Lab 01: PUF Evaluation

Ganesh Veluru and Jwala Sri Hari Badam
University of South Florida
Tampa, FL 33620

I. Introduction

Process variation refers to the natural and unavoidable deviations in the physical characteristics of electronic components, particularly transistors, during semiconductor manufacturing. [1] These variances occur due to factors like fluctuations in materials, temperature, and environmental conditions during the fabrication process. Process variation leads to subtle differences in transistor behavior, resulting in unique electronic fingerprints for each chip, even within the same production batch. Process variation is harnessed in hardware security through Physical Unclonable Functions (PUFs). PUFs utilize these inherent variations to generate unique and unpredictable responses to specific challenges or inputs. These responses serve as individualized digital fingerprints for each hardware device, making it extremely challenging for attackers to replicate or counterfeit devices. PUFs enhance security by leveraging these uncontrollable variations, providing a robust defense against current attacks like counterfeiting, unauthorized access, and reverse engineering, while also enabling secure authentication and cryptographic key generation.

The current project involves the evaluation of the quality of Physical Unclonable Function (PUF) by employing two FPGA (Field-Programmable Gate Array) boards. PUFs will be implemented on these boards and here we are using one particular PUF called Arbiter PUF, and the assessment will encompass two primary aspects: reproducibility and uniqueness. To evaluate reproducibility, the project will investigate whether the same challenge input applied to a PUF on the same FPGA board yields a consistent response under varying environmental conditions, such as temperature changes. This comprehensive evaluation aims to establish the reliability and robustness of PUF responses in different operational settings. Additionally, the project will examine the uniqueness of PUFs by analyzing whether two separate FPGA boards produce distinct outputs for identical challenge inputs. This investigation is crucial for assessing the reliability and effectiveness of PUF-based security measures and their suitability for real-world applications across diverse environmental conditions.

II. Reading Check

Question 1: What is the purpose of the arbiter in the APUF

Answer: In this project, we are implementing an APUF, where the fundamental building block is a switch comprised of two multiplexers (MUXes). Both MUXes receive the same challenge input, and their behavior, involving the selection of input to output, is influenced by this common challenge. The APUF structure comprises two distinct paths, termed the ”top” and ”bottom” paths. Each stage of the circuit features a multiplexer (MUX) that has the capability to potentially switch the paths and the schematic diagram is shown in Fig 1. These paths involve different transistors, resulting in slight delays that contribute to the APUF's uniqueness. The challenge input serves as the trigger for the APUF's operation, initiating a race between the two paths. This race is characterized by a pulse or transition from logic 0 to logic 1. As the pulse travels along the paths, it occasionally switches sides due to the MUX configurations. Ultimately, the race concludes at an Arbiter.

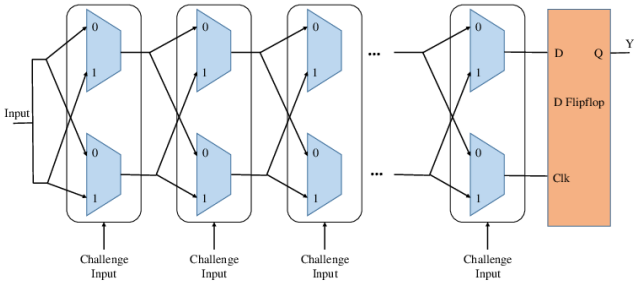


Fig. 1: Conventional Arbiter PUF Design

The purpose of the arbiter in an Arbiter Physical Unclonable Function (APUF) is to determine which of the two racing signal paths (the ”top” and ”bottom” paths) arrives first at the end of the circuit. It monitors the signals and decides whether the signal from the top path or the bottom path reaches it first. It functions as a decision-maker, specifically a D flip-flop (DFF), that assesses the timing of signal arrivals. If the signal from the top path arrives first, indicating a logic 1, it means that a logic 1 is stored in the DFF by the time the signal reaches the clock input. Conversely, if the signal from the bottom path wins the race, a logic 0 is stored in the DFF. This result is then transmitted to the DFF's output, often denoted as 'Q,' forming one bit of the APUF's response. The arbiter's role is critical because it adds an element of unpredictability to the APUF's responses. The slight differences in signal arrival times and path swaps contribute to the uniqueness and security of APUFs, making them suitable for authentication and cryptographic applications.

Question 2: Why might the PUF response bits change when different challenges are applied to the same circuit on the same

FPGA

Answer: The change in response bits when different challenges are applied on same circuit and same FPGA board is mainly due to PUF that contains complex circuits with multiple stages and configurations will result in divergent responses when there is even a small change in the challenge/input. Especially in APUF, it uses challenges to determine the paths that signals take within the circuit. Even small changes in the challenges can lead to variations in the paths that signals follow. This can cause different responses as the signals navigate through the circuit. The change in responses will also occur due to environmental conditions and manufacturing variations like change in temperature, voltage levels and differences in components. Inside the FPGA, there are tiny differences in how its components work, like the speed at which they operate. These differences are inherent and can't be controlled. When you give the PUF different challenges, these tiny variations interact with the challenges, resulting in different response patterns.

