Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Physically Unclonable Functions (PUFs) and CAMPUF: Utilizing CMOS Technology for Hardware-Based Security

Project Report

Candidates:
Lorenzo Ruotolo (s313207)
Marco Chiarle (s314730)
Alessandro Vargiu (s314294)
Edoardo Manfredi (s316714)

Referee:
Prof. Paolo Prinetto

Contents

List of Figures

List of Tables

Abstract

Physically Unclonable Functions (PUFs) have gained significant attention in the hardware security domain, due to their ability to generate unique identifiers from physical variations in hardware components, which helps in the authentication and key generation domains. This report provides a comprehensive description of a particular PUF design, called CamPUF, which is based on CMOS image sensors.

Inherent physical processes happening in image sensors are exploited, such as Fixed Pattern Noise and, in particular, one of its sources: the Dark Signal Non-Uniformity (DSNU). DSNU can be extracted by dark digital images, which means that a fingerprint based on this noise is harder to extract from illuminated images, where PRNU and other type of noises are prevalent. This design also includes an efficient and reliable key generation algorithm and a computationally light device authentication mechanism.

A complete implementation of the CamPUF is provided, along with a high-level view of the solution and an explanation of the theory which is at the basis of this design. This document will show a comprehensive description, a complete testing procedure, along with the steps to replicate it, and an overview of the results, which will highlight the strengths and the reliability of this PUF design. At last, we also present an *attack mitigation* section, to show the security against possible counterfeiting measures.

CHAPTER 1

Introduction

In this chapter, it is presented an overview of the project, its objectives, and the scope of topics covered. The project revolves around the exploration and implementation of Complementary Metal-Oxide-Semiconductor Arbiter Physically Unclonable Functions (CAMPUFs).

1.1 Project Overview

CAMPUFs are a class of hardware-based security primitives that leverage the inherent physical variations within CMOS integrated circuits. These variations arise during the semiconductor manufacturing process, resulting in unique fingerprints for each individual chip. The primary objective of this project is to study, analyze, and implement CAMPUFs to enhance hardware security and provide tamper-resistant cryptographic key generation.

1.2 Objectives

The project aims to achieve the following objectives:

- **State of Art**: understanding CAMPUFs regarding its theoretical foundation, mainly about properties, advantages and methods.
- **Desing and Implementation**: analyzing different CAMPUFs architectures and choosing one of the different methods.
- **Demonstration**: testing with practical applications that the implementation works with a certain degree of accuracy.

1.3 Document Organization

This document is organized in the following way.

- **Chapter 1**: Introduction. This chapter.
- **Chapter 2**: Background. In this chapter, all the theoretical concepts are discussed, starting with a summary about PUFs. Then, it is presented the principle behind the fingerprint extraction, that is the fixed-pattern noise, and, finally, it is explained how the fingerprint is exploited by way of the challenge-response pair.

- **Chapter 3**: Implementation overview. It is explained the researches about PUFs, and the choosing of camPUF as demonstrative project. Then, it is shown how the project has been developed and its high-level architecture.
- **Chapter 4**: Implementation details. This chapter contains details useful to understand the code base, how to test and modify effectively the code and some programming choices.
- **Chapter 5**: Results. Here there is a discussion about the dataset chosen, the results obtained, and the limits and future improvement of this project.
- **Chapter 6**: Conclusions.

Moreover, it is provided in the Appendix A an user manual containing all the minimal information needed to replicate the project rapidly.

CHAPTER 2

Background

In this era, as technology has become an integral part of modern life, digital security has become crucial. Hardware security is becoming even more significant, along with the need for development and research of robust hardware security measures.

As technology advances, so do cyber threats. The rise of IoT and interconnected devices is also a crucial factor. In this environment, Physically Unclonable Functions (PUFs) have emerged as a promising approach to address the challenges of hardware authentication, secure key generation, and anti-counterfeiting measures. Among the various PUF implementations, Complementary Metal-Oxide-Semiconductor Arbiter PUFs (CAMPUFs) stand out as a powerful hardware security primitive based on CMOS technology.

CAMPUFs leverage the unique physical variations inherent in CMOS integrated circuits to generate unpredictable and practically unclonable responses. The foundation of CMOS technology, with its low power consumption and high integration capabilities, makes it an ideal platform for building complex digital circuits and implementing PUFs.

This chapter will provide an overview of the concepts that are the basis for the CamPUF design and that are needed in order to understand it.

2.1 Physically Unclonable Functions

A Physically Unclonable Function (PUF) is a security metric in hardware systems based on inherent device physical variations to produce an unclonable, unique device response to a given input. Unclonability means that each PUF has a unpredictable response to a given input, due to the response being created by complex interactions between many random components. The response acts as a unique device identifier, a sort of "digital fingerprint" for the device.

2.1.1 Purposes

- **Key Generation**: PUFs can generate cryptographic keys directly from the unique physical responses of the ICs. These keys can be used for secure communication, encryption, and data protection.
- **Challenge-Response Mechanism**: PUFs operate on a challenge-response mechanism, where a unique challenge is provided to the device, and the corresponding response is generated based on the unique physical characteristics of that particular IC.
- **Randomness and Entropy Generation**: PUFs can serve as a high-quality source of randomness and entropy, critical for cryptographic protocols and secure communication.

- **Authentication**: PUF responses can be used for secure device authentication, verifying the identity and integrity of hardware components in various applications, such as secure booting and secure firmware updates.
- **Uniqueness and Unpredictability**: PUFs generate device-specific responses based on the inherent physical variations during manufacturing. As a result, each instance of the IC exhibits a unique response, making it practically impossible to clone or replicate the device.
- **Anti-Counterfeiting Measures**: PUFs play a crucial role in preventing counterfeiting and unauthorized duplication of hardware components, as the uniqueness of the responses ensures the authenticity of genuine devices.

