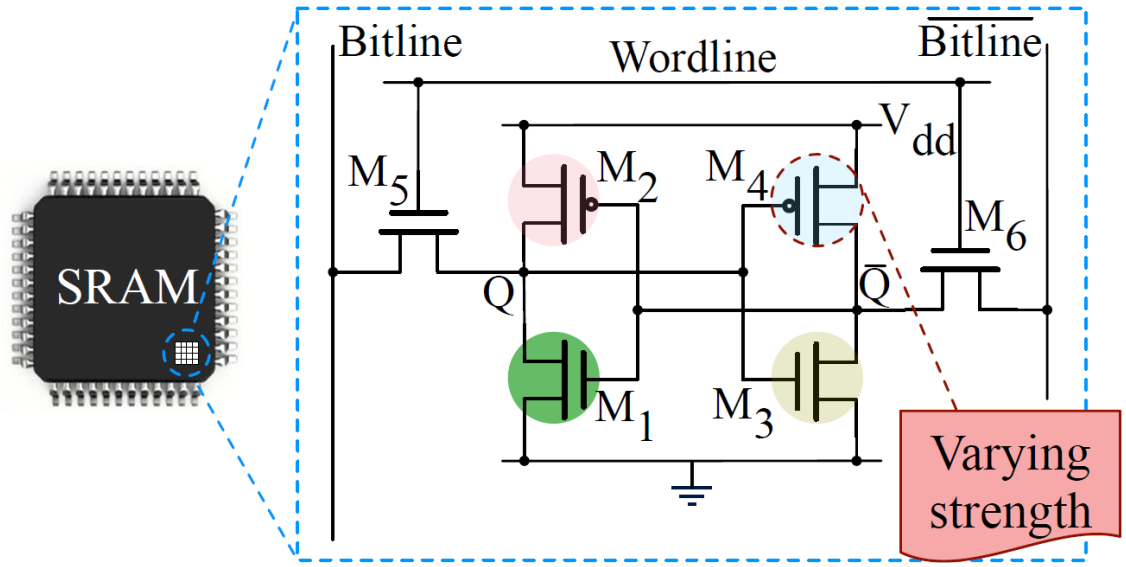

Figure 2.1: SRAM bit cells [3].

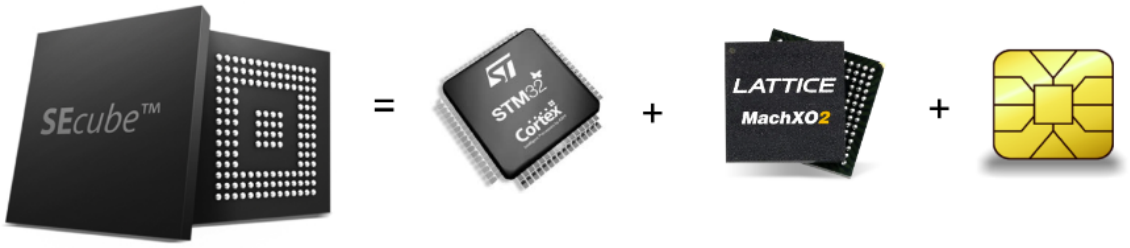

Figure 2.2: The three components of the SEcube™: the ARM Cortex-M4 processor, the FPGA and the EAL5+ SmartCard[6].

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 database 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 assess 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 mat ches 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);