Question 3: Why might the PUF response bits change when the same challenges are applied to the same circuit on different FPGAs

Answer: The change in response bits when same challenges are applied on different FPGA with same circuit is mainly due to manufacturing variations as each FPGA undergoes a manufacturing process that introduces tiny, uncontrollable variations in its electrical properties. These variations can include differences in transistor characteristics and interconnects. Even though they may have the same design, minute differences in components and manufacturing processes can result in variations, causing the PUF responses to differ. Another possible reason is small variations in timing, which can be influenced by environmental factors or manufacturing differences, can lead to differences in the timing of signals within the FPGA circuit. This, in turn, affects PUF responses. This property is the key in implementing hardware security mechanisms using PUFs.

III. Methods

In this experiment, we are utilizing CMOD S7 boards, as depicted in Fig 2. We have a PC and PHS- virtual machine at our disposal, and within this virtual environment, we are executing scripts to gather responses. These responses are obtained by sending challenges to the FPGA board after the PUF has been implemented.

A. Hardware and Software Setup

Our procedure for Hardware and Software setup will take the following steps:

Step 1: Install VirtualBox - Begin by installing VirtualBox on our PC. VirtualBox is a hypervisor that enables us to run virtual machines (VMs) on a single physical host.

Step 2: Prepare PHS-VM - We've been provided with a pre-configured virtual machine called PHS-VM. This VM comes equipped with essential tools, Python libraries, USB drivers, and compilers required for interacting with the FPGA board. The virtual machine allows us to communicate with the FPGA board using Jupyter Notebook as our interface.

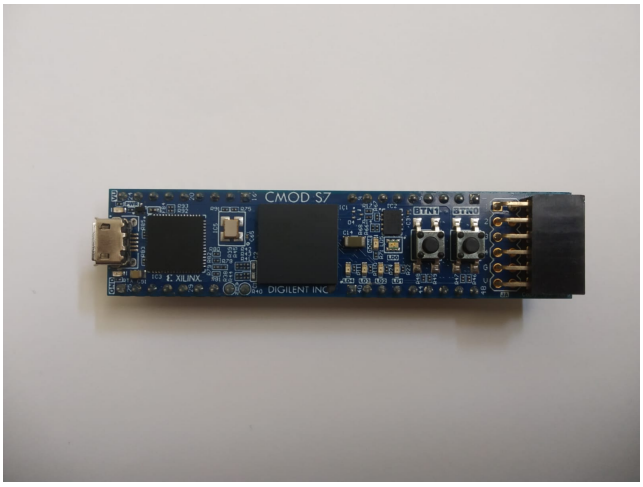


Fig. 2: CMOD S7 Board

Step 3: Import PHS-VM- Import the PHS-VM into VirtualBox by going to "File" and "Import Appliance." Once imported, we can start the VM by simply clicking on it. We will need to provide the necessary credentials to log in.

Step 4: Connect FPGA Board-After starting the VM, connect the FPGA board to our PC via USB. Upon successful connection, we should see a green light blinking on the FPGA board, along with one red static light. To confirm that the device is properly connected, we can navigate to "Device" and "USB" in the VM. We should see "Digilent Adept USB Device" listed if the connection is established correctly. If not, we should double-check our hardware, software installations, and connections. The complete setup of our experiment is shown in the Fig 3.

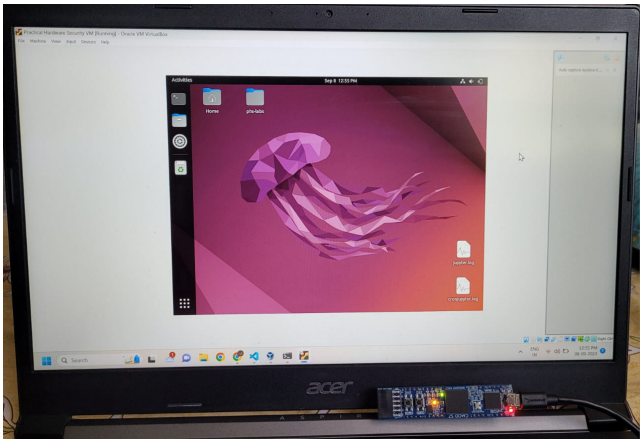


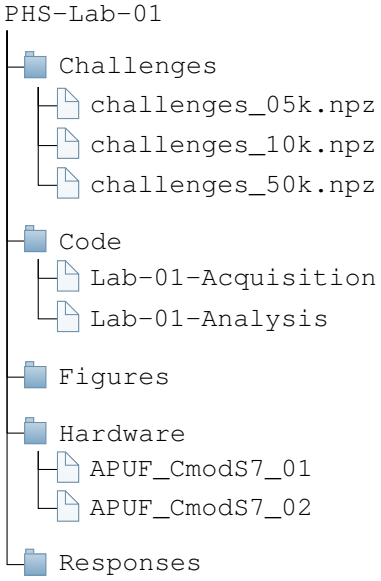
Fig. 3: Hardware and Software setup

Step 5: Access Jupyter Notebook-To start our investigation, we can open a web browser on our host machine and enter "http://localhost:8888/." This will allow us to access Jupyter Notebook locally. Inside Jupyter Notebook, we'll find various directories, and we should focus on the "PHS-Lab-01" directory.