2.1.2 Security Properties and Applications of PUFs

Key Security Properties of PUFs

- **Unpredictability**: The inherent randomness linked to PUF responses, even when given the same challenge multiple times. This property makes stonger the security of PUF-based authentication and key generation.
- **Uniqueness and Unclonability**: PUFs depend on the uniqueness of their physical microstructure. This microstructure depends on random physical factors introduced during manufacturing that gives unique responses. These factors are unpredictable and uncontrollable, which makes it virtually impossible to duplicate or clone the structure.
- **Resistance to Physical Attacks**: PUFs are resilient against various physical attacks, including invasive attacks like reverse engineering and probing, as well as non-invasive attacks like side-channel analysis.

Practical Applications of PUFs

- **Secure Key Generation**: PUF responses can be utilized to generate cryptographic keys without the need for additional storage of secret keys, enhancing the security of cryptographic protocols.This applciation is divided in Secret Key Distribution and Secret Key Storage.
- **Hardware Authentication**: PUFs can be employed for secure device authentication, ensuring the legitimacy of hardware components and protecting against unauthorized access.
- **Anti-Counterfeiting Measures**: PUFs serve as a powerful tool for detecting counterfeit devices, as the unique responses enable the verification of genuine products.
- **Secure Communication**: PUF-based keys are valuable for securing communication channels, enabling secure and authenticated data transmission.

2.1.3 Categories of Implementation

PUFs can be categorized based on different operational principles and sources of randomness

Intrinsic PUFs

Intrinsic PUFs are characterized by their reliance on the inherent physical variations present within a single chip during the semiconductor manufacturing process. These variations arise due to manufacturing imperfections, process fluctuations, and random dopant fluctuations. The unique characteristics of each individual chip create a fingerprint-like response, making Intrinsic PUFs ideal for hardware authentication and identification purposes.

Common Implementations of Intrinsic PUFs are:

- Delay PUFs: Delay PUFs exploit the differences in signal propagation delay along various paths within the circuit. By measuring these delays, a set of unique challenge-response pairs can be generated, forming the basis for secure authentication.
- Ring Oscillator PUFs: Ring oscillator PUFs use the frequency differences in ring oscillators, circuits comprising an odd number of inverters. The varying delays in these oscillators result in distinctive response patterns, enabling the generation of cryptographic keys and unique device identifiers.
- SRAM PUFs: SRAM PUFs exploit the randomness in the power-up behavior of standard static random-access memory on a chip as a PUF.
- VIA PUFs: the Via PUFs technologies are based on "via" or "contact" formation during the standard CMOS fabrication process.

Extrinsic PUFs

Extrinsic PUFs differ from Intrinsic PUFs as they derive their responses from external environmental factors that influence the behavior of the chip. These external factors may include temperature variations, light levels, power supply fluctuations, and electromagnetic interference. The responses generated by Extrinsic PUFs can change under different operating conditions, leading to additional sources of randomness.

Common Implementations of Extrinsic PUFs are:

- Ambient Light PUFs: Ambient light PUFs utilize the incident light level on the chip's surface to create distinct response patterns. Variations in light intensity cause corresponding variations in the generated responses, enabling unique identification.
- Temperature PUFs: Temperature PUFs exploit temperature-induced changes in the electrical characteristics of the chip. Different temperature levels result in varied responses, contributing to the unclonable behavior of the device.

2.2 Fixed Pattern Noise

Noise on digital cameras has a lot of sources, but two main categories can be identified

1. **Random Noise**: Not constant from frame to frame and can be reduced using frame averaging. In digital images, *temporal noise* is often observed. It is completely random and results from variations in generating a value for a single pixel, converting photons into electrons. It is related to *photon shot noise*, which depends on the number of photons that hit a single pixel during a prolonged exposure.
2. **Fixed Pattern Noise**: Caused by small differences in sensors pixels, often noticeable during long exposure shots. These small differences can be caused by variations in pixel size, or interferences in the sensor circuitry. External factors (such as temperature or exposure times) can also affect this type of noise.

   CMOS-based imaging devices are primarily affected by pattern noise. This is due to amplifiers present in the pixel columns and in all the CMOS sensor, introducing additional noise.

Fixed Pattern Noise comes from two main sources: *PRNU* (Photo Response Non-Uniformity) and *DSNU* (Dark Signal Non-Uniformity). Figure 2.1 shows fixed pattern noise on the background, the column patterns are present all over the image. Frame averaging usually works to eliminate random noise, but it worsen fixed pattern noise.

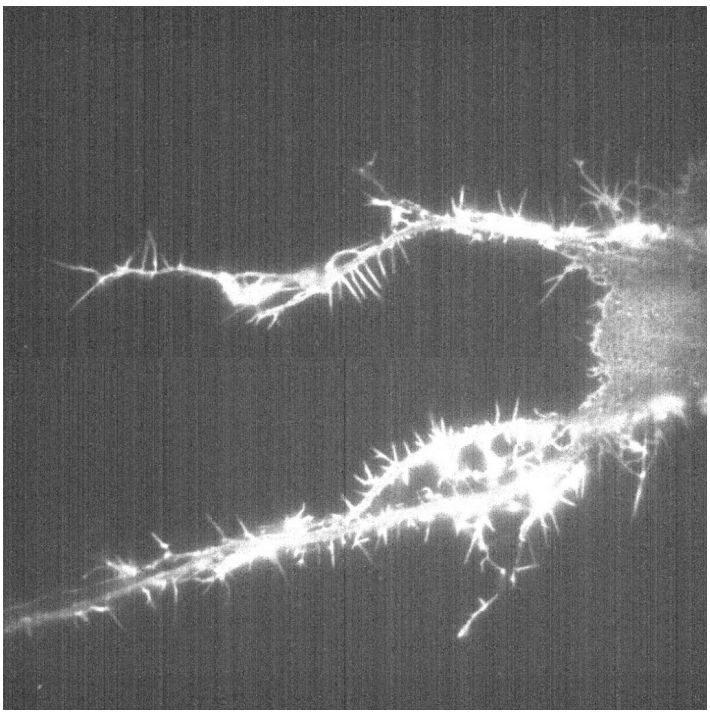

Figure 2.1: CMOS camera capting neurons. Average of 100 frames with 30ms exposure time.