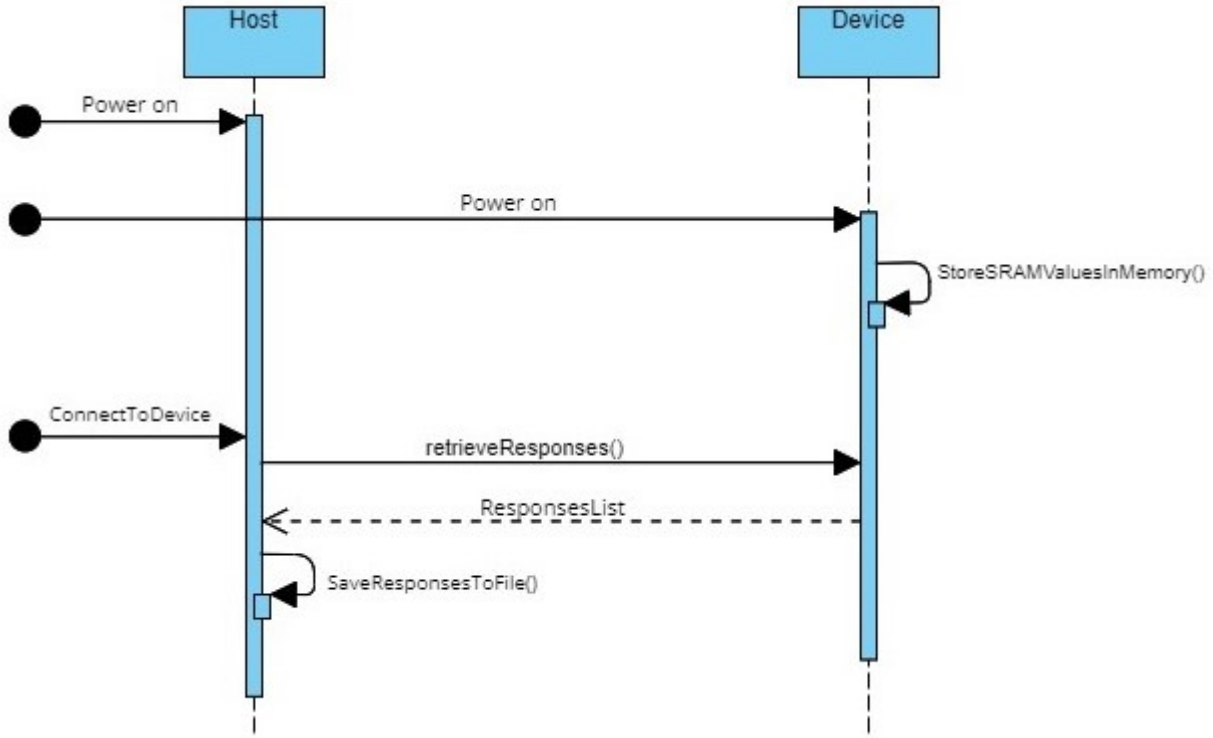

Figure 3.1: Flow 1

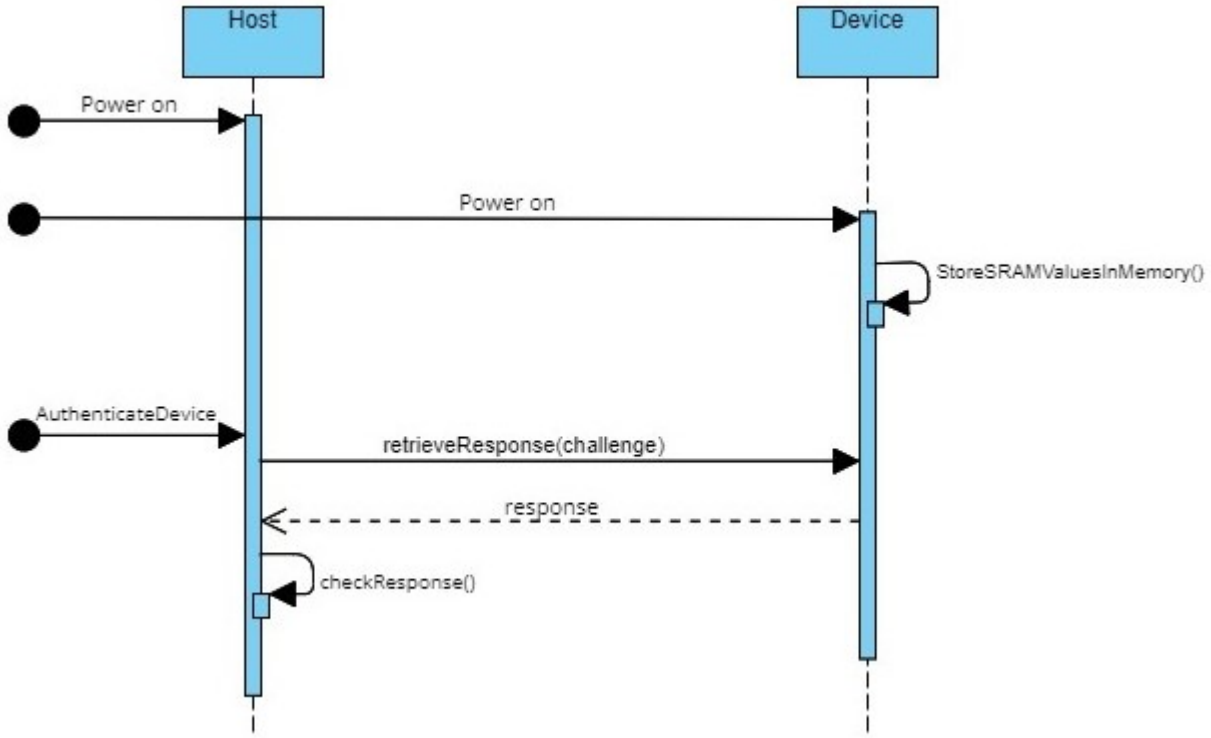

Figure 3.2: Flow 2

CHAPTER 4

Implementation Details

This chapter provides a more in-depth explanation of how the different functionalities have been implemented. The main operations that have been added to the L1 API for managing the PUFs are the following:

- Fill the host DB with a number of responses received from the device
- Challenge the device and determine the authenticity of such device

To perform these actions correctly, it is necessary to establish a communication between the host and the device (also referred to as *board*).

The board is responsible for

1. Reading the PUF responses from the RAM and store them in the flash;
2. Providing the list of all PUF responses to the host;
3. Returning a specific response corresponding to the challenge sent by the host.

On the other hand, the host is in charge of

1. Retrieving the list of PUF responses and storing them in a database;
2. Sending a challenge to the board, retrieving the corresponding response and then comparing it with the one stored in the database.

4.1 PUF retrieval and DB initialization

Before the PUFs can be retrieved, it is necessary to perform a memory erase: this guarantees that the whole PUFs retrieval process starts from a clean state and that the RAM cells assume their stable states at start up.

Then, when the board is powered on, it is necessary to access the SRAM as soon as possible to avoid compromising its content and to copy the values found in the SRAM to the flash memory so that they can be read later on. To perform this operation, tasks on both the host and board side have to be carried out.

4.1.1 Host side

To retrieve the correct data from the RAM (whose values are the PUFs needed to implement the whole authentication mechanism), it is necessary to access the memory immediately after the board's startup and before writing any data in the RAM.