Step 6: Unzip PHS-Lab-01 Content-Within the "PHS-Lab-01" folder, we can upload the provided "PHS-Lab-01.zip" file. Run the unzip code located in the "RUN-TO-ZIP.ipynb" file to extract the contents of "PHS-Lab-01.zip." Make sure that the name of the zipped file matches the current directory for successful extraction.

Step 7: Organizing the Extracted Content - After unzipping the provided archive, we'll observe the folder structure illustrated in the accompanying figure.

The directory is structured as follows:



a) The "Challenges" folder contains three challenge files in '.npz' format. '.npz' files are NumPy archive files in Python. They serve the purpose of serializing and deserializing NumPy arrays. Among these files, there are challenges of varying sizes: 5,000, 10,000, and 50,000 challenges, respectively. For our final analysis, we will use the responses generated based on the file containing 50,000 challenges.

b) Inside the "Code" folder, we discover two important files. The first is dedicated to "Data Acquisition." In this script, we will complete the necessary code to obtain responses for the provided challenges using the PUF implementation on the FPGA board. The second file pertains to "Data Analysis." Here, our objective is to analyze the acquired data using a combination of plots and statistical measures.

c) Figures generated during the analysis phase, such as plots and graphical representations, will be stored in the "Figures" directory.

d) Inside the "Hardware" folder, we find two essential bit files: APUF_CmodS7_01.bit and APUF_CmodS7_02.bit. These files represent the implementations of a 64x16 APUF (Arbitrarily Programmable Unclonable Function). This means they accept 64 bits as input and produce 16 bits as output. Additionally, the bit files contain a Universal Asynchronous Receiver/Transmitter (UART) implementation—a vital hardware communication protocol used for serial communication between devices. UART facilitates data transmission and reception between computers and various hardware devices.

e) The "Responses" folder will be the destination for storing the responses generated in response to the provided challenges. These responses are a crucial part of our experiment.

B. Experimental Procedure

Part A: Data Acquisition

Our procedure for Part A will take following steps:

Step 1: To send and receive data from the PUF, we connect the Cmod S7 board to our computer using a USB connection. This board has a special chip that helps translate USB messages into a format that a Universal Asynchronous Receiver/Transmitter (UART) circuit can understand. The provided PHSVM already has the necessary drivers installed, creating a Virtual Port. This allows us to send messages using software on our computer or through a programming language especially python with a serial library.

Step 2: To begin, we first import the necessary Python libraries and ensure that our FPGA board is properly connected. We use the command "! djtgcfg enum" to enumerate and list all connected Digilent JTAG devices, including FPGAs.

Step 3: Before we can send or receive data, a crucial step is programming the FPGA board with the provided bitstream files using a Bash script. This step is essential for configuring the FPGA.

Step 4: To establish communication, we need to determine which serial port the FPGA board is connected to. To accomplish this, we utilize the "serial.tools.list_ports.comports()" function in Python. Once we have identified the correct port, we initialize a serial connection using "serial.Serial(port, baudrate)" with the specified port and baudrate settings.

Step 5: Finally, we load data from an .npz file, which contains challenge data designed for the FPGA. This data is essential for the communication and interaction with the FPGA board.

Step 6: Next, our task is to transmit challenges and retrieve responses through the 'get_1resp()' function. This involves utilizing the serial connection object, providing the input string in hexadecimal format, and specifying the expected number of response bytes. Before sending the input to the FPGA, it's essential

to convert the hexadecimal string into bytes and then employ ‘ser.write()‘ for transmission. Subsequently, as we receive the responses, they will be converted into binary strings via ‘ser.read()‘. The code script to write input and read ouptut is shown in the Fig 4.

```python
def get_1resp(ser: serial.Serial, in_hex: str, resp_size: int) -> str:

    # convert hex string to bytearray
    in_bytes = bytearray.fromhex(in_hex)

    # send bytes to FPGA
    ser.write(in_bytes)

    # get response: reads n bytes from FPGA (remember: 1 byte = 8 bits)
    r_bytes = ser.read(resp_size)
    # convert bytearray to int
    r_int = int.from_bytes(r_bytes, byteorder='big')
    # convert int to resp_size*8 bits binary string
    r_bin = format(r_int, "0"+str(8*resp_size)+"b")

    return r_bin

print("✔️ OK to continue!")
```