PRNU is caused by responsitivity variation between pixels and it is the dominant noise in illuminated images. PRNU has a lot of properties that make it useful as an image fingerprint. It survives lossy JPEG compression and for this reason it has been used for source identification or forgery detection. This property, however, makes PRNU bad as the basis of a PUF. Since it survives JPEG compression, the PRNU fingerprint can be obtained by anyone, if the image is shared without proper access control (an example would be an image shared on social media).

To address this security problem, the PUF design uses DSNU fingerprint as its basis. DSNU is more difficult to be extracted in illuminated natural images, since it is not the dominant noise.

2.2.1 DSNU

Noise in digital cameras has a fluctuating behavior, and certain random noises errors can be positive or negative, which means that with a signal of zero and some negative noise, a pixel could have a negative value. From a software point of view, a negative value can be problematic. To avoid this, a positive offset is added to each pixel's value.

This offset creates a background effect so that, even without light, each pixel has a non-zero value, which is called *bias*. However, this bias is not constant through each pixel, because of CMOS column-to-column variations. This causes a fluctuation in the bias, which is the **DSNU**. Figure 2.2 shows DSNU on a dark image with no illumination. While illumination does not affect DSNU, temperature and exposure time does (increasing exposure causes sensor to heat).

2.3 Challenge-Response Authentication

Challenge-Response Authentication is a security protocol used to verify the identity of a user by providing a challenge, to which the user must respond correctly. Often, the challenge would be a random nonce (random number used only once, to avoid replay attacks). The user applies a transformation

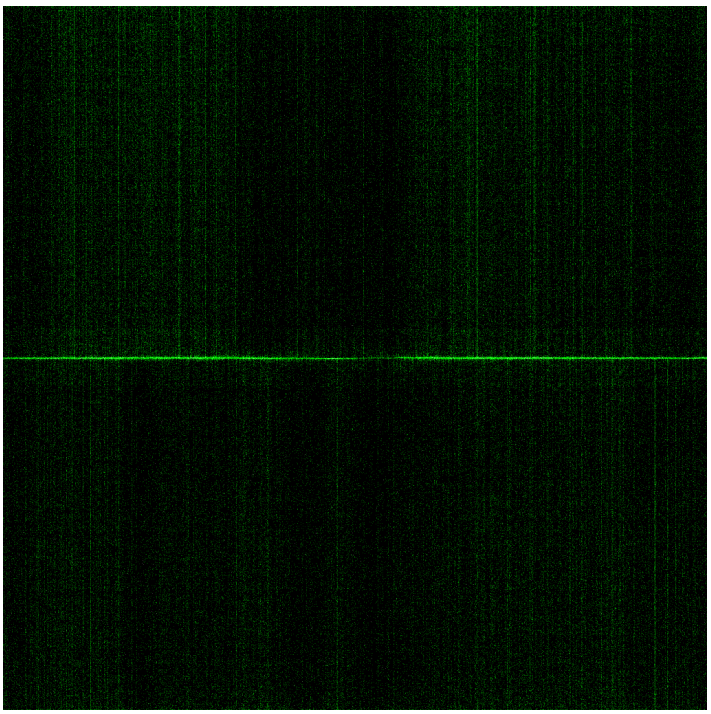

Figure 2.2: example of DSNU, 100 frames average with no illumination. The bias fluctuation is seen as regular column patterns.

on the random nonce and sends the response to the verifier. The CamPUF design uses the DSNU fingerprint as the basis for the challenge-response authentication protocol. This mechanism has the advantage of having a light communication overhead between the user's device and the verifier, which is useful, since CamPUF is created with low-power devices (mobile devices) in mind.

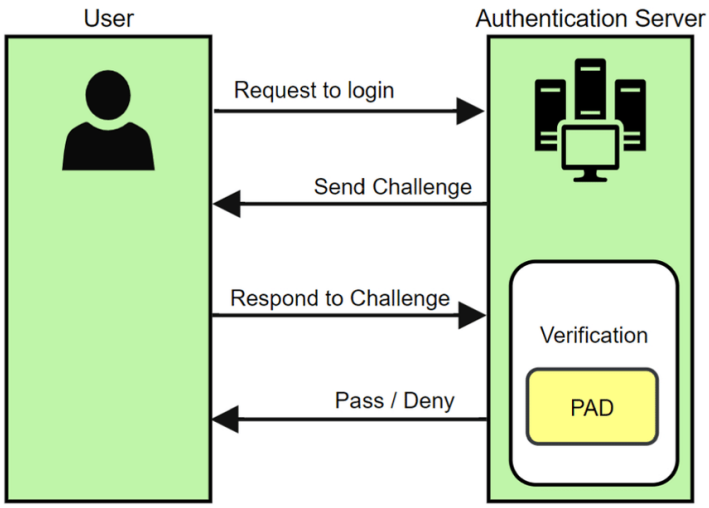

Figure 2.3: Simple schema which shows the Challenge-Response protocol

The Figure 2.3 shows all the protocol steps and parties involved.