Once this is done, the host works in three steps. The first one is to prepare the parameters to be send to the board, the second one is to get the response from the board and the last one is to access the DB to store them. All these steps are implemented in the *examples > puf_db_init.cpp* file.

In order to request the PUFs, no parameters need to be sent. In fact, only the list of PUFs provided from the board is expected. So, only a variable to store the board response is passed. The actual communication with the board is done through the use of functions implemented in the L1_security API. There we added the *L1GetPUFS* function (see Figure 4.1), which is responsible for transmitting the parameters to the board and receiving the results from it.

```
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 communicating with the board and receiving the list of PUFs.

As shown in the code in Figure 4.1, 1000 PUFs are expected. The communication between host and board is done through the use of buffers that will then be transmitted using an appropriate transmit/receive function. In the buffer, the data to be sent and the size of the data being transmitted or received need to be set. This buffer is then sent together with the command ID of the operation that must be performed on the board side. The response will be transmitted again through a buffer that will store a number bytes of the response, numeber that has to be provided. After that, the host is responsible for managing that data, which in this case is to write them in a DB.

4.1.2 Board side

In this stage the board is responsible for retrieving the initial data, the PUFs, from the SRAM and store them in the flash memory so that they can be safely accessed 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. This operation is therefore done in the startup assembly file in which the reset handler can be found. In this way, it is guaranteed that the RAM is read before spoiling its content with other code which would mean 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 found in Figure 4.2

```
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.