Fig. 4: Python Script to Send Challenges and Reading Responses

Step 7: To handle all 50,000 challenges efficiently, we will employ a Python for loop to execute the ‘get_1resp()‘ function for each challenge. The resulting responses will be aggregated and stored in a NumPy array. It is imperative to save these responses in a file using ‘np.savez()‘, as they will be subject to further analysis in subsequent steps.

```python
def get_all_resp(challenges: np.ndarray, resp_size: int):
    # initialize an empty list (or NumpPy array)
    L=[]

    # iterate through each challenge and get respective response
    N = len(challenges)
    for i in tnrange(N , desc='Generating APUF Responses'):
        L.append(get_1resp(ser,challenges[i],2))
        # do something

    #print(type(L))
    # If you used a list, convert responses to Numpy Array
    responses=np.array(L)
    # return your NumPy array of responses
    return responses
```

Fig. 5: Python Script to get all Responses using Loops

Step 8: This procedure should be replicated for FPGA0, yielding collections of responses denoted as fpga0_resp0_50k, fpga0_resp1_50k, and fpga0_resp2_50k. Similarly, for FPGA1, the identical process should be carried out, resulting in datasets named fpga1_resp0_50k, fpga1_resp1_50k, and fpga1_resp2_50k. Up to this point, we have completed the data acquisition using the APUF_CmodS7_01.bit bitstream file. Now, we need to replicate the entire process for the APUF_CmodS7_02.bit bitstream file, targeting both FPGA0 and FPGA1.

Step 9: Consequently, we will obtain an additional set of six responses: fpga0_resp0_50k_grad, fpga0_resp1_50k_grad, fpga0_resp2_50k_grad for FPGA0, and fpga1_resp0_50k_grad, fpga1_resp1_50k_grad, fpga1_resp2_50k_grad for FPGA1. These responses will provide us with a comprehensive dataset for further analysis and evaluation.

Part B: Data Analysis

For the Analysis Part we are duplicating the Analysis Jupyter notebook for the second bitstream file implementation and renaming the files as Lab-01-Analysis_PUF1.ipynb and Lab-01-Analysis_PUF2.ipynb

After importing the necessary libraries and loading data from response files, our next step is to calculate both intra and inter Hamming distances. The Hamming distance(HD) is a measure of dissimilarity between two binary sequences of equal length. Intra-chip Hamming distance refers to the dissimilarity or difference between responses generated by the same FPGA board when exposed to the same set of challenges or inputs. This metric is often referred to as reproducibility, and ideally, it should be close to 0, indicating that the responses are highly consistent [2]

On the other hand, inter-chip Hamming distance measures the difference between responses generated by two different FPGA boards when exposed to the same set of challenges or inputs. This metric is associated with uniqueness, and an ideal value should be close to 50 percent, signifying a high degree of distinctiveness. To compute these Hamming distances, we utilize the spatial.distance.pdist() function with the metric set to ”hamming.” We calculate pairwise intra Hamming distances between the three responses obtained from FPGA0 and similarly for the responses from FPGA1. Additionally, for obtaining inter-Hamming distances, we compare responses between both FPGA0 and FPGA1. The Python script to calculate the pairwise intra HD is shown in Fig 6 and for inter HD

IV. Results

To calculate average inter-chip HD for two PUFs, Pi and Pj, implement the same PUF circuit and have N-bit responses Ri and Rj to the same challenge, C, then the uniqueness is given by the average inter-chip HD among the k devices and is defined as [3]:

$$Uniqueness = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD\,(Ri, Rj)}{n} X100 \quad (1)$$

The reliability metric is utilised to quantify the ability of a PUF design to reproduce a response. For a device Pi,

```
x0=fpga0_resp0
y0=fpga0_resp1
z0=fpga0_resp2

#gets pairwise HD between x,y,x
#pairwise_hd = spatial.distance.pdist([list(x),list(y),list(z)], metric="hamming")

#List to store all pairwise hamming distances for FPGA-0 responses
intra_HD_fpga0=[]

#Loop calculates the pairwise hamming distance between 3 responses of FPGA-0
for i in range(len(x0)):
    pairwise_hd_fpga0 = spatial.distance.pdist([list(x0[i]),list(y0[i]),list(z0[i])], metric="hamming")
    intra_HD_fpga0.append(pairwise_hd_fpga0)

# converting list to numpy array with HD probabilities
intra_HD_fpga0 = np.array(intra_HD_fpga0)

# Converting HD probabilities into percentages
intra_HD_fpga0 = intra_HD_fpga0*100
print("✔ OK to continue!")
```