CHAPTER 3

Implementation Overview

In this chapter you should provide a general overview of the project, explaining what you have implemented staying at a high-level of abstraction, without going too much into the details. Leave details for the implementation chapter. This chapter can be organized in sections, such as goal of the project, issues to be solved, solution overview, etc.

It is very important to add images, schemes, graphs to explain the original problem and your solution. Pictures are extremely useful to understand complex ideas that might need an entire page to be explained.

Use multiple sections to explain the starting point of your project, the last section is going to be the high-level view of your solution...so take the reader in a short 'journey' to showcase your work.

3.1 Introduction

The goal of this project was to design a CMOS-based PUF on a CMOS camera-based device. The initial phase of the project involved an extensive literature review, to explore the state of the art for CMOS-based PUFs. After gathering documentation about various implementations, the goal was to identify a reproducible PUF design and then to implement it, using a device of our choice.

3.2 Research Process

The goal of this phase was to identify a well-documented and reproducible PUF design that could serve as a basis for our implementation. Some material was already available through the project proposal. A comprehensive search through the internet was conducted and led to some other papers related to PUF technologies and various PUF designs.

The usage of smartphone camera as a PUF basis is a relatively recent field, but a lot of PUF designs have been proposed. From exploiting the randomness of oxide breakdown in CMOS sensors, to pixel variation noise. Even SRAM memory modules attached to some camera models were exploited to generate PUFs.

A particular implementation emerged as a candidate for the project. It contained a complete description of the design implementation from a mathematical point of view. The setup used for the experiment validation was well described, together with the theory principles that were at the basis of the design. The implementation is called CamPUF and the research paper is referenced in [?]

3.3 Methodology

Here are underlined some design choices that were made in order to adapt the implementation to suit the objectives of this project. The source code is written in **Python**. This language seemed the optimal choice for the project. It is a general purpose programming language, one of the must popular ones and for this reason it is rich of extensions, libraries and modules with comprehensive tools to deal with all kinds of paradigms and science fields. Libraries such as OpenCV, scipy and numpy provided us with the tools to do signal data processing.

A particular challenge was given by the image dataset choice. The program works with RAW images to extract the pixels values. A small dataset of dark images was obtained using a medium-end digital camera (Canon EOS 750D). A lot of problems were encountered testing this dataset, the hamming distance values with these images were very high, which means that authentication using the same dataset was unsuccessful. Another small dataset was tested, with raw images provided by an Iphone and the same problem was encountered.

An explanation for this behavior could be that the images were not completely dark, which means that the DSNU extraction phase captures other type of noise other than DSNU, leading to a wrong fingerprint or sensors in modern phones could have DSNU removing capabilities, which leads to the same result: a lot of random noise on data extraction. The choice fell on a dataset provided by the authors of the CamPUF design that used two Android phones from the last decade, plus an application developed by them which uses the Camera2 API. The DSNU in this dataset was maximized, setting the ISO to the maximum and shutter speed set to the minimum. More information is disclosed in the 5 chapter.

3.4 Solution Architecture

3.4.1 DSNU Fingerprint Extraction

The first part of the implementation consists in a Signal Data Processing part, specifically the extraction of the DSNU fingerprint from a couple of frames of a dark image. An average of the frames is done to remove temporal noise components and a **denoising filter** is applied to eliminate noise residual. A **DCT Filter** is also used for removing low-frequency components, so that the key generation is based only on high-frequency components, which are discarded in the JPEG compression algorithm. This simple data processing operations are performed using the external libraries previously mentioned.

3.4.2 Enrollment phase

At this point, the device has generated its DSNU fingerprint and wants to register it to the *authenticator* server. The goal of this phase is to register the location of the brightest and darkest pixels of the high frequency component. The indices of all these locations are stored and then sent to the authenticator.

3.4.3 Authentication phase

This phase is based on comparing the relative brightness of the registered pixels with the response sent from the device. The *Hamming distance* is used as a metric for measuring the similarity between the two compared keys. A threshold is selected a priori to discriminate between legitimate and illegitimate response.

Figure 3.1 shows the authentication flow of CamPUF.

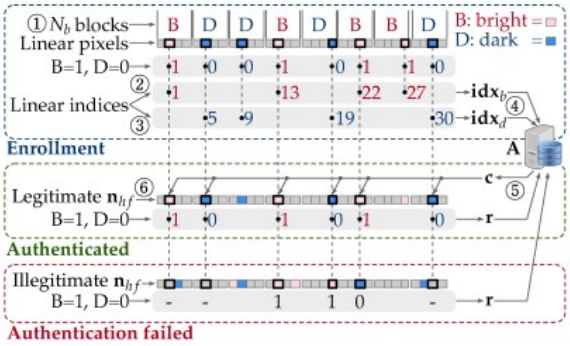

Figure 3.1: Device authentication flow

CHAPTER 4

Implementation Details

4.1 Prerequisites

4.1.1 Definitions

The following are the elements used to explain the enrollment and authentication phase.

- L: length of the key. 256 in the proposed implementation.
- N_m: noise margin.
- N_b: number of blocks in which the fingerprint is divided. Given by $L + N_m$
- D: number of pixels in the image.

4.1.2 Code structure

All the code can be found inside the `src` folder. The files are

- `constants.py`. The project-wide constants are set there. The `dataset_path` variable determines which is the base directory of the dataset.
- `extract_dsnu.py`. The purpose of this module is providing the `get_hf_noise` function, which returns the fingerprint of a given image.
- `server.py`. All the server-related operation, like the indeces registration, are implmented here.
- `device.py`. This file exposes a function that derives the response key.
- `auto_testing.py`. This module is what is run

4.1.3 Setup

The application has been tested with Python 3.11.3 running on Arch Linux, with Linux 6.4.7.