As indicated in the reference manual for the MCU STM32F429[8], the memory mapping of the SRAM is 0x020000000 - 0x02002FFFF, and 0x08000000 - 0x081FFFFF for the flash memory. The SRAM is therefore scanned in that range of addresses and the retrieved content is then loaded in a section of the flash memory that is not used by the board for its normal functioning.

At the end of the execution of the assembly code, the 1000 expected PUFs are stored in the flash memory which will be accessible once entered in the *main.cpp* and in the execution loop.

Once the PUFs are in memory, it is necessary to access them. This is done in the *puf_retreive*() function that can be 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 found on the host side.

```
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 read from the flash the PUFs that we stored at startup. This 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 PUFs, it can be used to check the authenticity of the board by comparing the PUFs in the DB to the ones provided by the board. Again, to implement this functionality, some tasks on both host and board side must be performed.

4.2.1 Host side

The approach is similar to the one used for reading the PUFs, but in this case different parameters must be transmitted. In fact, the challenge needs to be sent to the board. The host is therefore responsible of generating the challenge, using it to access the DB. Then the challenge is also sent to the board and the response to it is expected in return.

The next step is to perform the actual transmit/receive, which is done using the L1ChallengePUF function of the L1 API (see figure 4.4).

```
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

In this function, some data must be transmitted. Since the TXRX works with bytes, it is necessary to provide the size of the data to be sent/received also in bytes. This size is equal to 4, since the transmitted challenge is 32bit long. As a response, 32 bits are also expected, hence again 4 bytes.

Once the host has received the actual response, it compares it with the the expected one found in the DB, that we have previously fetched. This comparison is done using a hamming distance. The chosen distance is of 4 bits. This choice is based on statistical results that have been obtained, trading off accuracy and safety. These operations are done in the *examples* > *puf_challenge.cpp* file shown in Figure 4.5

4.2.2 Board side

The board at this point has been provided with the challenge. However, the data has been sent using little-endian encoding: so, before being able to use the parameter passed, it is necessary to reconstruct the received data to obtain the correct challenge. At this point, the flash memory can be accessed using the challenge as an address. The value at that address will be the actual PUF response to be returned to the host.

The board has now completed its tasks and it is waiting for the next instruction. The host, on the other side, will have to manage the response provided and complete the authentication process.

```
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

```
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

The aim of the project has been successfully reached and the implemented PUF can check with a good accuracy that the device is the correct one. Unfortunately, achieving 100% accuracy in recognizing the device is unfeasible. This is due to the fact that sometimes a few bits can change their standard value when the device is switched on. For this reason, a non-null hamming distance can be present between the response received and the expected one.

Figure 5.1 shows a graphical representation of the results obtained executing 1000 PUF authentications on the same device with 10 different challenge-response pairs. It is possible to observe that by increasing the value of the thresholds for the hamming distance the number of responses recognized as correct increases. Obviously, it would be incorrect to increase the value of the threshold too much because it would increase the risk of mistaking a device for another one, thus allowing attackers to counterfeit the original device. In this project, it was decided that a threshold equal to 4 is a good compromise between security and efficiency of the implementation.

5.1 Known Issues

One of the issues of this implementation is that it is not secure from the man-in-the-middle attack, since an attacker could easily steal the challenges and responses of the PUF. There are some optimizations that can be done in order to avoid this kind of attack. The main one would be to eliminate a challenge-response pair once it has been used: by doing this, even if an attacker manages to steal that pair, it will not be able to use it for a later authentication and replicant attacks can be therefore avoided.

Another type of security improvement is the encryption of the data in order to ensure confidentiality in the communication. The encryption should be used, in particular, during the first communication between device and host, i.e., when the device sends all the challenge-response pairs to the host. It is also important to state that the type of encryption and the necessity to encrypt or not depend on the type of device and by the level of sensibility of the data that it can manage.

One more improvement could be to perform the responses match check on the board side. By doing this, the possibility of any manipulation of the response provided by the board before making the match check could be eliminated.

5.2 Future Work

There are some implementations that could be done to improve this project, starting from the ones explained in the previous paragraph.