Fig. 6: Python script to calculate pairwise intra Hamming Distance

```
x0=fpga0_resp0
x1=fpga1_resp0


#List to store the hamming distance between two responses
inter_HD = []

#Loop calculates the pairwise hamming distance between responses of FPGA-0 and FPGA-1
for i in range(len(x0)):
    pairwise_HD_inter = spatial.distance.pdist([list(x0[i]),list(x1[i])], metric="hamming")
    inter_HD.append(pairwise_HD_inter)

# converting list to numpy array with HD probabilities
inter_HD = np.array(inter_HD)

# Converting HD probabilities into percentages
inter_HD = inter_HD*100

print("✔ OK to continue!")
```

Fig. 7: Python script to calculate inter Hamming Distance

reliability is represented as a single value by finding the average intra-chip HD of s response samples, Rj; this is taken at different operating conditions compared to a baseline N-bit reference response, Ri, taken at nominal operating conditions. The average intra-chip HD is defined as follows:

$$HD\_INTRA = \frac{1}{s}\sum_{t=1}^{s}\frac{HD\,(Ri,Rj)_t}{N}X100 \qquad (2)$$

The Reliability can be represented as

$$Reliability = 100 - HD\_INTRA \qquad (3)$$

The overall minimum and maximum values for pairwise intra hamming distance and inter hamming distance are calculated using numpy min() and max() functions and for each pair wise the statistical results are calculated using pandas describe() where we will get mean, std, min, max values pairwise. The code snippet for this is shown in the Fig 8. To plot the results that we obtained we are using plt.hist() and the code snippet to draw the histogram is shown in Fig 9.

All these procedure have done on both PUF implementations APUF_CmodS7_01.bit and APUF_CmodS7_02.bit bitstream files.

```
def summary(np_arr):
    '''
        Input : Numpy array
        output : Summary DataFrame
    '''
    df= pd.DataFrame(np_arr)
    df = df.describe().round(3)
    df = df.loc[['min','max','mean','std']]
    return df
```

```
# intra-chip HD for FPGA0
print('{:=^80}'.format("Statistical Results for Intra-chip Hamming Distance for FPGA0"))
overall_min_HD_fpga0 = np.min(intra_HD_fpga0)
overall_max_HD_fpga0 = np.max(intra_HD_fpga0)
overall_std_HD_fpga0 = np.std(intra_HD_fpga0)


df_HD_fpga0= summary(intra_HD_fpga0)

# Changing the columns name to 'XY','XZ','YZ'
# XY represents pairwise HD between fpga0_resp0 and fpga0_resp1 similarly XZ and YZ.
df_HD_fpga0.columns = ['XY','XZ','YZ']
overall_avg_HD_fpga0 = (sum(df_HD_fpga0.loc[['mean']].values[0])/3).round(3)

print("Overall Minimum value:", overall_min_HD_fpga0)
print("Overall Maximum value:", overall_max_HD_fpga0)
print("Overall Standard Deviation value:", overall_std_HD_fpga0.round(3))
print("Overall Average value:", overall_avg_HD_fpga0,"\n")
print(df_HD_fpga0)
```

Fig. 8: Python script to get Statistical Results using Numpy and Pandas

```
plt.figure(figsize=(12, 8))

# Creating a histogram for inter HD
plt.hist(inter_HD, bins=np.arange(0,100,5),density = True,alpha=0.4, edgecolor='white',color='green')
mean, std = norm.fit(inter_HD)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = norm.pdf(x, mean, std)

plt.plot(x, p, 'k', linewidth=1.5)

plt.xlabel('Hamming Distance Percentage',fontsize=14)
plt.ylabel('Probability Mass Function',fontsize=14)
plt.title('Histogram of Inter-Chip Hamming Distances',fontsize=16)

plt.xticks(np.arange(0, 100, 5))
plt.xlim(left =0)
plt.savefig('../Figures/PUF1_Inter-Chip_HD.pdf')
plt.show()
```

Fig. 9: Python Script to represent data Graphically

The statistical results for pairwise intra HD for all 3 responses for FPGA-0 and FPGA-1 are shown in TABLE I. The table contains results for both bit file implementations.

The statistical results for intra HD after averaging pairwise values for FPGA-0 and FPGA-1 are shown in TABLE II. The table contains results for both bit file implementations.

The statistical results for inter HD for both bitstream file implementations are shown in TABLE III.

The figures illustrating the pairwise intra HD comparisons using the implementation of APUF_CmodS7_01.bit on FPGA-0 and FPGA-1 are presented in Fig 10.

Fig 11 displays the graphical representations of the pairwise intra HD for the implementation of APUF_CmodS7_02.bit on both FPGA-0 and FPGA-1.

Fig 12 illustrates the graphical representations of the intra HD for the implementation of APUF_CmodS7_01.bit on both FPGA-0 and FPGA-1.