4.2 DSNU fingerprint extraction

The DSNU fingerprint extraction is implemented in the module `extract_dsnu.py`. The only function called outside the module is `get_hf_noise`, while all the others are used internally.

The function `get_hf_noise` returns a Numpy array containing the high frequencies of the noise, and accepts as inputs a list of paths pointing to the images, width and height of the images and a boolean variable, used to determine if the resulting noise should be shown or not.

In order to obtain the fingerprint, the images are processed through the following steps.

1. **Calculating the average between images.** Before being processed, the images are opened in different ways, depending on their extension. After that, because the RAW images in the chosen dataset do not contain information about their height and width nor margins, they are reshaped using the two input parameters and the margins are cropped. In that way, a 2-D Numpy array is obtained independently the image format. This is, then, summed into an accumulator. This is sub-steps are repeated for each image in the list. Finally, when the loop ends, the average is computed, dividing the accumulator by the length of the input list.

2. **Denoising the image.** At this point, the average frame is denoised, invoking the `wiener` function included into `scipy.signal` module. The second parameter of the function indicates the windows dimension in which the filter is applied, 5×5 in this case. It should be noted that the filtering produces a floating point matrix, although before the filtering the images is an integer matrix. This is the reason of the interpolation after the filter.

3. **Retrieving the noise.** The noise is simply obtained subtracting the original image, with its denoised version. The function responsible of that is `absdiff` contained into the cv2 library.

4. **Removing the low frequency components.** In this step the unique noise pattern should be already available, but it could be also retrieved in dark images that could be shared online. Since online images are usually compressed, cutting high frequency components, only these components should be used to extract the fingerprint. Thus, a high-pass filter is applied to the noise. This is done firstly transforming it into the *discrete cosine domain*, using the function `dct2`, which is a wrapper of `dct` function provided by `scipy.fftpack`, suited for a two dimensional array. Then, a filtering matrix is built. Each element $d_{i,j}$ of the matrix is defined as

$$d_{i,j} = \begin{cases} 1, & \text{if } i \geq H \cdot c \text{ and } j \geq W \cdot c \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

   where H and W are the height and the width respectively, and c is the cutoff constant between 0 and 1. c is set to 0.5, resulting in a matrix of non-zeros values in a quarter only at the bottom-right.

   The actual filtering is realized multiplying the filtering matrix and the DCT noise using `np.multilply`, which implements the *Hadamard product*. The last step of the extraction is the the inverse of DCT operation, returning the noise to the original domain.

5. **Plotting** When the function `get_hf_noise` is called, an additional flag `plot_results` could be set to plot the resulting noise. If the flag is true, then three graph will show the original image, the noise of the image and the high frequency of the noise. This could be useful to check if some pattern are evident by visual inspection.

4.3 Authentication server

The server code is located in the `server.py` module. A minimal but functional server should be able of enrolling and authenticating a device.

4.3.1 Enrolling a new device

The enrollment operation is implemented by the function `enroll`. It requires the high frequency noise as input, as the server should not receive the RAW images. The function produces a pair of arrays, containing one the linear indeces of bright pixels and the other of dark pixels, that are used in the authentication phase to generate the challenge to send to the device requiring authentication. The actual indeces are computed by an auxiliary function, `get_indeces`. This function performs the following steps. The input matrix is flattened into an array, and divided into N_b blocks. It could happen that the division results into blocks of different lengths; in these cases, given L the length of the array, `np.array_split` adds an element to the first $L\%N_b$ elements. This detail must be considered when the linear index of a certain pixel is derived, indeed an additional $L\%N_b$ term must be added to each element within a block with an index bigger than the reminder itself. At this point, for each block the brightest pixel, corresponding to the biggest element, is found, and a new list of these pixels is built, keeping track of their block and linear index. Sorted the new list by brightness, from the brighter half the `idx_bright` is built by means of pixels linear indeces. Finally, in the other half of the blocks, the darker pixels are sought, and, similarly to before, `idx_dark` is built.

4.3.2 Authenticate the device

Creating the challenge. When a device requires authentication, the server sends a challenge to it. The challenge depends on the previously `idx_bright` and `idx_dark` registered lists. The function that creates the challenge is `get_challenge`. Using the function `random.sample`, $L/2$ elements are selected from `idx_dark` and the remaining $L/2$ from `idx_bright`. The challenge is, thus, derived merging the selected values into a single array, and then sorting it.

Checking the response. After the device sends back the response key, the server must compare it with the reference key. Since the server derives at run time the key, it should be computed before. The function `get_reference_key` is called with the challenge and the bright indeces as parameters. It just creates a list of $0s$ and $1s$, returning a 1 if the i-th element of the challenge is in `idx_bright`, otherwise a 0. The server function that determine if the device is authenticated is `authenticate`, that takes the reference key and the response key as inputs, and returns true or false, depending on the outcome. An additional information about the hamming distance between the two keys is returned as well, although it is more a debugging information. The function `are_equal` computes the hamming distance and compares it with a threshold defined in `constants.py`. If the distance does not exceed the threshold, the function returns true, authenticating the device.

4.4 Device to authenticate

The device produces the response key by means of the function `get_responde_key`. Using the fingerprint and the challenge, a list of integer is built. First of all, using the indeces in the challenge, the corresponding pixels are selected and stored in a list. Then, the median of the list is computed, and finally the response key is built. If the i-th element is greater than the median there is a 1, otherwise it is a 0.

4.5 Testing

The interaction between the device and the server is tested inside `auto_testing.py`. In this script, the fingerprint of the given image(s) path is extracted and enrolled once. Then, for each file in the authentication file list the authentication is tried, following the steps shown before.