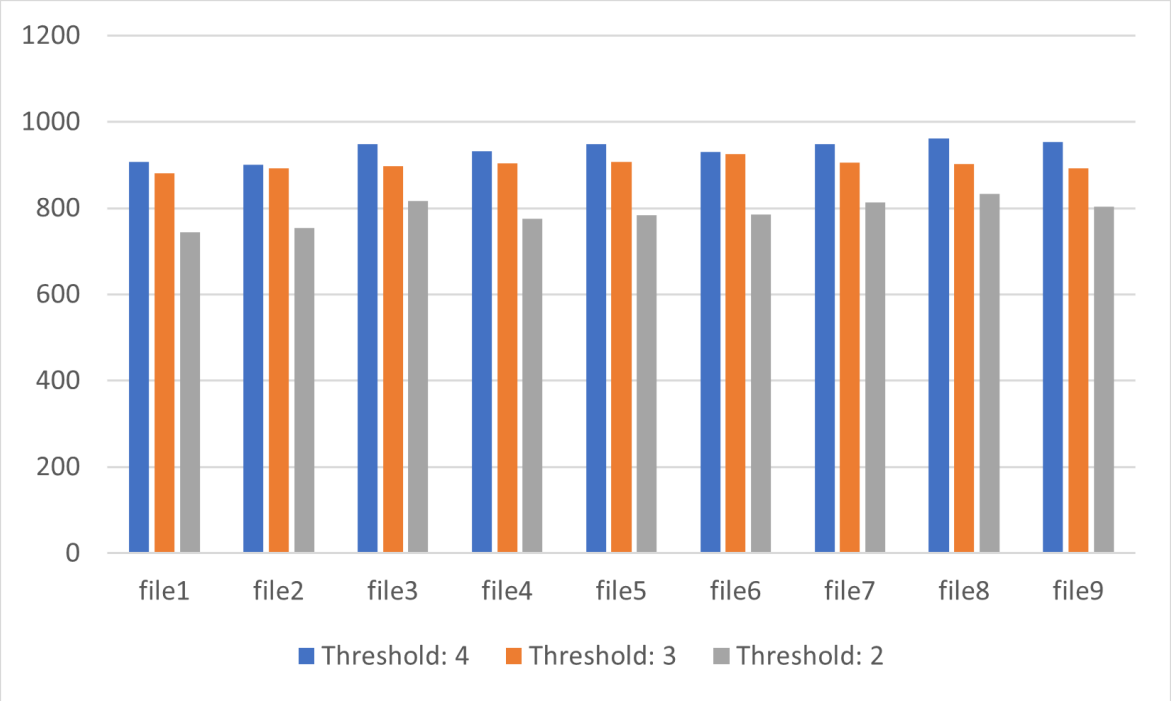

Figure 5.1: Results of PUF authentications with different hamming distance thresholds.

The main one could be to store the challenge-response pairs in the host side encrypting them. In this way, even if the file were stolen by an attacker, he could not be able to use those pairs.

Another improvement could be evaluating and storing in a secure place the hash value of the file containing the challenge-response pairs. This kind of implementation can be used to ensure the integrity of the challenge-responses 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 had been previously stored in another place. If the values are the same, then the file is not corrupted.

CHAPTER 6

Conclusions

In conclusion, this project offers a possible implementation of a physical unclonable function (PUF) in order to test the authenticity of a device. In particular, at the beginning of the report it is explained what a PUF is and why it is important to identify the authenticity of IoT devices.

After that, a specific type of PUF, the SRAM PUF, was presented: its functionality and its properties were listed.

In the last part, it was explained in detail the implementation done in this project, giving some details on which operations have to be executed on the host side and on the bord side.

In the end, the results obtained with this approach were analyzed. Moreover, some possible implementations and improvements to increase the security of the PUF were presented.

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] *STM32F427xx STM32F429xx*, https://www.st.com/en/microcontrollers-microprocessors/stm32f429-439.html#overview.