TABLE I: Statistical Results for Pairwise Intra HD of FPGA-0 and FPGA-1

Parameter	**APUF_CmodS7_01**						**APUF_CmodS7_02**					
Board	**FPGA-0**			**FPGA-1**			**FPGA-0**			FPGA-1		
Pairs	**XY**	**XZ**	**YZ**	**XY**	**XZ**	**YZ**	**XY**	**XZ**	**YZ**	**XY**	**XZ**	**YZ**
Minimum	0	0	0	0	0	0	0	0	0	0	0	0
Maximum	50	50	50	56.25	50	56.25	50	56.25	56.25	56.25	50	50
Mean	0.41	0.43	0.41	0.36	0.39	0.34	0.44	0.44	0.38	0.34	0.35	0.32
SD	2.33	2.41	2.36	2.22	2.35	2.17	2.46	2.42	2.82	2.19	2.27	2.09

TABLE II: Statistical Results for Intra HD of FPGA-0 and FPGA-1

Parameter	**APUF_CmodS7_01**		**APUF_CmodS7_02**	
Board	**FPGA-0**	**FPGA-1**	**FPGA-0**	**FPGA-1**
Minimum	0	0	0	0
Maximum	50	56.25	56.25	56.25
Mean	0.42	0.36	0.42	0.34
Standard Deviation	2.37	2.25	2.39	2.19

TABLE III: Statistical Results for Inter HD of FPGA-0 and FPGA-1

Parameter	**APUF_CmodS7_01**	**APUF_CmodS7_02**
Minimum	0	0
Maximum	62.5	62.5
Mean	3.99	3.99
Standard Deviation	7.08	7.04

Fig 13 displays the graphical representations of the intra HD for the implementation of APUF_CmodS7_02.bit on both FPGA-0 and FPGA-1.

Fig 14 illustrates the graphical representations of the inter HD between FPGA-0 and FPGA-1 for both bit file implementations.

V. Discussion

From the statistical analysis, it is evident that the average intra-HD for FPGA0 is approximately 0.42 percent. This finding signifies that when the first bitfile is utilized, the PUF consistently generates nearly identical responses when exposed to the same set of challenges. Similarly, for the second bitfile, the average intra-HD is close to zero which is 0.42. This strong consistency in responses is a noteworthy characteristic of a robust PUF.Turning our attention to FPGA1, the mean intra-HD also hovers around zero for both bitfile implementations. This observation indicates that the PUF implementations on both FPGA boards exhibit intra-HD in line with expectations.

Furthermore, examining the standard deviation values for both FPGAs reveals that they are relatively low. A low standard deviation implies that the responses generated by each FPGA board maintain remarkable consistency and reproducibility. In essence, the responses cluster tightly around a central value, showcasing minimal variation between them. This consistency underscores the FPGA boards' capability to produce stable and dependable responses when presented with specific challenges. Such reliability is of paramount importance in diverse applications, especially those related to security and authentication.

When we examine the inter-HD between FPGA0 and FPGA1, we find that they are approximately 4 percent. A lower inter-Hamming distance is indicative of similarity between the responses generated by two distinct entities, like the two FPGA boards in our case, when exposed to the same challenges. In simpler terms, when we compare the responses produced by these two entities, we observe less dissimilarity between them. In the context of our experiment, a low inter-HD between responses from these two different FPGA boards implies that the responses they produce are relatively similar in terms of their binary patterns. This similarity suggests that there may be limited variation in how these FPGA boards process and react to identical challenges. While a low inter-HD can be advantageous in specific scenarios, such as when consistency and similarity in responses are desired, it may raise concerns in applications where the uniqueness and unpredictability of responses are vital. For instance, in security or authentication systems, it's essential to have responses that are distinct and difficult to predict.

There could be several factors contributing to the observed low inter-HD between the responses generated by the two FPGA boards. One factor is that both FPGA boards were configured with identical PUF implementations. Consequently,

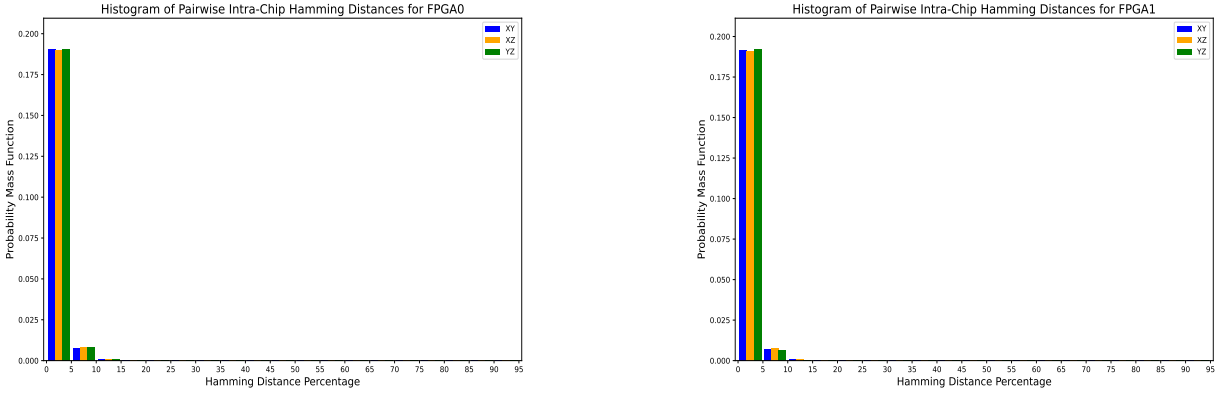


Fig. 10: Histogram of Pairwise Intra Chip HD for FPGA-0 and FPGA-1 using APUF_CmodS7_01.bit