CHAPTER 5

Results

5.1 Testing of the CamPUF

[TODO: write]

5.1.1 Dataset

After some testing, this dataset was eventually chosen [**?**]. It is composed of various sets of both RAW and JPEG images taken with five different 12-megapixel Sony IMX377 camera sensors, used by Google Nexus 5X smartphones [**?**]. The key aspect in the preference of this dataset over the others tested is that all the RAW images are completely dark photos taken with the sensor lens fully covered. This is critical since the DSNU extraction efficiency is highly influenced by the presence of any light source exposed to the sensor.

Another advantage of choosing this dataset is that it was made with the purpose of testing another CamPUF implementation, having different relevant configuration ready to test, as for images taken in different room temperatures (25°C, 35°C and 45°C). It is worth noting that for any real-world practical implementation, the images provided to the enrollment/authentication algorithm should be taken in a similar way, in RAW format and trying to cover the sensor lens as much as possible.

[TODO: referring to pictures below explain why newer sensor models could have less DSNU noise to rely on]

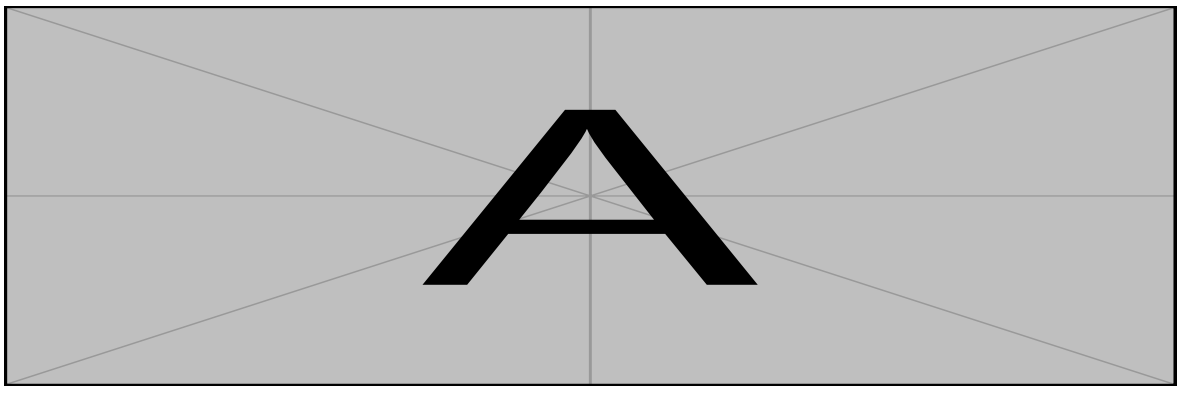

Figure 5.1: This is the image *caption*.

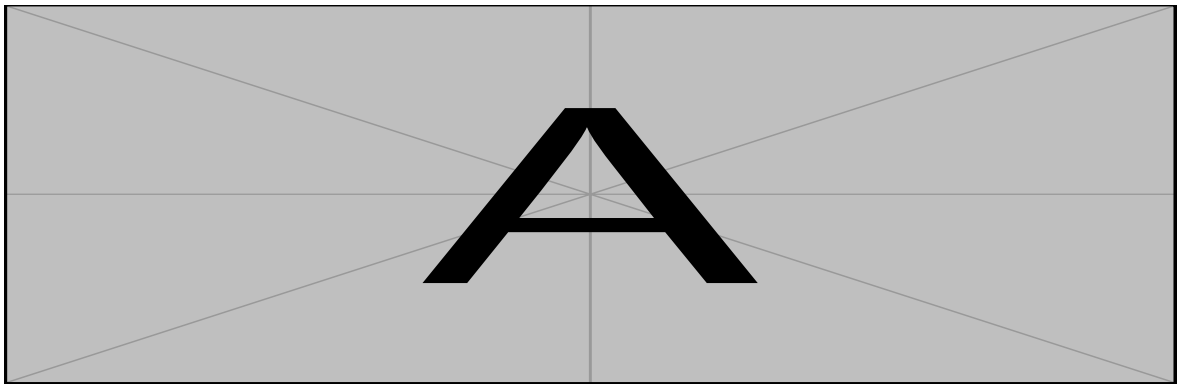

Figure 5.2: This is the image *caption*.

5.1.2 Setup

The dataset was then thoroughly tested using a python script *auto_testing.py* that automated the following steps:

1. Reading one or multiple images (and in this case, making the pixel-wide average).
2. Obtaining the reference key after enrolling with that image.
3. Trying to authenticate with a set of images, one after the other, by comparing the response key with the reference key generated from the previous step. If the Hamming Distance of the two keys is below a certain threshold, then the couple (*enrollment image*, *authentication image*) is said to be authenticated.
4. After trying all the images combinations, the script computes the Hamming Distance average for the couple (*enrollment image*, *authentication set*).

The main difference when picking the *authentication set* to use is the source sensor of the images with respect to the source sensor of the *enrollment image*. Two different cases are distinguished:

- **Intra-Sensor** testing: both the *enrollment image* and the *authentication set* come from the same source sensor.
- **Inter-Sensor** testing: The *enrollment image* and the *authentication set* come from a different sensor.

Parameters

The main parameters used were generally:

- **Key Length**: 256 bits, both for the reference key and the response key.
- **Hamming Distance Threshold**: 4 bits.
- **Number of Frames Averaged**: 5 frames for the enrollment, 1 frame for the authentication.
- **Number of Images tested for Authentication**: generally 50 for each run, 20 in the case of temperature testing. **??**

Expected Results

For the algorithm to be of any use, when testing **Intra-Sensor** couples it is expected to yield positive results on the authentication, meaning that the *Intra-Sensor Hamming Distance* between the reference key and the response key must be low enough, ideally zero. This is needed in order to avoid any false negatives where devices are incorrectly unauthorized.