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 Master on Embedded Systems at the 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™

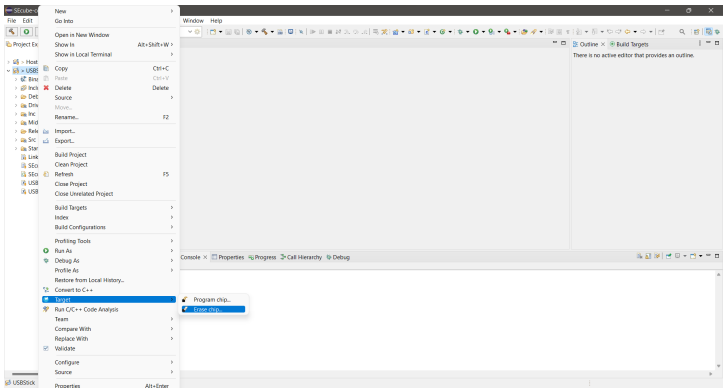

Figure A.1: Erase flash

- Flash the SEcube

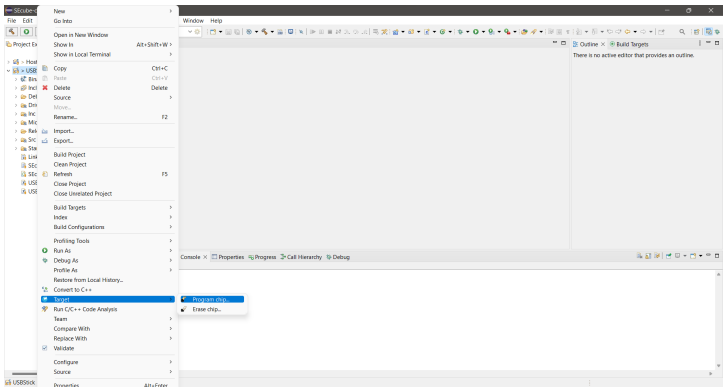

Figure A.2: Chip flash

- Run on the host the "puf_db_init.cpp"

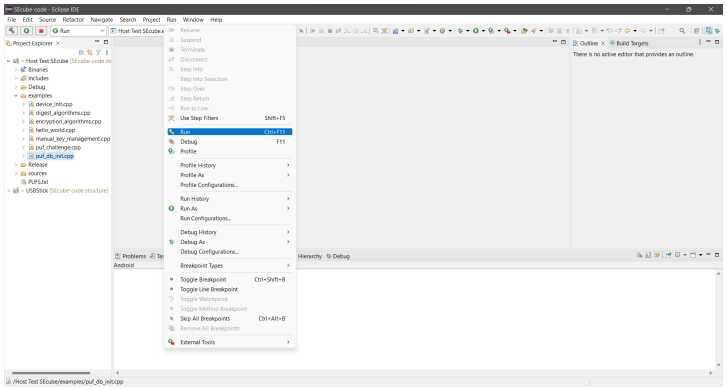

Figure A.3: Puf DB init

- Run on the host the "puf_challenge.cpp"

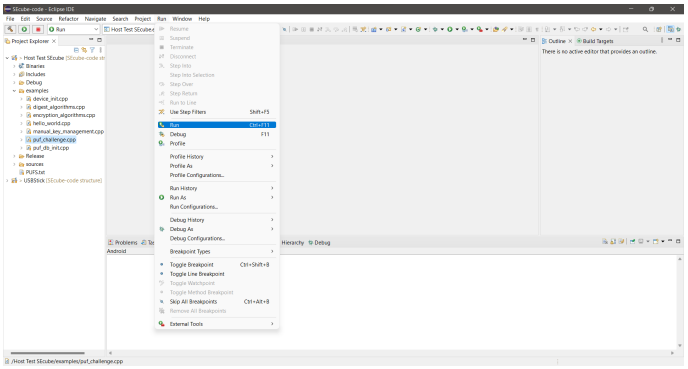

Figure A.4: PUF challenge

APPENDIX B

API

In the following lines it is explained some API functions which might be used by other software that want to implement a PUF authentication:

- void L1::L1GetPUFS(uint32_t* puf);
  1 parameter: pointer to an empty buffer that will be fed with the values of the PUF
  purpose: Function responsible for communicating with the board and receiving the list of PUFs
- void L1::L1ChallengePUF(uint32_t challenge, uint32_t* response)
  1 parameter: challenge to send to the board
  2 parameter: pointer to a buffer where the response will be stored
  purpose: function for transmitting challenge and response PUF