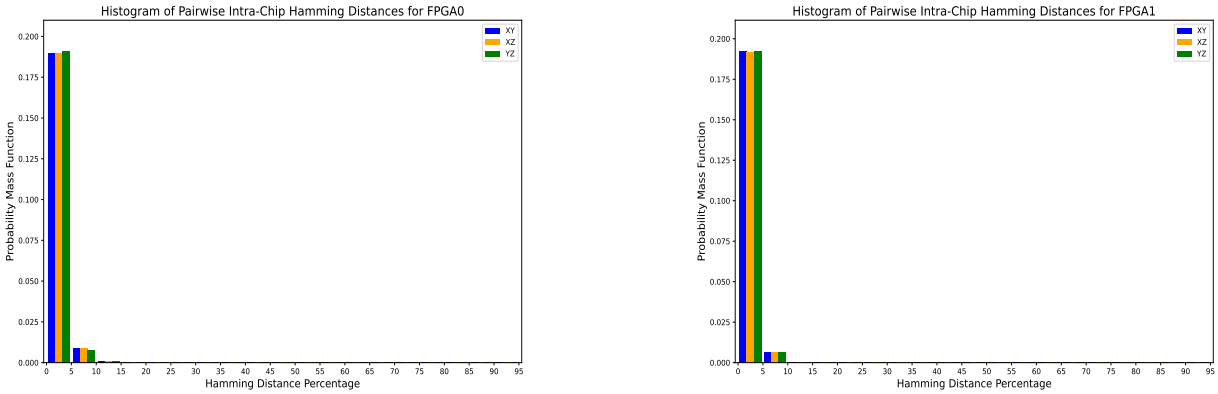


Fig. 11: Histogram of Pairwise Intra Chip HD for FPGA-0 and FPGA-1 using APUF_CmodS7_02.bit

they may produce highly similar responses when presented with the same challenges. This similarity in responses can lead to a low inter-Hamming distance, as the responses share common patterns due to the consistent underlying PUF design. Another possible explanation for the low inter-Hamming distance could be bias in the APUF (Arbiter PUF) design. Bias, in this context, denotes systematic similarities or tendencies in how the APUF reacts to different challenges. If bias exists within the APUF design, it can result in responses exhibiting a consistent pattern across various instances of the APUF. This, in turn, contributes to the observed lower inter-Hamming distances between responses generated by different FPGA boards.

VI. Conclusion

In conclusion, this lab experiment provided us with valuable hands-on experience in working with FPGAs, implementing Physical Unclonable Functions (PUFs), and conducting data analysis. It highlighted the importance of reproducibility, diversity, and security considerations in PUF-based authentication and security systems. Moving forward, we aim to delve deeper into understanding when the observed low inter-Hamming distance can be advantageous and when it might raise security concerns. We are also interested in exploring methods to increase the inter-Hamming distance while preserving other desirable PUF properties. Additionally, we plan to explore various PUF designs and their impact on inter-Hamming distances, which could lead to valuable insights in this field of research.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Experiment 1: PUF Evaluation," in *Practical Hardware Security Course Manual*. University of South Florida, Sep 2023.

[2] ——, "Lecture Note 3: 03 CDA4323 Hardware Security Primitives/Security and Reliability Metrics," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[3] Y. C. N. H. M. CHONGYAN GU, WEIQIANG LIU and F. LOMBARDI, "A Flip-Flop Based Arbiter Physical Unclonable Function (APUF) Design with High Entropy and Uniqueness for FPGA Implementation," *Emerging Topics in Computing*, 2019.