Another crucial assumption is related to the opposite case, where **Inter-Sensor** couples are expected to yield negative results on the authentication, meaning that the *Inter-Sensor Hamming Distance* between the reference key and the response key must be high enough, in this case ideally around 50% of the key length to match the expected Hamming Distance of two random bit sequences. Finally, this is also needed to avoid any false positives where devices are incorrectly authorized.

5.1.3 Results

The results were gathered after several runs of the *auto_testing.py* script. The results obtained from the testing process are in line with the expected outcomes, validating the effectiveness of the CamPUF implementation. The graph below presents the Hamming Distance percentages for different scenarios, considering 1 frame and 5 frames for both **Intra-Sensor** and **Inter-Sensor** testings:

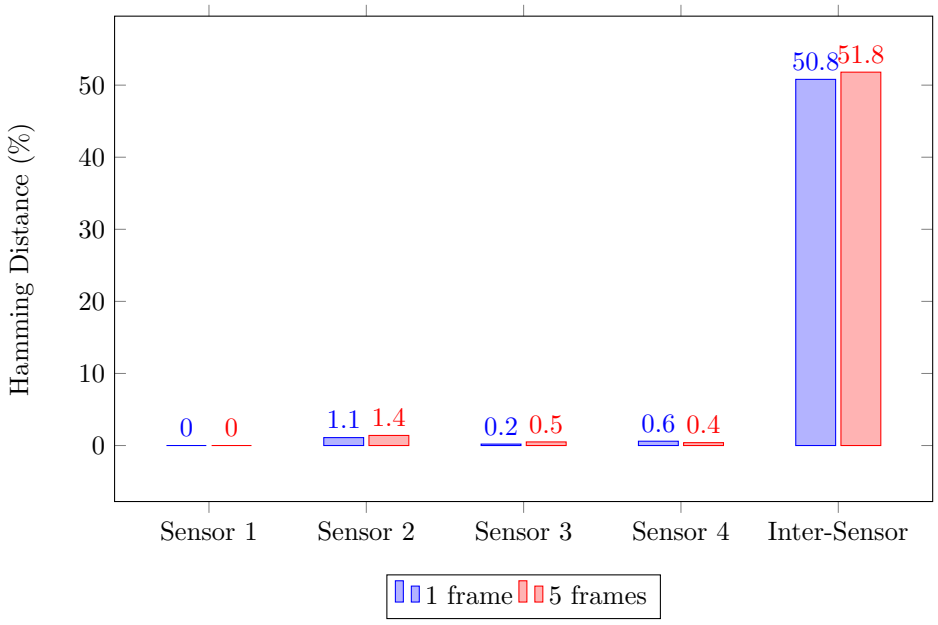

Figure 5.3: Hamming Distance Percentages for Intra-Sensor and Inter-Sensor Testings

As evident from the graph, the CamPUF algorithm performed exceptionally well in **Intra-Sensor** testing. Even with just one image for enrollment, the Hamming Distance between the reference key and the response key was found to be remarkably low. This demonstrates that a single image is sufficient to achieve accurate intra-sensor authentication, showcasing the efficiency of the algorithm.

Moreover, it was observed that a relatively low threshold of 4 bits is adequate to determine authentication success. This means that if the Hamming Distance falls below the threshold, the authentication process is successful. Fine-tuning this threshold allows for balancing between security and convenience. Lowering the threshold can increase security but may require more attempts to successfully authenticate. Conversely, increasing the threshold may reduce security but lead to faster and easier authentication.

The **Inter-Sensor** testing results align with expectations, showing Hamming Distances around

50%, which is in line with the expected Hamming Distance between two random bit sequences. This confirms that CamPUF effectively discriminates between images taken from different sensors, providing a reliable mechanism to prevent unauthorized access.

In conclusion, the testing results demonstrate that the CamPUF implementation is capable of providing robust and accurate authentication.

Temperature Resilience

One critical aspect of the CamPUF implementation is its resilience to changes in temperature, which can impact the quality of sensor outputs. Higher temperatures often lead to increased random noise coming from the electrons inside the CMOS sensors, resulting in less noticeable DSNU noise that the algorithm relies on. We conducted testing under varying temperatures to assess the algorithm's performance in such scenarios.

The sensor tested for Intra-Sensor testing was Sensor 1, and the temperature changes refer to the room temperature relative to the images used for the authentication phase. The enrollment was always done with a room temperature of 25°C.

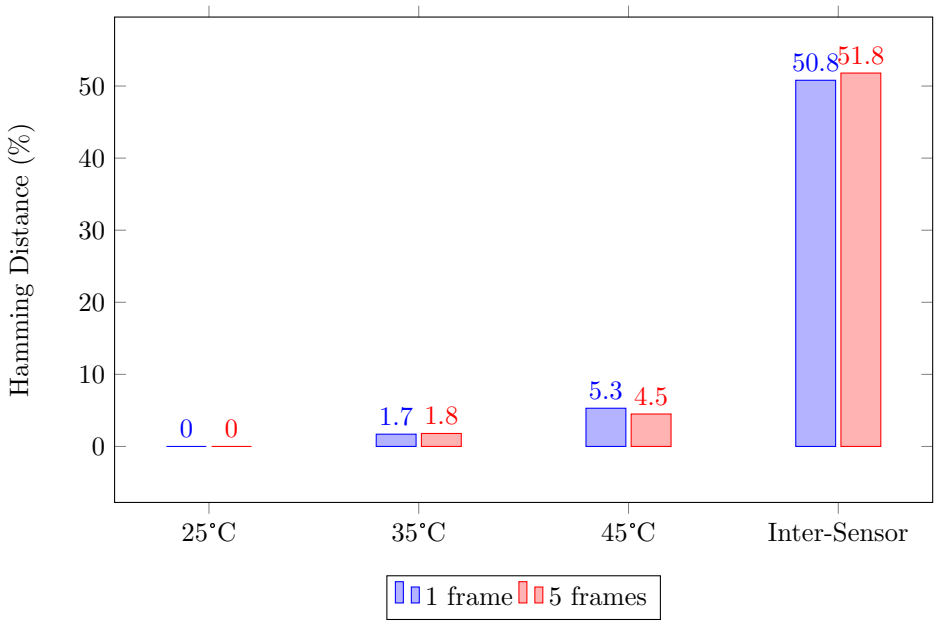

Figure 5.4: Hamming Distance Percentages for Intra-Sensor and Inter-Sensor Testings at Different Temperatures