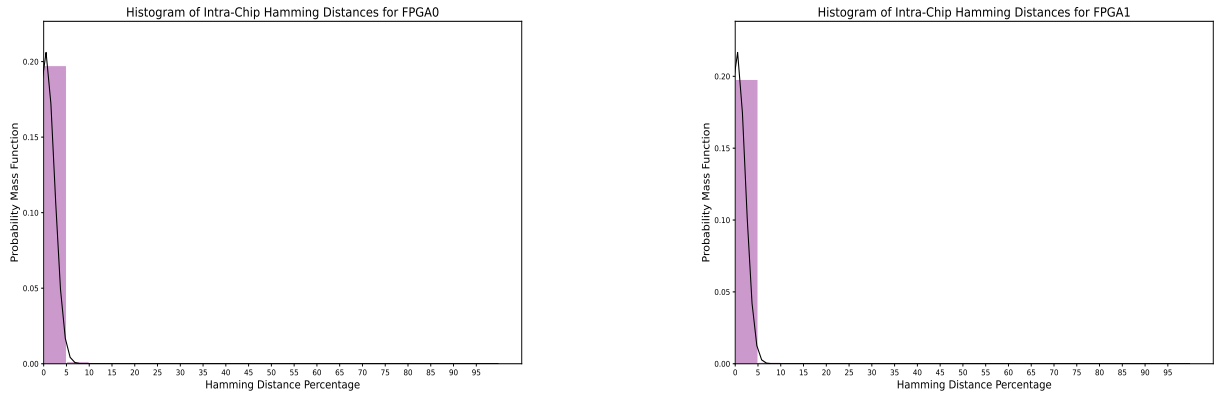


Fig. 12: Histogram of Intra Chip HD for FPGA-0 and FPGA-1 using APUF_CmodS7_01.bit

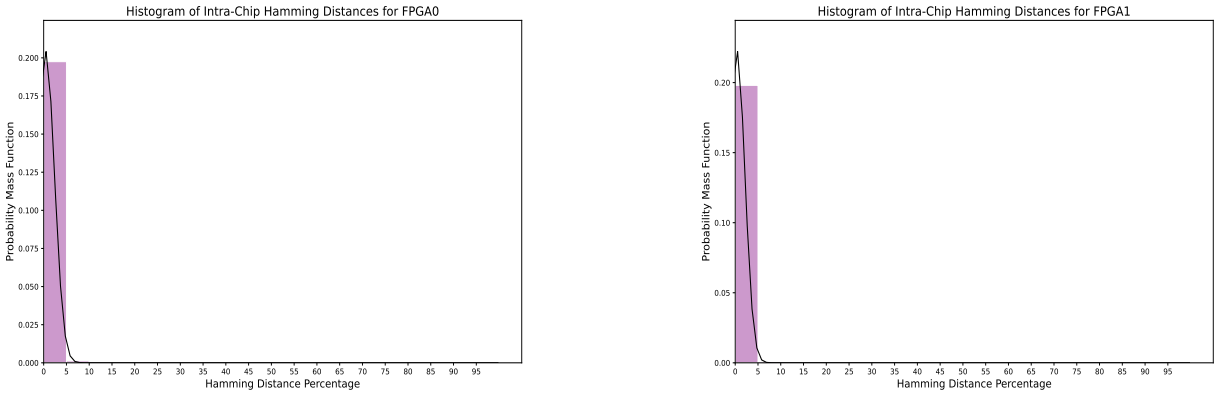


Fig. 13: Histogram of Intra Chip HD for FPGA-0 and FPGA-1 using APUF_CmodS7_02.bit

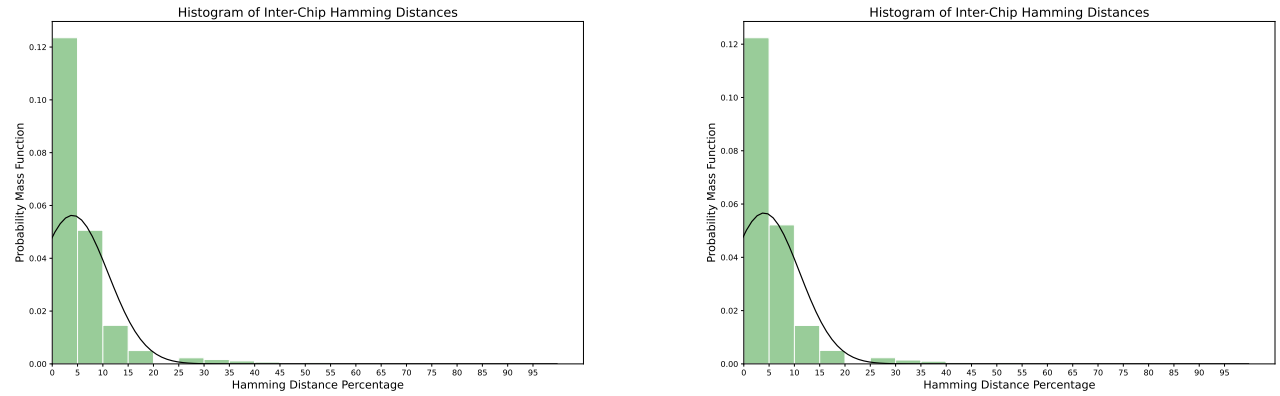


Fig. 14: Histograms of Inter Chip HD using APUF_CmodS7_01.bit and APUF_CmodS7_02.bit