As expected, the results show that the average Intra-Sensor Hamming Distance rises with increasing temperatures. At 25°C, the Hamming Distances remain negligible (0.0%) for both one frame and five frames enrollments, indicating that even with minimal data, the algorithm achieves accurate authentication. However, at higher temperatures of 35°C and 45°C, the Intra-Sensor Hamming Distances slightly increase to 1.7% and 5.3% (for one frame enrollment), and 1.8% and 4.5% (for five frames enrollment), respectively.

Despite the slight increase in Intra-Sensor Hamming Distances with rising temperatures, they remain significantly low compared to the Inter-Sensor Hamming Distances, which are consistently around 50.8% for one frame enrollment and 51.8% for five frames enrollment. This highlights the algorithm's robustness and its ability to maintain reliable authentication even under varying temperatures.

In conclusion, the CamPUF implementation demonstrates promising temperature resilience. While the average Intra-Sensor Hamming Distance increases with higher temperatures, it remains negligible

compared to the Inter-Sensor Hamming Distance. This reinforces the algorithm's effectiveness in providing secure and accurate device authentication across different environmental conditions.

5.1.4 Attacks Mitigation

TODO:

5.2 Known Issues

The main limitation of this implementation of CamPUF is the need for a dataset of images highly similar to the one used for testing described in 5.1.1. Even if the TODO:

5.3 Future Work

CHAPTER 6

Conclusions

This final chapter is used to recap what you did in the project. No detail, just a high-level summary of your project (1 page or a bit less is usually enough, but it depends on the specific project).

Bibliography

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

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

APPENDIX A

User Manual

A.1 Dependencies

- `Git`. In order to clone the repository with the project.
- `Python 3.11+`

In addition, the following Python libraries must be installed

- `matplotlib`. Graph library for math visualizations
- `scipy`. Python library that provides mathematical instruments and algorithms
- `cv2`. Computer vision library, used to extract data from raw images
- `rawpy`. Library used to open RAW images such as CR2.

A.2 Installation

In a shell clone the project with the following command

```
$ git clone https://github.com/doreado/camPUF.git
```

A.3 Prepare the images dataset

An images dataset could be found at `https://wisest.ece.wisc.edu/research-old/mobile-and-embedded-security/campuf-dataset/`.

Inside *src/auto_testing.py* the `img_test_enroll_dir` and `img_test_auth_dir` variables must be set to the folder containing the enrollment and authentication images respectively. The script searches for images automatically. The number of images used for the enrollment can be adjusted modifying the `num_frames_enroll` (the default value is 5). After the enrollment, the script tries to authenticate all the images in the authentication folder.

A.4 Run the script

```
$ cd camPUF/src
$ python3 auto_testing.py
```

If everything works, the output shows some information and the authentication outcomes, as shown in figure A.1

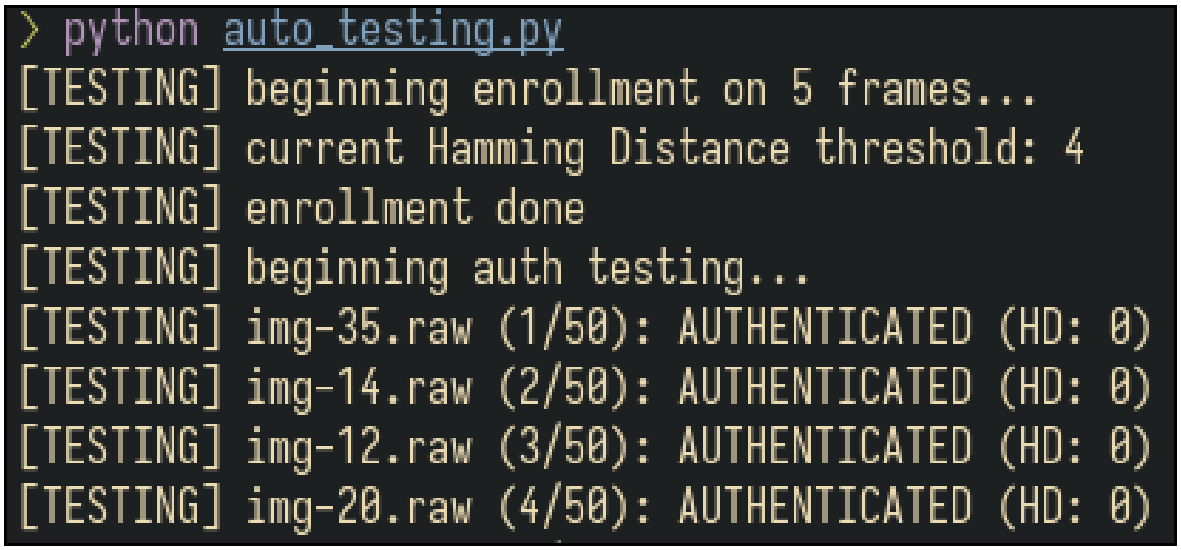

Figure A.1: The output of the script has this format